# Writeup Summary

Chase Kanipe

chasekanipe@gmail.com

***Overview*— I decided this writeup needed a summary because the sub-sections ended up being quite long. All the key information is in this summary and each section has a more detailed separate writeup below.**

## I. Ghidra Decompiler

My first task was to examine the workings of the decompiler. There was specific interest in finding anything in the Ghidra code base related to data flow or taint analysis.

The ghidra decompiler is a project mostly separated from the rest of ghidra. It's written in C++, and the rest of ghidra's Java codebase communicates with it over stdin and stdout. I wrote a few scripts to communicate with the decompiler in a headless way so programs can be decompiled from the terminal. The decompiler operates on p-code generated by the SLEIGH engine, and transforms it until it is structured in a form that can be translated more directly to C tokens. The p-code is only in SSA form during the main simplification loop.

## II. Ghidra Decompiler Accuracy

The general consensus is that the ghidra decompiler is superior to its competitors. It far surpasses many popular ones like r2dec and snowman. The only real competitor to it is Hex-Rays (part of IDA), but the ghidra decompiler consistently produces fewer extraneous variables and goto. Some disadvantages are that you can't change data types (you can do this in the ghidra GUI, but it's not part of the decompiler) and that p-code can be difficult to work with.

The decompiler is very accurate for the compilation of basic functions. There are often minor differences like for loops being replaced with while loops, some extra variables, and some issues identifying data types. Generally though, it's accurate enough that (according to some NSA people I talked to), that those who use it rarely look at the disassembly because the decompilation is sufficient.

There are a few places the decompiler consistently fails. For one, it's generally going to decompile structs into several variables, though this can be fixed manually in ghidra. This may not matter for the analysis sake though as long as the programs are equivalent. Ghidra also doesn't do well with obfuscation (there is work being done on this, see here) though I don't think this matters to Correct Computation. It also tends to fail at type identification when there are heavier compiler optimizations.

## III. P-Code

There are two important forms of p-code: the lower level p-code produced by sleigh, and the higher-level pcode produced by the decompiler that is translated into C tokens. The IR isn't actually muti-level, but the two forms are usually structured very differently. The low level p-code is very verbose (for example a shr instruction in x86 corresponds to 30 p-code instructions). Ghidra supports showing the lower level p-code by default and I wrote a plugin to display the higher level p-code.

There are some challenges with p-code. For one, it's based on 20 year old research. It was not designed to be human readable, which can make working with it tricky. There would be some challenges with translating it to LLVM. For one, there are large syntactical differences that make any one to one correspondence between the instructions unclear. Perhaps more significantly, the p-code is not in SSA form except during the main simplification loop while LLVM is. This means that during the translation process the p-code would need to be translated into SSA form. I don't know how hard this would be, but it looks like the translation process overall would be non-trivial.

## IV. Recompilation

While there appear to be some challenges with translating p-code directly to LLVM (though it’s certainly possible), the more indirect route of recompiling the decompiled code to LLVM is a potential alternative. Since the ghidra decompiler is good enough that I could attempt this without much difficulty, I did some basic experimentation. I found that a full decompilation of most programs produced a large number of extraneous functions, but that these were patterned in such a way that they could be filtered out algorithmically. Ghidra would also occasionally fail to identify the return type of a function. I had a lot of success getting the recompilation to LLVM working with smaller programs but I have yet to see if it would work with large applications.

## V. Clang Static Analyzer

Because I had a lot of success recompiling the decompiled code to LLVM I spent some time experimenting with using clang static analyzer on the newly compiled code. Any bugs the analyzer found in the original source code it also found in the recompiled version. In fact, I did find one edge case where CSA found a bug via the decompiled code that it couldn’t in the original source. It is well known that symbolic execution techniques have issues with path explosion in loops. In cases where the compiler optimizations could either unroll or optimize away the loop, CSA then has the potential to find a bug that it was skipping over before. There is a tradeoff, however, because heavier compiler optimizations make the decompilation less accurate - including, in one case, an incorrect calculation of a buffer size.

# Decompiler Design Writeup

Chase Kanipe

chasekanipe@gmail.com

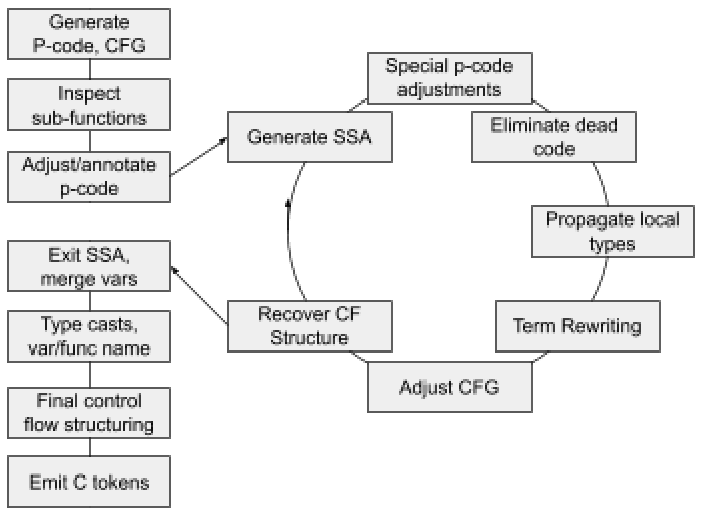
***Overview*— This writeup summarizes the high-level design of the ghidra decompiler. This information was mostly synthesized from the ghidra class, ghidra docs, decompiler docs, SLEIGH docs, this post, this presentation and various other references.**

## I. Introduction

The ghidra decompiler is mostly separated from the rest of ghidra. While the rest of ghidra is written in Java, the decompiler is a standalone C++ project that communicates over stdin and stout using a binary protocol specified in the DecompileProcess class and implemented in the DecompInterface class. Some other ghidra features relevant to this project include the scripting support in Java and Python2 (the API is described here), and headless mode which allows ghidra to be more easily integrated with other tools.

## II. Decompiler Overview

Below is an explanation of ghidra's decompilation process.



1) *Specify Entry Point:* This is a starting address for a particular function.

2) *Generate Raw P-code:* The p-code is generated by the SLEIGH engine. This engine was originally based on SLED (designed by Norman Ramsey and Mary Fernandez, paper here). SLEIGH is a standalone tool designed to be used via it's API. It is designed such that it can be used in a processor independant way. The library can be built independently of the rest of ghidra for integration with other tools, this process is described here.

One of my other writeups is devoted to pcode, but to give a basic example, here's how the MOV instruction translates to the p-code COPY instruction.

```
MOV   RAX,RSI
RAX = COPY RSI
```

Sleigh will also add metadata to the p-code instructions (for example adding parameters associated with a function call, or adding analysis-derived instructions not present in the raw p-code)

3) *Generate Basic Blocks and CFG:* The basic blocks are generated using the p-code instructions and the control flow graph is generated using from the basic blocks.

4) *Inspect Sub-functions:* First ghidra follows each direct call and looks up parameter information. This is repeated for any indirect calls that can be converted to direct calls. Prototype information is either inferred or set to default, but this can be overridden by the user later.

5) *Adjust/Annotate P-code:* The database is used to search for known values of memory locations coming into the function. These are used

by inserting p-code COPY instructions that assign the correct value to the corresponding memory location in the prototype or the beginning of the function.

6) *The Main Simplification Loop:* This is the main loop where p-code is simplified into a form that can be translated to C tokens. **Section III** of this writeup is devoted to details of the main simplification loop.

7) *Perform Final P-Code Transformations:* This phase transforms the output of the main simplification loop enhance readability of the final output and prepare it for the conversion to C tokens.

8) *Exit SSA Form and Merge Low-level Variables:* In this step the static variables of the SSA are merged into higher level variables by exiting the SSA form and eliminating/merging the SSA phi-nodes. Merging must avoid a high-level variable holding different values in several memory locations at the same time. This is related to register coloring in compiler design.

9) *Determine Expressions and Temporary Variables:* In this step the the final expression forms are determined, some variables are forced to be explicit because they are read too often or because making it implicit would propagate another variable too far.

10) *Merge Low-level Variables:* The current form still contains too many variables to translate accurately to C code. More variables are merged in this step.

11) *Add Type Casts:* Type casts are added to the code.

12) *Establish Function Prototypes:* Function prototypes are determined and names are selected or generated.

13) *Select Variable Names:* At this point all the high level variables have been selected, so names are generated for them.

14) *Final Control Flow Structuring:* Switch cases and jumps are determined.

15) *Emit C Tokens:* The final C is emitted by translating the higher level p-code into the corresponding C tokens.

## III. Main Simplification Loop

The main simplification loop translates the lower level p-code to a higher level version that can be more easily translated to C tokens.

1) *Generate SSA Form:* The first step is to generate a static single assignment form (SSA) of the IR. SSA's require that each variable is assigned exactly once and that each variable is defined before it is used. This process normally splits some of the variables in the original IR into several versions. SSA's are normally used because they simplify the application of many compiler optimizations, though in ghidra the process is somewhat different. In ghidra the SSA helps improve constant propagation, dead code elimination, and more.

2) *Dead Code Elimination:* Dead code elimination is essential to the decompiler because a large percentage of machine instructions have side-effects on machine state, such as the setting of flags, that are not relevant to the function at a particular point in the code. The decompiler detects dead code down to the bit, in order to appropriately truncate variables in these situations.

3) *Propagate Local Types:* In this step the decompiler attempts to infer higher level information about the types of the variables. This information is inferred from, for example, computations that the variable is used in and how it is stored in memory.

4) *Perform Term Rewriting:* This section accomplishes most of the simplifications. Following formal methods styles of term rewriting, lists of rules are applied to the syntax tree. This

functions similar to some compiler methods except the goal is simplification rather than optimizations.

5) *Adjust CFG:* Deals with unreachable code, unused branches, empty basic blocks, redundant predicates

6) *Recover CF structure:* The decompiler attempts to recover higher-level control flow objects.

## IV. Data Flow Analysis

Ghidra's data flow analysis capability is one of its key features. It can show where data comes from for any register or variable. IDA has "dumb" text highlighting but it's much less sophiticated than in ghidra. Some source code related to the data flow analysis can be viewed in the GraphAST.java class.

## V. Comparisons

There are many open and closed source decompilers, each of which has it's advantages and disadvantages. The most popular ones include: Snowman, IDA Hex-Rays, r2dec, and of course ghidra. Of these, the only serious competitor to ghidra is Hex-Rays, so I'll focus on that for comparison. Below is a comparison table of the decompiler features.

| **IDA Hex-Rays** | **Ghidra Decompiler** |
|---|---|
| Microcode IR | P-Code IR |
| Limited architectures | Most architectures |
| Variables can be mapped | Variables cannot be mapped |
| Cross references data | No cross references |
| Produces more goto tokens | Produces fewer gotos tokens |
| Can change data types | Can't change data types |
| Rudimentary data flow | Better data flow/slicing |
| Produces more variables | Produces fewer variables |

The general consensus is that the ghidra decompiler is superior to the alternatives. Hex-Rays produces many more extra variables and goto statements in ghidra. I've been told by some NSA people that the ghidra decompiler is good enough that people using it rarely have to look at the disassembly or graph view.

# Decompiler Accuracy Writeup

Chase Kanipe

chasekanipe@gmail.com

**Fibonacci**

Original C

```c
int fib() {
  int n = 30, first = 0, second = 1, next, c;
  printf("First %d terms of the series are:\n",
n);
  for (c = 0; c < n; c++)
  {
    if (c <= 1)
      next = c;
    else {
      next = first + second;
      first = second;
      second = next;
    }
    printf("%d\n", next);
  }
  return 0;
}
```

Decompiled C

```c
undefined8 fib(void) {
  uint local_1c;
  uint local_18;
  uint local_14;
  uint local_10;

  local_1c = 0;
  local_18 = 1;
  printf("First %d terms of the series
are:\n",0x1e);
  local_10 = 0;
  while ((int)local_10 < 0x1e) {
    if ((int)local_10 < 2) {
      local_14 = local_10;
    } else {
      local_14 = local_18 + local_1c;
      local_1c = local_18;
      local_18 = local_14;
    }
    printf("%d\n",(ulong)local_14);
    local_10 = local_10 + 1;
  }
  return 0;
}
```

Decompilation is basically the same as the original C insofar as it's relevant to the analysis. Generates a for loop rather than a while loop, unsigned integers with casts rather than signed integers, casts to a long. Doesn't infer the function return type.

**Decimal to Binary**

Original C

```
int conv() {
  int n, c, k;

  printf("Enter an integer in decimal number
system\n");
  scanf("%d", &n);
  printf("%d in binary number system is:\n", n);

  for (c = 31; c >= 0; c--)
  {
    k = n >> c;

    if (k & 1)
      printf("1");
    else
      printf("0");
  }
printf("\n");
return 0;
}
```

Decompiled C

```
undefined8 conv(void) {
  long in_FS_OFFSET;
  uint local_1c;
  int local_18; // c
  uint local_14;
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  puts("Enter an integer in decimal number
system");
  __isoc99_scanf(&DAT_00101172,&local_1c);
  printf("%d in binary number system
is:\n",(ulong)local_1c);
  local_18 = 0x1f;
  while (-1 < local_18) {
    local_14 = (int)local_1c >> ((byte)local_18 &
0x1f);
    if ((local_14 & 1) == 0) {
      putchar(0x30);
    }
    else {
      putchar(0x31);
    }
    local_18 = local_18 + -1;
  }
  putchar(10);
  return 0;
}
```

Decompilation here is also equivalent to the original code as far as the analysis is concerned. Again, the for loop is replaced with a while loop and a counter, and the ints are replaced with uints with casts.

**Binary Search**

Original C

```
int binarySearch() {
   int c, first, last, middle, n, search,
array[100];

   printf("Enter number of elements\n");
   scanf("%d",&n);

   printf("Enter %d integers\n", n);

   for (c = 0; c < n; c++)
      scanf("%d",&array[c]);

   printf("Enter value to find\n");
   scanf("%d", &search);

   first = 0;
   last = n - 1;
   middle = (first+last)/2;

   while (first <= last) {
      if (array[middle] < search)
         first = middle + 1;
      else if (array[middle] == search) {
         printf("%d found at location %d.\n",
search, middle+1);
         break;
      }
      else
         last = middle - 1;
      middle = (first + last)/2;
   }
   if (first > last)
      printf("Not found! %d isn't present in the
list.\n", search);
   return 0;
}
```

Decompiled C

```
undefined8 binarySearch(void)
{
  long in_FS_OFFSET;
  uint local_1c0;
  uint local_1bc;
  int local_1b8;
  int local_1b4;
  int local_1b0;
  int local_1ac;
  uint local_1a8 [102];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  puts("Enter number of elements");
  __isoc99_scanf(&DAT_00101172,&local_1c0);
  printf("Enter %d integers\n",(ulong)local_1c0);
  local_1b8 = 0;
  while (local_1b8 < (int)local_1c0) {
    __isoc99_scanf(&DAT_00101172,local_1a8 +
(long)local_1b8,(long)local_1b8 * 4);
    local_1b8 = local_1b8 + 1;
  }
  puts("Enter value to find");
  __isoc99_scanf(&DAT_00101172,&local_1bc);
  local_1b4 = 0;
  local_1b0 = local_1c0 - 1;
  local_1ac = local_1b0;
  do {local_1ac = local_1ac / 2;
    if (local_1b0 < local_1b4) {
LAB_00100c5b:
      if (local_1b0 < local_1b4) {
        printf("Not found! %d isn\'t present in
the list.\n",(ulong)local_1bc);
      }if (local_10 != *(long *)(in_FS_OFFSET +
0x28)) {__stack_chk_fail();}
      return 0;
    } if ((int)local_1a8[local_1ac] <
(int)local_1bc) {
      local_1b4 = local_1ac + 1;
    } else {
      if (local_1a8[local_1ac] == local_1bc) {
        printf("%d found at location
%d.\n",(ulong)local_1bc,(ulong)(local_1ac + 1));
        goto LAB_00100c5b;
      }
      local_1b0 = local_1ac + -1;
    }
    local_1ac = local_1b0 + local_1b4;
  } while( true );
}
```

Decompilation is relatively good here. Some extra code constructs and variables.

**Heap Example**

Original C

```
int heapTest() {
   char *str;

   /* Initial memory allocation */
   str = (char *) malloc(15);
   strcpy(str, "tutorialspoint");
   printf("String = %s,  Address = %u\n", str,
str);

   /* Reallocating memory */
   str = (char *) realloc(str, 25);
   strcat(str, ".com");
   printf("String = %s,  Address = %u\n", str,
str);

   free(str);
   return(0);
}
```

Decompiled C

```
undefined8 heapTest(void) {
  char cVar1;
  undefined8 *__ptr;
  char *__ptr_00;
  ulong uVar2;
  char *pcVar3;
  byte bVar4;

  bVar4 = 0;
  __ptr = (undefined8 *)malloc(0xf);
  *__ptr = 0x6c6169726f747574;
  *(undefined4 *)(__ptr + 1) = 0x696f7073;
  *(undefined2 *)((long)__ptr + 0xc) = 0x746e;
  *(undefined *)((long)__ptr + 0xe) = 0;
  printf("String = %s,  Address =
%u\n",__ptr,__ptr);
  __ptr_00 = (char *)realloc(__ptr,0x19);
  uVar2 = 0xffffffffffffffff;
  pcVar3 = __ptr_00;
  do {
    if (uVar2 == 0) break;
    uVar2 = uVar2 - 1;
    cVar1 = *pcVar3;
    pcVar3 = pcVar3 + (ulong)bVar4 * -2 + 1;
  } while (cVar1 != '\0');
  *(undefined4 *)(__ptr_00 + (~uVar2 - 1)) =
0x6d6f632e;
  (__ptr_00 + (~uVar2 - 1))[1] = 0;
  printf("String = %s,  Address =
%u\n",__ptr_00,__ptr_00);
  free(__ptr_00);
  return 0;
}
```

Decompilation is again relatively good here. Several extra variables.

## Structs

Original C

```
struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

int structTest() {
   struct Books Book1;

   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming
Tutorial");
   Book1.book_id = 6495407;

   printf( "Book 1 title : %s\n", Book1.title);
   printf( "Book 1 author : %s\n", Book1.author);
   printf( "Book 1 subject : %s\n",
Book1.subject);
   printf( "Book 1 book_id : %d\n",
Book1.book_id);
}
```

Decompiled C

```
void structTest(void)

{
  long in_FS_OFFSET;
  undefined8 local_e8;
  undefined4 local_e0;
  undefined2 local_dc;
  undefined8 local_b6;
  undefined local_ae;
  undefined8 local_84;
  undefined8 local_7c;
  undefined4 local_74;
  undefined2 local_70;
  undefined local_6e;
  uint local_20;
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  local_e8 = 0x6172676f72502043;
  local_e0 = 0x6e696d6d;
  local_dc = 0x67;
  local_b6 = 0x696c41206168754e;
  local_ae = 0;
  local_84 = 0x6172676f72502043;
  local_7c = 0x755420676e696d6d;
  local_74 = 0x69726f74;
  local_70 = 0x6c61;
  local_6e = 0;
  local_20 = 0x631caf;
  printf("Book 1 title : %s\n",&local_e8);
  printf("Book 1 author : %s\n",&local_b6);
  printf("Book 1 subject : %s\n",&local_84);
  printf("Book 1 book_id : %d\n",(ulong)local_20);
  return;
}
```

It doesn’t appear that ghidra can automatically detect structs at least in the limited way I used them here. Structs can be manually defined in ghidra if need be. However, it may not actually matter for the sake of program analysis whether or not ghidra recognizes the structs as long as the programs are equivalent.

# P-Code Writeup

Chase Kanipe

chasekanipe@gmail.com

***Overview*— This covers various aspects of p-code. I first examine the syntax of p-code, then compare it to other IRs and directly to LLVM.**

## I. Overview

The machine language translation and raw p-code generation is accomplished by the SLEIGH engine. Though it is integrated with the decompiler, SLEIGH can also be used as a standalone library for disassembly and p-code generation as described here. The ghidra decompiler uses it to generate the initial p-code, then manipulates this p-code so it can be translated to the corresponding C tokens.

The p-code instruction set is relatively small. The table below contains a complete list of instructions.

| | | |
|---|---|---|
| COPY | INT_ADD | BOOL_OR |
| LOAD | INT_SUB | FLOAT_EQUAL |
| STORE | INT_CARRY | FLOAT_NOTEQUAL |
| BRANCH | INT_SCARRY | FLOAT_LESS |
| CBRANCH | INT_SBORROW | FLOAT_LESSEQUAL |
| BRANCHIND | INT_2COMP | FLOAT_ADD |
| CALL | INT_NEGATE | FLOAT_SUB |
| CALLIND | INT_XOR | FLOAT_MULT |
| USERDEFINED | INT_AND | FLOAT_DIV |
| RETURN | INT_OR | FLOAT_NEG |
| PIECE | INT_LEFT | FLOAT_ABS |
| SUBPIECE | INT_RIGHT | FLOAT_SQRT |
| INT_EQUAL | INT_SRIGHT | FLOAT_CEIL |
| INT_NOTEQUAL | INT_MULT | FLOAT_FLOOR |
| INT_LESS | INT_DIV | FLOAT_ROUND |
| INT_SLESS | INT_REM | FLOAT_NAN |
| INT_LESSEQUAL | INT_SDIV | INT2FLOAT |
| INT_SLESSEQUAL | INT_SREM | FLOAT2FLOAT |
| INT_ZEXT | BOOL_NEGATE | TRUNC |
| INT_SEXT | BOOL_XOR | CPOOLREF |
| | BOOL_AND | NEW |

P-code operates over varnodes - quoting from the Ghidra documentation: "A varnode is a generalization of either a register or a memory location. It is represented by the formal triple: an address space, an offset into the space, and a size. Intuitively, a varnode is a contiguous sequence of bytes in some address space that can be treated as a single value. All manipulation of data by p-code operations occurs on varnodes."

## II. IR High-Level Comparisons

P-code differs from other IRs in various ways. These are summarized in the table below.

| **P-Code** | **LLVM** |
|---|---|
| - Not easily readable<br>- Single level<br>- Based on 20 y.o. research<br>- Only SSA during decomp | - Human readable<br>- Single level<br>- Based on modern research<br>- SSA for scalar registers |
| **Binary Ninja** | **Hex Rays** |

| | |
|---|---|
| - Human readable | - Not easily readable |
| - Multi-level IR | - Single level |
| - Based on modern research | - Idk; Not well documented |
| - SSA form | - Idk; Not well documented |

The main strength of p-code is that SLEIGH supports many architectures, the integration with ghidra, and that it has some documentation. Unfortunately the design for it was done 20+ years ago, so there are some design differences between it and other IRs. The most relevant non-syntactical difference between it and LLVM is that raw p-code is not in SSA form while LLVM is. This means that if you want to convert from p-code to LLVM, you'll also have to convert it to SSA form.

## III. Direct Comparisons

Un-optimized p-code can be very verbose. Consider the simple C function.

```
int main(void)

{
  printf("%d\n",10);
  return 0;
}
```

The corresponding x86 and raw p-code are shown below.

| **x86 Code** | **Raw P-Code** |
|---|---|
| `PUSH RBP` | `$U2510:8 = COPY RBP`<br>`RSP = INT_SUB RSP, 8:8`<br>`STORE ram(RSP), $U2510` |
| `MOV RBP, RSP` | `RBP = COPY RSP` |
| `SUB RSP, 0x10` | `CF = INT_LESS RSP, 16:8`<br>`OF = INT_SBORROW RSP, 16:8`<br>`RSP = INT_SUB RSP, 16:8`<br>`SF = INT_SLESS RSP, 0:8`<br>`ZF = INT_EQUAL RSP, 0:8` |
| `MOV dword ptr [RBP + local_c], 0x5` | `$U620:8 = INT_ADD RBP, -4:8`<br>`$U1fe0:4 = COPY 5:4`<br>`STORE ram($U620), $U1fe0` |
| `ADD dword ptr [RBP + local_c], 0x5` | `$U620:8 = INT_ADD RBP, -4:8`<br>`$U1fe0:4 = LOAD ram($U620)`<br>`CF = INT_CARRY $U1fe0, 5:4`<br>`$U1fe0:4 = LOAD ram($U620)`<br>`OF = INT_SCARRY $U1fe0, 5:4`<br>`$U1fe0:4 = LOAD ram($U620)`<br>`$U1fe0:4 = INT_ADD $U1fe0, 5:4`<br>`STORE ram($U620), $U1fe0`<br>`$U1fe0:4 = LOAD ram($U620)`<br>`SF = INT_SLESS $U1fe0, 0:4`<br>`$U1fe0:4 = LOAD ram($U620)`<br>`ZF = INT_EQUAL $U1fe0, 0:4` |
| `MOV EAX, dword ptr [RBP + local_c]` | `$U620:8 = INT_ADD RBP, -4:8`<br>`$U1fd0:4 = LOAD ram($U620)`<br>`EAX = COPY $U1fd0`<br>`RAX = INT_ZEXT EAX` |
| `MOV ESI, EAX` | `ESI = COPY EAX`<br>`RSI = INT_ZEXT ESI` |
| `LEA RDI, [DAT_00100704]` | `RDI = COPY 0x100704:8` |
| `MOV EAX, 0x0` | `RAX = COPY 0:8` |
| `CALL printf` | `int printf(char * __format, ...)`<br>`RSP = INT_SUB RSP, 8:8`<br>`STORE ram(RSP), 0x100673:8`<br>`CALL *[ram]0x100520:8` |
| `MOV EAX, 0x0` | `RAX = COPY 0:8` |
| `LEAVE` | `RSP = COPY RBP`<br>`RBP = LOAD ram(RSP)`<br>`RSP = INT_ADD RSP, 8:8`<br>`RIP = LOAD ram(RSP)` |
| `RET` | `RSP = INT_ADD RSP, 8:8`<br>`RETURN RIP` |

As you can see each x86 instruction corresponds to many p-code instructions. x86 instructions like `shr` can translate to as many as 30 p-code instructions.

The decompiler will condense this p-code into a representation more easily translatable into C tokens in the main simplification loop. Ghidra doesn't natively support the displaying of this higher-level p-code, so I wrote a short plugin to do it using the DecompInterface class.

The correlations between this higher level p-code and the C tokens are clear. This higher level p-code also has some type information embedded in it.

**High Level P-code**

```
CALL (ram, 0x100520, 8) , (unique,
0x10000021, 8) , (const, 0xa, 8)

(unique, 0x10000021, 8) COPY (const,
0x100704, 8)

(register, 0x0, 8) COPY (const, 0x0, 8)

RETURN (const, 0x0, 8), (register, 0x0, 8)
```

This same function translated into LLVM produces.

**LLVM Code**

```
define i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %2 = call i32 (i8*, ...) @printf(i8*
getelementptr inbounds ([4 x i8], [4 x i8]*
@.str, i32 0, i32 0), i32 10)
  ret i32 0
}
 -- Typedata annotations removed --
```

It looks like translation between the high-level p-code and llvm would be non-trivial. Translating from  the low level p-code to llvm would probably be easier, but tools already exist for lifting x86 to a low level llvm so I don't see the point of doing that. There is also the added complication of translating the p-code to SSA form. The easier strategy would be to recompile the decompiled code to LLVM.

## IV. Conclusions

P-code is different from LLVM in many ways. The most significant difference for purposes of translation, is that LLVM is in SSA and p-code isn't. This means that to translate p-code to LLVM it will also need to be translated into SSA form.

# Recompiling Ghidra Output Writeup

Chase Kanipe

chasekanipe@gmail.com

One of the goals of this project is to be able to analyze binary files with tools like clang static analyzer. To do this, the binary must first be lifted to LLVM. An alternative route is to just recompile the decompiler output into LLVM. There are, however, complications introduced by the fact that the ghidra decompiler inevitably won't output code in a form that is ready for compilation. This writeup contains information relating to my initial attempts at understanding and automating this recompilation process.

**Fibonacci**

Original C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int fib()
{
  int n = 30, first = 0, second = 1, next, c;
  printf("First %d terms of Fibonacci series are:\n", n);

  for (c = 0; c < n; c++)
  {
    if (c <= 1)
      next = c;
    else
    {
      next = first + second;
      first = second;
      second = next;
    }
    printf("%d\n", next);
  }
  return 0;
}


int main() {
  fib();
}
```

To dump the decompiled code for the whole file I wrote a script that takes advantage of ghidras headless mode so the decompilation process can stay in the terminal. As you can see from the output below, the initial decompiled file contains several hundred lines of extraneous functions and declarations.

Decompiler Output (both columns)

```
typedef unsigned char   undefined;
typedef unsigned char    byte;
typedef unsigned char    dwfenc;
typedef unsigned int    dword;
typedef unsigned long    qword;
typedef unsigned char    undefined1;
typedef unsigned int    undefined4;
typedef unsigned long    undefined8;
typedef unsigned short    word;
typedef struct eh_frame_hdr eh_frame_hdr,
*Peh_frame_hdr;

struct eh_frame_hdr {
    byte eh_frame_hdr_version; // Exception
Handler Frame Header Version
    dwfenc eh_frame_pointer_encoding; // Exception
Handler Frame Pointer Encoding
    dwfenc eh_frame_desc_entry_count_encoding; //
Encoding of # of Exception Handler FDEs
    dwfenc eh_frame_table_encoding; // Exception
Handler Table Encoding
};

typedef struct fde_table_entry fde_table_entry,
*Pfde_table_entry;

struct fde_table_entry {
    dword initial_loc; // Initial Location
    dword data_loc; // Data location
};

typedef struct Elf64_Phdr Elf64_Phdr,
*PElf64_Phdr;

typedef enum Elf_ProgramHeaderType {
    PT_DYNAMIC=2,
    PT_GNU_EH_FRAME=1685382480,
    PT_GNU_RELRO=1685382482,
    PT_GNU_STACK=1685382481,
    PT_INTERP=3,
    PT_LOAD=1,
    PT_NOTE=4,
    PT_NULL=0,
    PT_PHDR=6,
    PT_SHLIB=5,
    PT_TLS=7
} Elf_ProgramHeaderType;

struct Elf64_Phdr {
```

```
typedef struct Elf64_Rela Elf64_Rela,
*PElf64_Rela;

struct Elf64_Rela {
    qword r_offset; // location to apply the
relocation action
    qword r_info; // the symbol table index and
the type of relocation
    qword r_addend; // a constant addend used to
compute the relocatable field value
};

typedef struct Elf64_Ehdr Elf64_Ehdr,
*PElf64_Ehdr;

struct Elf64_Ehdr {
    byte e_ident_magic_num;
    char e_ident_magic_str[3];
    byte e_ident_class;
    byte e_ident_data;
    byte e_ident_version;
    byte e_ident_pad[9];
    word e_type;
    word e_machine;
    dword e_version;
    qword e_entry;
    qword e_phoff;
    qword e_shoff;
    dword e_flags;
    word e_ehsize;
    word e_phentsize;
    word e_phnum;
    word e_shentsize;
    word e_shnum;
    word e_shstrndx;
};
typedef struct evp_pkey_ctx_st evp_pkey_ctx_st,
*Pevp_pkey_ctx_st;

struct evp_pkey_ctx_st {
};

typedef struct evp_pkey_ctx_st EVP_PKEY_CTX;

int _init(EVP_PKEY_CTX *ctx)

{
  int iVar1;
```

```
    enum Elf_ProgramHeaderType p_type;
    dword p_flags;
    qword p_offset;
    qword p_vaddr;
    qword p_paddr;
    qword p_filesz;
    qword p_memsz;
    qword p_align;
};

typedef enum Elf64_DynTag {
    DT_AUDIT=1879047932,
    DT_AUXILIARY=2147483645,
    DT_BIND_NOW=24,
    DT_CHECKSUM=1879047672,
    DT_CONFIG=1879047930,
    DT_DEBUG=21,
    DT_DEPAUDIT=1879047931,
    DT_ENCODING=32,
    DT_FEATURE_1=1879047676,
    DT_FILTER=2147483647,
    DT_FINI=13,
    DT_FINI_ARRAY=26,
    DT_FINI_ARRAYSZ=28,
    DT_FLAGS=30,
    DT_FLAGS_1=1879048187,
    DT_GNU_CONFLICT=1879047928,
    DT_GNU_CONFLICTSZ=1879047670,
    DT_GNU_HASH=1879047925,
    DT_GNU_LIBLIST=1879047929,
    DT_GNU_LIBLISTSZ=1879047671,
    DT_GNU_PRELINKED=1879047669,
    DT_HASH=4,
    DT_INIT=12,
    DT_INIT_ARRAY=25,
    DT_INIT_ARRAYSZ=27,
    DT_JMPREL=23,
    DT_MOVEENT=1879047674,
    DT_MOVESZ=1879047675,
    DT_MOVETAB=1879047934,
    DT_NEEDED=1,
    DT_NULL=0,
    DT_PLTGOT=3,
    DT_PLTPAD=1879047933,
    DT_PLTPADSZ=1879047673,
    DT_PLTREL=20,
    DT_PLTRELSZ=2,
    DT_POSFLAG_1=1879047677,
    DT_PREINIT_ARRAYSZ=33,
    DT_REL=17,
    DT_RELA=7,
    DT_RELACOUNT=1879048185,
    DT_RELAENT=9,
    DT_RELASZ=8,
    DT_RELCOUNT=1879048186,
    DT_RELENT=19,
    DT_RELSZ=18,
    DT_RPATH=15,
    DT_RUNPATH=29,
    DT_SONAME=14,
    DT_STRSZ=10,
    DT_STRTAB=5,
    DT_SYMBOLIC=16,
    DT_SYMENT=11,
    DT_SYMINENT=1879047679,
```

```
  iVar1 = __gmon_start__();
  return iVar1;
}

void FUN_00100510(void)
{
  (*(code *)(undefined *)0x0)();
  return;
}

// WARNING: Unknown calling convention yet
parameter storage is locked

int printf(char *__format,...)

{
  int iVar1;

  iVar1 = printf(__format);
  return iVar1;
}

void __cxa_finalize(void)
{
  __cxa_finalize();
  return;
}

void _start(undefined8 param_1,undefined8
param_2,undefined8 param_3)
{
  undefined8 in_stack_00000000;
  undefined auStack8 [8];


__libc_start_main(main,in_stack_00000000,&stack0x0
0000008,__libc_csu_init,__libc_csu_fini,param_3,
                    auStack8);
  do {
                    // WARNING: Do nothing block
with infinite loop
  } while( true );
}
// WARNING: Removing unreachable block
(ram,0x00100587)
// WARNING: Removing unreachable block
(ram,0x00100593)

void deregister_tm_clones(void)
{
  return;
}

// WARNING: Removing unreachable block
(ram,0x001005d8)
// WARNING: Removing unreachable block
(ram,0x001005e4)

void register_tm_clones(void)
{
  return;
}

void __do_global_dtors_aux(void)
{
    DT_SYMINFO=1879047935,
    DT_SYMINSZ=1879047678,
    DT_SYMTAB=6,
    DT_TEXTREL=22,
    DT_TLSDESC_GOT=1879047927,
    DT_TLSDESC_PLT=1879047926,
    DT_VERDEF=1879048188,
    DT_VERDEFNUM=1879048189,
    DT_VERNEED=1879048190,
    DT_VERNEEDNUM=1879048191,
    DT_VERSYM=1879048176
} Elf64_DynTag;

typedef struct Elf64_Shdr Elf64_Shdr,
*PElf64_Shdr;

typedef enum Elf_SectionHeaderType {
    SHT_CHECKSUM=1879048184,
    SHT_DYNAMIC=6,
    SHT_DYNSYM=11,
    SHT_FINI_ARRAY=15,
    SHT_GNU_ATTRIBUTES=1879048181,
    SHT_GNU_HASH=1879048182,
    SHT_GNU_LIBLIST=1879048183,
    SHT_GNU_verdef=1879048189,
    SHT_GNU_verneed=1879048190,
    SHT_GNU_versym=1879048191,
    SHT_GROUP=17,
    SHT_HASH=5,
    SHT_INIT_ARRAY=14,
    SHT_NOBITS=8,
    SHT_NOTE=7,
    SHT_NULL=0,
    SHT_PREINIT_ARRAY=16,
    SHT_PROGBITS=1,
    SHT_REL=9,
    SHT_RELA=4,
    SHT_SHLIB=10,
    SHT_STRTAB=3,
    SHT_SUNW_COMDAT=1879048187,
    SHT_SUNW_move=1879048186,
    SHT_SUNW_syminfo=1879048188,
    SHT_SYMTAB=2,
    SHT_SYMTAB_SHNDX=18
} Elf_SectionHeaderType;

struct Elf64_Shdr {
    dword sh_name;
    enum Elf_SectionHeaderType sh_type;
    qword sh_flags;
    qword sh_addr;
    qword sh_offset;
    qword sh_size;
    dword sh_link;
    dword sh_info;
    qword sh_addralign;
    qword sh_entsize;
};

typedef struct Elf64_Dyn Elf64_Dyn, *PElf64_Dyn;

struct Elf64_Dyn {
    enum Elf64_DynTag d_tag;
    qword d_val;
};
```

```
  if (completed_7697 != '\0') {
    return;
  }
  __cxa_finalize(__dso_handle);
  deregister_tm_clones();
  completed_7697 = 1;
  return;
}

void frame_dummy(void)
{
  register_tm_clones();
  return;
}

undefined8 fib(void)
{
  uint local_18;
  uint local_14;
  uint local_10;
  uint local_c;

  local_c = 0;
  local_10 = 1;
  printf("First %d terms of Fibonacci series
are:\n",0x1e);
  local_18 = 0;
  while ((int)local_18 < 0x1e) {
    if ((int)local_18 < 2) {
      local_14 = local_18;
    }
    else {
      local_14 = local_10 + local_c;
      local_c = local_10;
      local_10 = local_14;
    }
    printf("%d\n",(ulong)local_14);
    local_18 = local_18 + 1;
  }
  return 0;
}

undefined8 main(void)

{
  fib();
  return 0;
}

void __libc_csu_init(EVP_PKEY_CTX
*param_1,undefined8 param_2,undefined8 param_3)
{
  long lVar1;

  _init(param_1);
  lVar1 = 0;
  do {
    (*(code
*)(&__frame_dummy_init_array_entry)[lVar1])((ulong
)param_1 & 0xffffffff,param_2,param_3)
    ;
    lVar1 = lVar1 + 1;
  } while (lVar1 != 1);
  return;
}
```

```
typedef struct Elf64_Sym Elf64_Sym, *PElf64_Sym;

struct Elf64_Sym {
    dword st_name;
    byte st_info;
    byte st_other;
    word st_shndx;
    qword st_value;
    qword st_size;
};
```

```
void __libc_csu_fini(void)
{
  return;
}

void _fini(void)
{
  return;
}
```

Similar extraneous functions and data are generated by the compiler every time, which allows this data to be filtered out algorithmically. I’ve written a script to do so. The output of running this script on the above output is below.

Filtered Decompiled Code

```
undefined8 fib(void)
{
  uint local_18;
  uint local_14;
  uint local_10;
  uint local_c;

  local_c = 0;
  local_10 = 1;
  printf("First %d terms of Fibonacci series are:\n",0x1e);
  local_18 = 0;
  while ((int)local_18 < 0x1e) {
    if ((int)local_18 < 2) {
      local_14 = local_18;
    }
    else {
      local_14 = local_10 + local_c;
      local_c = local_10;
      local_10 = local_14;
    }
    printf("%d\n",(ulong)local_14);
    local_18 = local_18 + 1;
  }
  return 0;
}

undefined8 main(void)

{
  fib();
  return 0;
}
```

This is much better. But it's not yet in a compilable form. The portions highlighted in red below are causing compilation issues.

Filtered Decompiled Code

```
undefined8 fib(void)
{
  uint local_18;
  uint local_14;
  uint local_10;
  uint local_c;

  local_c = 0;
  local_10 = 1;
  printf("First %d terms of Fibonacci series are:\n",0x1e);
  local_18 = 0;
  while ((int)local_18 < 0x1e) {
    if ((int)local_18 < 2) {
      local_14 = local_18;
    }
    else {
      local_14 = local_10 + local_c;
      local_c = local_10;
      local_10 = local_14;
    }
    printf("%d\n",(ulong)local_14);
    local_18 = local_18 + 1;
  }
  return 0;
}

undefined8 main(void)

{
  fib();
  return 0;
}
```

Again these can be fixed algorithmically. The resulting code compiles successfully.

Final Compilable Code

```
int fib(void)

{
  int local_18;
  int local_14;
  int local_10;
  int local_c;

  local_c = 0;
  local_10 = 1;
  printf("First %d terms of Fibonacci series are:\n",0x1e);
  local_18 = 0;
  while ((int)local_18 < 0x1e) {
    if ((int)local_18 < 2) {
      local_14 = local_18;
    }
    else {
      local_14 = local_10 + local_c;
      local_c = local_10;
      local_10 = local_14;
    }
    printf("%d\n",(long)local_14);
    local_18 = local_18 + 1;
  }
  return 0;
}

int main(void)

{
  fib();
  return 0;
}
```

# Using Ghidra with CSA Writeup

Chase Kanipe

chasekanipe@gmail.com

***Overview*— This report summarizes my attempts to use the ghidra decompiler with clang static analyzer. The most interesting result was that there are some cases where doing analysis on the decompiled code is better than doing analysis on the original source code.**

## I. Some Examples

I went through the process of decompiling some programs and analyzing them with clang static analyzer. I did have to make a few manual edits to some of the decompiled programs to get them to recompile. Some examples are below.

### *A. Stack Vulns*

Due to limited time I only had time to investigate various stack buffer overflow vulns. I ran clang static analyzer on five of the vulnerable programs from here, both on the original source code and on the decompiled form. Any vulns CSA found on the source code it also found in the recompiled versions. It did find vulns in all 5 of the programs I tested, though I could make CSA fail by adding loops or recursion in the right places. I did, however, find some interesting results related to compiler optimization which I'll document in the next section.

## II. Compiler Optimizations

### *A. Advantages*

There are some edge cases where running clang static analyzer on the decompilation from a binary actually finds bugs that it would miss if it had been run on the original source code. Here is one such example.

**Original Code**

```
int main(int argc, char **argv)
{
  volatile int modified;
  char buffer[64];

  if(argc == 1) {
      errx(1, "please specify an
argument\n");
  }

  modified = 0;
  int num = 0;
  for (int i = 0; i < 128; i++) {
    num++;
  }
  strncpy(buffer, argv[1], num);

  if(modified == 0x61626364) {
      printf("you have correctly got the
variable to the right value\n");
  } else {
      printf("Try again, you got 0x%08x\n",
modified);
  }
}
```

It is well known that symbolic execution engines often have trouble dealing with loops due to path explosion. For what I assume is this reason, CSA doesn't find the overflow in the above code.

However, this loop is optimized out of the compiled binary by gcc upon compilation. After this code is decompiled and CSA is run on it, it does find the bug. So surprisingly, even if the original source code is available, there may be some edge case benefits to compiling and decompiling the code prior to analysis.

## B. Problems

There are some problems created by compiler optimizations though. Consider the same buffer overflow vuln.

**Original Code**

```
int main(int argc, char **argv)
{
  volatile int modified;
  char buffer[64];

  if(argc == 1) {
      errx(1, "please specify an
argument\n");
  }

  modified = 0;
  int num = 0;
  for (int i = 0; i < 128; i++) {
    num++;
  }
  strncpy(buffer, argv[1], num);

  if(modified == 0x61626364) {
      printf("you have correctly got the
variable to the right value\n");
  } else {
      printf("Try again, you got 0x%08x\n",
modified);
  }
}
```

I compiled this binary with compiler optimizations enabled (in this case, gcc -O3). The decompiled code looks like this.

**Decompiled Code**

```
int main(char[] param_1,long param_2)
{
  long in_FS_OFFSET;
  char[] local_58 [72];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET +
0x28);
  if ((int)param_1 == 1) {
    errx(param_1,"please specify an
argument\n");
  }
  __strncpy_chk(local_58,*(undefined8
*)(param_2 + 8),0x80,0x40);
  __printf_chk(1,"Try again, you got
0x%08x\n",0);
  if (local_10 != *(long *)(in_FS_OFFSET +
0x28)) {
                    // WARNING: Subroutine
does not return
    __stack_chk_fail();
  }
  return 0;
}
```

The compiler optimizations selected in this case are the highest that gcc allows while still conforming to standard compliance (-Ofast will break standard compliance). It's not surprising that the decompiled C is less accurate in this case; I had to make some manual edits to get it to compile. As you can see from the highlighted portion, the decompiler incorrectly estimates the size of the char buffer, which means it could miss the buffer overflow. In this particular example it still finds it, but if between 65 and 72 bytes were copied, it would miss the overflow.