Function Boundary Detection in Stripped Binaries

Jim Alves-Foss
University of Idaho
Center for Secure and Dependable Systems
Moscow, Idaho, USA
jimaf@uidaho.edu

Jia Song
University of Idaho
Center for Secure and Dependable Systems
Moscow, Idaho, USA
jsong@uidaho.edu

ABSTRACT

Automated cyber defense tools require the ability to analyze binary applications, detect vulnerabilities and automatically patch those vulnerabilities. The insertion of security mechanisms that operate at function boundaries (e.g, control flow mitigation, stack guards) require automated detection of those boundaries. This paper introduces a publicly available function boundary detection tool for 32 and 64-bit Intel binaries running under Linux, that is more accurate than other reported approaches.

CCS CONCEPTS

• **Security and privacy** → *Software reverse engineering*.

KEYWORDS

binary analysis, function boundary detection

ACM Reference Format:
Jim Alves-Foss and Jia Song. 2019. Function Boundary Detection in Stripped Binaries. In *2019 Annual Computer Security Applications Conference (ACSAC '19), December 9–13, 2019, San Juan, PR, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3359789.3359825

1 INTRODUCTION

Automated binary analysis can be used to help detect security vulnerabilities or other flaws in executable binaries without source code, documentation or knowledge of the executable. In addition, as shown in the DARPA Cyber Grand Challenge (CGC) [4, 15, 16], binary analysis can assist in the automatic patching of binaries to help mitigate vulnerabilities. Effective binary analysis starts with disassembly, followed by extraction of the structure and organization of the code: the locations of executable code, read-only data and dynamic data; the locations of functions; and the location and organization of more complex data structures. Some of this information is readily available in the binary's file headers and metadata; information that is not part of the executable, but is needed by the operating system to load and run the program. The rest must be derived from the instructions and data of the program.

Functions are logical constructs in higher level source code that are supported by basic microprocessor instructions and features. A compiler can take the higher level source code and faithfully translate the functions into machine instructions, or can modify the functions. For example, a compiler can take a small function and embed the code for that function within the body of the calling function, called an inline function. A compiler can take a single function and divide it into multiple logical units within the binary, not necessarily contiguous in memory. A compiler can create multiple entry points to a function, merge functions or delete functions. For optimization purposes, a compiler does not need to use the `call` instruction to call a function, but can use a `jump` to directly transfer control. In addition to function organization, a compiler can manage parameter passing using one of many different calling conventions, passing arguments on the main stack, using registers, or even a shadow stack. If the binary uses the stack to pass arguments, it may push them onto the stack as needed, or allocate sufficient stack space at the beginning of the function. Most compilers will store the generated function boundaries in a symbol table, but that table may be stripped from the binary. These variations make automatic function detection in stripped binaries a difficult task that has been studied using basic heuristics[17], graph theory [3], machine learning [6] and neural networks [14].

This paper introduces a publicly available function detection algorithm that takes stripped binaries and returns a list of possible function boundary locations. This algorithm is implemented as part of the Jima tool suite, developed for the CGC for binary vulnerability analysis and repair. We use the same datasets used in prior research [3, 6, 14] to demonstrate that our technique performs well, without the need for extensive machine learning or neural network training. We also evaluate our tools and some related tools against the SPEC CPU 2017 test suite and the Chrome browser. Our results demonstrate that an algorithmic approach, augmented with a few heuristics, can accurately detect function boundaries. This makes this algorithm an enabling technology in support of automated security analysis tools, and vulnerability mitigation technologies such as control-flow integrity [1, 17].

In the remainder of this paper, Section 2 introduces the problem definition and related work. Section 3 introduces the function detection algorithm used by Jima along with some examples for explanation. The evaluation of the algorithm is provided in Section 4. A summary of the experimental results is provided in Section 5. An interpretation of the results is provided in Section 6. This paper concludes with an Appendix that provides more details about the problem definition, notation used and development of "ground truth".

```
474ef0:  subq     $0x8,%rspn
474ef4:  callq    *%rdi
474ef6:  jmpq     404204
474efb:  jmpq     4041fc
```

Listing 1: Example function, with extra `jmpq` at the end.[1,2]

2 PROBLEM DEFINITION

The problem consists of analyzing stripped binaries, those without symbol table or debug information, to determine the list of functions within the binary. Specifically we are looking for *function starts* and *function boundaries*, where function starts are a part of the function boundary, as defined by Bao et al. [6]. This definition assumes all instructions of a function are contiguous in memory, effectively reporting the address of the first and last instruction of a function. For the interested reader, formal definitions of the task goals and the terms precision, recall, F1 values and a discussion of how we determine ground truth are provided in Appendix A.1.

2.1 Example of Discrepancy

There are several cases where the listed function addresses in an unstripped binary, what we consider the *ground truth*, may not coincide exactly with the definitions Appendix A.1. For example, in Listing 1, the compiler includes two jump statements at the end of a function. There is no other jump to the statement at address `0x474efb`; therefore this code is unreachable and technically not an executable instruction of the function. Its existence is probably a compilation artifact that wasn't detected and deleted by optimization. Ultimately we need to determine what is the relevant ground truth. For the purposes of this paper, ground truth is the list of functions in the unstripped binary, as defined in Appendix A.1.

We do claim that if Jima was perfect for function start detection, then these discrepancies are not much of an issue, since we could just define ground truth as just the function starts. To be consistent with prior publications, we decided to follow their lead and report both function starts and function boundaries.

2.2 Related Work

There has been prior work related to function identification in binaries. The recent results summarized here are by Bao et al. [5, 6], Shin et al. [14], and Andriesse et al [3].

2.3 Machine Learning

Bao et al. [6] utilize machine learning techniques to automatically detect function starts and function boundaries. Starting with ground truth, they train their machine learning algorithm, generating specific signatures for each file in the dataset, and analyzed using a 10-fold evaluation. They have made a VM available with their code and datasets [5], which allows researchers to duplicate their results (as long as you add a new virtual drive with over 500GB of disk space for temporary data storage), and to see how they calculated the comparisons. The reported runtime for the full dataset is over 500 hours. We noticed that the test scripts execute 10 versions of the test program in parallel, and therefore may run more efficiently on a machine, and VM, with several CPUs. Given the training data, the actual function start and boundary detection algorithms run faster, with function start taking about 1-2 hours for a 1024 binary dataset and over 100 hours for full function boundary detection. This was an improvement in correctness and efficiency over prior work [13]. Some of the results from this work were then incorporated into the Binary Analysis Platform (BAP) [7], avoiding the need for additional training. We compare the results of Jima with their published results, and BAP in Section 4

2.4 Neural Networks

Shin et al. [14] use a recurrent neural network for their training and function detection algorithms. Unfortunately, they did not provide access to their code. They also used the Bao et al. datasets to evaluate the effectiveness of their results, and compared them directly with Bao et al. published results. Their algorithm improved on the results presented by Bao et al. and is reported to only require a little over 80 hours of training. We include their published results in direct comparison with ours in Section 4.

2.5 Algorithmic

In 2017, Andriesse et al [3] introduced Nucleus, a "compiler-agnostic" function detection algorithm for binaries. They use linear disassembly coupled with in-line data and padding code detection followed by basic block detection. These blocks are then connected into control flow graphs. Based on the premise that intraprocedural control flow instructions tend to be different than interprocedural instructions, they isolate subgraphs of basic blocks for each function. The function detection code of Nucleus is publicly available [12] and therefore was used in our comparisons.

2.6 Other Tools

In 2019, the National Security Agency released Ghidra, a reverse engineering toolkit that includes automated function boundary detection.. We have not seen a published report of how their tool conducts function boundary detection. As a publicly available tool, we have also included a comparison with Ghidra [2]. We also attempted comparison with IDA Free [11] and BAP [7], both publicly available tools.

3 METHODS

The Jima function detection algorithm, as outlined in Listing 2, is implemented in several phases. Each phase requires either a full or partial pass over the code, or its intermediate representation. The phases of the Jima analysis are summarized here, and described in more detail in subsequent subsections:

(1) Disassembly. In this phase, Jima uses `objdump` (a linear disassembler) to generate a listing of assembly instructions. These are then parsed and stored in the Jil data structure, a custom data structure designed to store an abstract representation of the binary and the program (see Section 4.3 for further discussion). During parsing, Jima records all control flow operations (i.e., `call, jump, ret`) along with their target addresses, if explicit in the instruction. Other information about the binary is also collected, including memory

[1]From xgcc in gcc 8.2.0, default 64-bit compilation.
[2]This code is listed in AT&T format, for 64-bit Intel assembly, which lists the target register/address as the last operand of the instruction.

```
Linear Disassembly
  Detect Calls, Call Targets, Returns
  Detect Jumps, Jump Targets
  Detect Embedded data sections

Analyze Exception Handler Tables
Analyze Table-based Jump Ptrs

functionList = callTargets
processFunctionList()

repeat:
  processTerminalCallChain()
  functionList = missingFunctionList
  missingFunctionList = emptyList
until functionList.empty()


where processFunctionList() is:
  while not functionList.empty():
    addr = functionList.pop()
    processFunction(addr)
    if functionList.empty():
      functionList = missingFunctionList
      missingFunctionList = emptyList

where processFunction(addr) is:
  maxAddr = next inflectionOut
  done = False
  while not done:
     addr = nextInflectionOut
     if is_jumpTable_pointer(addr):
       update maxAddr
     elif is_call_to_a_terminal(addr):
       maxAddr = nextInflectionOut
     else:
       apply heuristics
     if addr == maxAddr:
       done = True
       if gap exists:
          add gap to missingFunctionList

where processTerminalCallChain() is:
  for each non-terminal function:
    if calls terminal on all paths:
      set function as terminal
      add callees to missingFunctionList
      remove callees from known functions
```

Listing 2: Jima Function Detection Algorithm

ranges of code and data, explicit references to code and data addresses, and information about dynamic libraries.

(2) Exception Handler Analysis. In this phase, Jima uses data in the exception handler tables to map exception handling code to the parent function, as discussed in Section 3.2.
(3) Jump Pointer Analysis. In this phase, Jima iterates through all jump pointer operations, attempting to detect the location and size of the related jump tables, or jump addresses, discussed in Section 3.3.
(4) Function Detection. In this phase, Jima iterates through all detected call destinations and scans the code forward from the call destination through subsequent *inflection points* (see Section 3.4) until all paths terminate the function or the next call destination is reached.
(5) Missing Function Detection. During previous function detection phases, Jima looks for the existence of executable code in the gaps between detected functions. This phase performs function detection on all possible missing functions, and repeats until no new possible missing functions are found, discussed in Section 3.5.
(6) Terminal Function Call Chain Detection. Any function that does not return from execution, such as the `abort` function, is a terminal function. Software often wraps calls to terminal functions with one or more layers of additional functions, which may implicitly be terminal functions. A call to a terminal function does not return, and therefore the instruction after the call may not be part of the same function as the call. The Jima function detection algorithm detects these implicit terminal functions as part of its analysis, discussed in Section 3.6.

3.1 Detection of Explicit Calls and Jumps

During disassembly, Jil categorizes each instruction. If the instruction is an explicit call or jump (has a hardcoded address), it records the source and target addresses as well as the reverse in lists for 'CALLS', 'JUMPS', 'CALLED_BY' and 'JUMPED_BY'. It also records returns. Initially all calls and jumps are a mapping from the source address to a single target address. However, with jump pointers and call pointers, the targets may be one of a list of addresses. The 'CALLED_BY' and 'JUMPED_BY' lists map the target address to a list of source addresses. These lists are used by Jima when detecting function starts and function boundaries.

Note that in Intel 32-bit position independent code, a compiler can generate a call to the next instruction, which then pops the value into a register, obtaining the current instruction counter. Jima detects this behavior and does not include these calls in the preceding list of calls. Intel 64-bit code has a separate instruction pointer register, and does not need to use this trick.

3.2 Exception Handler Analysis

ELF data files contain two exception handling sections, eh_frame and eh_frame_hdr, which contain information about the exception handlers. The details of these exception handling sections is beyond the scope of this paper, but it is sufficient to say that they contain tables that map regions of instructions to specific exception handlers. When an exception occurs, the tables are searched to find the region containing the current instruction pointer address, and to get the appropriate handler (and possible pre-processing code). The compilers analyzed here (i.e., gcc, icc, clang) append the exception handlers to their parent functions, and include them in the size of the function. Normal control flow analysis will not detect these handlers as part of the parent function, since they are not normal. This is true for linear and recursive analysis and both static and dynamic analysis (unless the dynamic analysis forces and exception to occur). As an added heuristic, Jima decodes these tables to correctly include the handlers within the parent function's boundaries. The other algorithmic tools appear to do this as well, but not the machine learning and neural network tools.

3.3 Jump Pointer Analysis

Jump pointers exist in the code in one of three general forms:

(1) Global offset table. References to the global offset table of the form `jmpq *0x176d9f2(%rip)` or `jmpl *0x8114120` are used to reference functions in shared libraries. The value stored at the referenced address is initially a reference to code that calls the dynamic loader to determine the actual start address of the function in the dynamically loaded library, which is then stored at the referenced location for future use. These tables are stored in the `.plt` (procedure linkage table) section of the file and Jima does not consider them as possible function starting points.

(2) Jump table values. Compilers will often use jump tables to store entry points for different branches of a `switch` statement. The expression is calculated and used to generate an offset into the jump table. The value stored in the jump table is either the specific start address of the relevant `case` statement (see Section 3.3.1), or with position independent code (see Section 3.3.2) is used to calculate the start address of the selected `case` statement.

(3) Other. We have seen other uses of jump pointers, specifically references into a table of functions. This has been used to jump to a called function, where the call is the last statement executed by the current function. This allows the compiler to optimize a sequence of nested calls with a single return to the initial caller. There may be other uses as well.

3.3.1 Use of Jump Tables. Without the existence of jump pointers, function boundary detection would be a much easier problem. Consider the code in Listing 3 which is part of a switch statement.

- Prior to this code, the program evaluates the expression used by the switch statement, and the result is stored in the memory location pointed to by the value in register `%rdx`.
- The instruction at address `0x417021` compares this expression value with `0xb`.
- If the value is greater than 11 (`0xb`), the instruction at location `0x417024` jumps to the default case, or end, of the switch statement. In this case the switch is the last instruction nested in a loop, so the jump is backwards in the code, and not helpful in determining a possible upper bound of the jump targets.
- The expression value is copied into the lower 32 bits of register `%r10d` by the instruction at `0x41702a`[3].
- The instruction at address `0x41702d` looks up a jump target address in the table starting at address `0x495040` (see Table 1). The lookup will be indexed by the value in register `%r10`, multiplied by 8, to step through the 8-byte addresses that represent the memory addresses of the start of the corresponding `case` statements.

[3]64-bit Intel uses the 'd' suffix to indicate the lower 32bits of 64-bit registers that do not have a backward compatible 32-bit register name. In addition, a move to the lower 32 bits will automatically zero out the upper 32-bits.

Table 1: Jump Table reference from Listing 3.

Offset	Bytes in Memory	Jump Address
0x495040	54 35 40 00 00 00 00 00	0x403554
0x495048	44 70 41 00 00 00 00 00	0x417044
0x495050	a8 70 41 00 00 00 00 00	0x4170a8
0x495058	b0 70 41 00 00 00 00 00	0x4170b0
...		

```
417021:  cmpl     $0xb,(%rdx)
417024:  ja       403554
41702a:  movl     (%rdx),%r10d
41702d:  jmpq     *0x495040(,%r10,8)
```

Listing 3: Jump pointer for switch statement[2].

Table 2: Position Independent Jump Table reference from Listing 4. Final address calculated by adding the value in the table to 0x4e54a8 (5133480).

Offset	Bytes in Memory	Hex Value	Jump Address
0x4e54a8	78 fb f8 ff	0xfff8fb78	0x475020
0x4e54ac	d0 fb f8 ff	0xfff8fbd0	0x475078
0x4e54b0	f8 fb f8 ff	0xfff8fbf8	0x4750a0
0x4e54b4	08 fc f8 ff	0xfff8fc08	0x4750b0
...			

3.3.2 Position Independent Jump Table. In Intel 64 bit code, we can use the `%rip` register to generate position independent offsets. The code in Listing 4 and jump table in Table 2 use this technique to reference the start of the jump table and the jump addresses.

- The instructions at addresses `0x474fff - 0x475005` set and check the jump table index, currently in `%cl`, making sure it is in the range 0 to 12, and again, jumping backward in the code, still preventing easy calculation of an upper bound on the jump targets.
- The start of the jump table is loaded into `%r8` at instruction `0x47500b`. That address is calculated by adding the offset (`0x70496`) to the current instruction pointer, `%rip` which contains the address of the next instruction (`0x475012`).
- The value in `%cl` is copied to `%ecx`, zeroing out all of the highbits.[4]
- Line `0x475015` loads the position independent offset for the jump address into `%rax`. This is a negative value.
- Line `0x475019` adds that value to the table start address in `%r8`, storing the result in `%rax`, to be used by the jump at address `0x47501c`.

3.3.3 Jump Table Analysis. To analyze a jump table, Jima starts at the address of the jump pointer and symbolically stores the index register, or other registers, used to calculate the jump. Jima then works backwards through the code, looking for a place where those target registers are set, or at least bounded by a comparison. In the

[4]This includes the high 32 bits in the `%rcx` register. In Intel 64-bit mode, moving a value into a 32-bit register always zero's out the high 32 bits of the corresponding 64-bit register.

```
474fff:  andl     $0xf,%ecx
475002:  cmpb     $0xc,%cl
475005:  ja       404209
47500b:  leaq     0x70496(%rip),%r8
#  The leal instruction puts 0x4e54a8 into %r8. This is
#  offset (0x70496) + %rip (0x475012, which
#  is the address of the next instruction.)
475012:  movzbl   %cl,%ecx
475015:  movslq   (%r8,%rcx,4),%rax
475019:  addq     %r8,%rax
47501c:  jmpq     *%rax
47501e:  xchg    %ax,%ax
475020:  mov     (%rdx),%r8    #  jmp loc 0
475023:  lea     0x8(%rdx),%rax
...
475072:  retq
475073:  nopl    0x0(%rax,%rax,1)
475078:  movq    %rdx,%rax     #  jmp loc 1
47507b:  xorl    %r8d,%r8d
```

Listing 4: Example jump pointer usage with position independent code[1,2]

code of Listing 4, Jima will start looking for a bound on register `%r10`. It will detect the `movl` instruction and replace the current register of interest with `(%rdx)`. Jima will next detect the jump above (`ja`) instruction and note that it was traversing the path where this conditional jump failed. Finally Jima will detect the comparison instruction that set the condition code for the conditional jump.

Once Jima has detected the sequence of instructions that can set the bounds of the jump table, it performs a simple symbolic execution of the code starting from the comparison instruction, `cmpl`. To avoid state explosion problems, Jima stores bounds of values (low and high) for large ranges, and a list of exact values for small ranges (less than 512 values).

- Here Jima records that there was a comparison, and because the jump pointer of interest is on the path where the conditional jump failed, Jima knows that the value in `(%rdx)` is in the range 0-10 (since `ja` is an unsigned comparison, we know both the upper and lower bounds)
- Jima places these values in a list of possible values for `(%rdx)`.
- Jima then continues and copies that list to `%r10d`.
- When Jima reaches the jump pointer, it generates a list of all possible jump table addresses, and dereferences these to get a list of possible jump target addresses. These addresses are then added to the list of jump targets for the source address, and are added to list of jumped_by addresses. These addresses are used in the function detection algorithm as it walks through instructions seeking the last instruction of the function.

3.3.4 Other Aspects of Jump Table Detection. Code is not always executed in a linear fashion, therefore Jima is aware of jumps to code (from the JUMPED_BY list) and will skip back to the jump source if needed. The index register is not always set with a comparison function, but sometimes bounded using logical `and`, or a logical `shift`. In these cases, the register is often set to a limit of a small number of addresses, such as 7, 8, 15, or 16. Sometimes the address into the table is calculated separately and stored in a register which is the operand for the indirect jump. Jima handles all of these cases and a few others. To limit the backward analysis, Jima will stop analysis if it finds the start of the function, finds an instruction that is the target of multiple jumps, or evaluates more than 50 previous instructions without resolution. In this case Jima failed function pointer analysis – it is either not a jump pointer for a switch statement, or the code was just too complex.

3.4 Function Detection

The main function detection process of Jima (see Listing 2) starts with a sorted list of possible function addresses. Initially these are call target found during linear disassembly. Jima then used this list, setting the first address as a possible function start address, and the next one as the next function address. The algorithm then processes instructions from the start address until it reaches the next function address, or reaches a function exit point. There are several key concepts that assist in the function detection process, which are:

- **Inflection Points.** Inflection points are locations where there is a potential for non-linear control flow, and are boundaries for basic blocks. An *inflection-out* is an instruction that leaves the current sequential instruction flow such as a jump, return, or a call to a terminal function. An *inflection-in* is an instruction that is the target of a call or jump. These inflection points are usually the bounds of basic blocks discussed in compiler and reverse engineering literature.
- **Jump tables.** A jump pointer can use a jump table to select one of several possible jump targets. Jima assumes all jump targets are within the bounds of the current function.
- **Terminal Functions.** A call to a terminal function is marked as an inflection-out that does not return.

The Jima function detection algorithm processes function code using inflection-out points. Jima sets the max function address as the next inflection-out and then iterates through the inflection-out points until it finds a terminating case. If the inflection-out point is a jump pointer for a discovered jump table, the algorithm assumes that the target addresses are within the current function, and sets the max address to the maximum of the jump targets and the current max address. If the inflection-out is a call to a non-terminal function the algorithm processes a few heuristics to determine if the subsequent code is still part of the function, and if it is, the max address is set to the max of the current max address and the next inflection-out point. These heuristics include the following.

- Embedded data belongs to the preceding function.
- After disassembly, Jima scans all targets of calls. If over 90% of them are aligned to a specific byte boundary, Jima takes that into account when determining if the next instruction (and corresponding basic-block) is still part of the existing function or a new function.
- If the last detected instruction is followed by a jump over a sequence of NOPs, that is assumed to be the end of the function. In addition, if the instruction is followed by several NOPs then Jima assumes it is at the end of the function.

- All exception handling code marked for instructions in the current function, belong to the current function.

3.5 Missing Function Detection

After the end of the function, if there is a gap to the next known function start, then the next executable address is stored in a list of possible missing functions. After the current list of function starts is completed, Jima restarts the process with the list of missing functions, until there are no more possible missing functions.

3.6 Terminal Function Detection

The Jima function detection algorithm currently starts by assuming the following list of glib functions are terminal (non-returning) functions: `abort, exit, Exit, errx, err, __assert_fail, _exit` and `__stack_check_fail`[5]. The algorithm looks at each caller of terminal functions. Each calling function start address is added to the list of possible missing functions, and is removed from the list of previously functions. Subsequent reanalysis of this function may now change the detected function boundaries due to the call to the newly recognized terminal function; possibly resulting in new terminal functions and additional missing functions. This process is repeated until no new terminal functions are found.

4 METHODS

This section summarizes the experimental set up for the empirical results obtained when evaluating Jima with respect to other work. We ran our experiments on an Intel(R) Core(TM) i7-5820K CPU running at 3.30GHz. This processor has 12 logical cores and 32GB of RAM. The machine was running Ubuntu 18.04 LTS. The evaluated Jima code was written in Python3.

4.1 Datasets

To compare our work with prior work, we use three datasets. Statistics for these datasets is provided in Tables 3 & 4.

Unix Utilities. The first dataset comes from Bao et al. [6]; which was also used by Shin et al. [14], Andriesse et al. [3] and Di Federico et al. [10]. This dataset consists of 2064 Linux binaries derived from 129 unique programs (see Table 3). Bao et al. built this dataset from the 129 programs from the binutils (16 programs), coreutils (104 programs) and findutils (9 programs) packages. Each program was compiled using four different optimization levels (levels O0 through O3) using three different compilers, gcc, Intel icc, and clang[6] into both 32-bit and 64-bit executables. The dataset and documentation were made available on-line by Bao et al. [5].

These tools are relatively small, do not contain exception handlers, and do have shared code within the tool suites. Due to this, we have concerns about the generalizability of the results that were also raised by Andriesse et al. [3]. To summarize, the tool suites have shared code basis, and the tools are small. Also, for the machine learning experiments [6], variants of each utility (compiled under different compiler or with different optimization option) were in the training set. However, since this dataset has been used by several prior authors, [3, 6, 10, 14], we include it for completeness.

SPEC CPU 2017. The second dataset utilizes the SPEC CPU 2017 [9] benchmarks. This data set consists of a range of different programs built using C, C++ and Fortran. We used the INTSPEED, INTRATE, FPSPEED and FPRATE benchmarks. We removed duplicates binaries that were named differently for performance benchmarking purposes. This resulted in 28 benchmarks that compiled for 32-bit executables and 32 benchmarks that compiled for 64-bit executables. We built these with the four optimization options (level O0 to O3) using the gcc, clang and icc compilers. For both gcc and clang datasets we used the gfortran compiler, and for icc we used the ifort compiler. We had to set the -fPIC flag to build several of the binaries. The -fPIC flag forces the compiler to generate position independent code. Therefore this dataset evaluates tool performance on position independent code, Fortran code and exception handling. Note that some benchmarks did not compile for all options. We felt there was sufficient diversity in the dataset that we did not correct for this.

Chrome Browser. The final dataset is actually just one program, the Chrome browser, built with the default options. This generated a 99 MB file that contains over 353,000 unique functions. An examination of the symbol table showed many function names for the same address; all duplicates were removed in calculation of the ground truth in the work presented here.

4.2 Ground Truth

Ground truth for each of the binaries was determined by examining the symbol table within the unstripped version of the binary, looking for function entries in the '.text' section. There are a few nuances related to the use of ground truth data when comparing our results with that of other techniques, these are detailed in the Appendix A.2. This definition of ground truth for function starts and function boundaries assumes that all bytes of a function are stored consecutively in the binary, and that there is no code shared between functions.

4.3 Implementation

The Jima tool suite is written in Python3. Although the use of Python may limit performance, it also enables collaboration, portability and extensibility. The Jima binary analysis and repair activities center around the Jil data format, a custom data structure that maintains information about the contents of a binary. The Jima tool suite includes several functions and utilities to examine, evaluate and modify the Jil data structure. For reuse, Jima stores the Jil data structure in a pickled (compressed) file. For the purposes of this paper, we use JLift, the Jima tool that performs first phase disassembly and categorization of the binary and instructions, creating the initial Jil data structure. We then run JilTool, which conducts jump pointer, dynamic library, function pointer and embedded data analysis, followed by function detection[7].

To determine the efficiency of our results, we generate a ground truth file, as discussed in Section 4.2. We then ran the Jima, Ghidra [2]

[5]Future plans will enable us to evaluate dynamically loaded libraries to automatically detect terminal library routines, starting with knowledge of terminal system calls, and then build up from there.

[6]We extended this data set by compiling the programs with clang compiler.

[7]We have posted our work as an ACSAC artifact, and also on GitHub at https://github.com/CenterForSecureAndDependableSystems/FunctionBoundary.

Table 3: Characteristics of Unix utility datasets

	Unix Tools					
	ELF x86			ELF x86-64		
	clang	gcc	icc	clang	gcc	icc
Number of Binaries	508	516	516	508	516	516
Number of Functions	172,131	145,279	164,139	170,932	146,153	154,204
Avg. Number of Functions	339	281	318	336	283	299
Size of Stripped Binaries (MB)	101	84	102	110	87	128

Table 4: Characteristics of Other datasets

	SPEC CPU 2017						Chrome
	ELF x86			ELF x86-64			ELF x86-64
	clang	gcc	icc	clang	gcc	icc	clang
Number of Binaries	108	112	110	123	128	128	1
Number of Functions	461,102	564,927	672,483	527,702	645,893	735,507	353,301
Avg. Number of Functions	4,272	5,044	6,113	4,290	5,046	5,726	353,301
Size of Stripped Binaries (MB)	411	394	509	1,087	1,316	1,012	99

and Nucleus [12] against each of the datasets. We also ran IDA Free 7.0 [11], and BAP 1.50 [7] against some of the datasets. Other results are taken from those self-reported by the authors of the tools.

5 SUMMARY OF RESULTS

5.1 Unix Utilities DataSet

A direct comparison of Jima function detection algorithm, Nucleus [3, 12] and Ghidra [2] to the published results of Bao et al. [6] and Shin et al. [14] is presented in Table 5. We show average percentages of precision, recall and F1 over specific datasets, to be consistent with the format or results shown in prior work [3, 6, 14]. Although a useful first pass, this does not provide a detailed comparison as we discuss in Section 6. We do not break down our results by optimization level, since we found limited use of that data in all but the icc compiler (see Section 5.3).

For the most direct comparison, we ran our experiments for Ghidra, Nucleus and Jima on all of the reported datasets, BAP and IDA Free on a limited subset due to some execution issues. We also used the same gcc and icc compiled datasets provided by Bao et al. [5]. We report the results from Bao et al. and Shin et al. prior work as reported by the authors.

For function start detection, Jima performs slightly better than the prior work. For function boundary detection, Jima results are several percentage points above prior work, especially in 64-bit mode, and on average is better than Ghidra or Nucleus. We present these results with a few caveats:

- Bao et al. [6] consider a function boundary to be a match if the detected end is not beyond the real end, therefore a short function is a match. We do not know how Shin et al. [14] calculated matches. The compilers, especially icc, sometimes include padding NOPs in the function length, which probably explains the approach used by Bao et al. For the experiments we ran, we used an exact match to define success.
- Shin and Byteweight tools were developed with evaluation on only two compilers, and we were not able to rerun, so we use their reported values.
- IDA Free does not have a scripting capability that we could use, unlike Ida Pro, so we manually ran a limited set of tests to see if the tool's results were comparable.
- BAP took a very long time to run on some of the data sets, and had difficulty on some of the tests. Also, it tended to return all possible bytes in a function, resulting in overlapping function boundaries.

As can be seen by these results, Jima performs consistently across all types of files, with slightly worse performance for icc. When combined together, the results are very encouraging and led to the writing of this paper.

5.2 Other DataSets

In addition to the Unix Tools dataset of Bao et al., we used SPEC CPU 2017 and Chrome[8] to test more complex and larger code, including position independent code and exception handling. The F1 function start and function boundary results are summarized in Table 7 for IDA 7.0 free, Nucleus, Ghidra 9.0.1. and Jima for each compiler and architecture. We only ran IDA Free on the gcc version due to difficulty in obtaining the data, and the fact that the results were much worse than Jima. For some of the datasets BAP required too much memory and either crashed or took too long to complete.

As can be seen from the results in Tables 5- 7, Jima consistently performed better than IDA and Nucleus on the Unix utilities and SPEC datasets. Jima also performed well on Chrome.

5.3 Intel icc

The icc benchmarks were a bit disappointing, and require further explanation. We wondered if low numbers were a byproduct of optimization level, or possibly numbers of functions. Some of the

[8]Chrome was compiled using the default compilation options using clang compiler in 64-bit mode.

Table 5: Summary of Function Start/Function Boundary identification results compared with previous work, Byteweight* and Shin* are self-reported values for gcc and icc; Nucleus, Ghidra and Jima are averaged across all 3 compilers. Jima improvement is compared to best of other tools.

	UNIX Utilities Data Set % Precision, Recall and F1					
	ELF x86			ELF x86-64		
	Precision	Recall	F1	Precision	Recall	F1
Byteweight*	98.41/92.78	97.94/92.29	98.17/92.53	99.14/93.22	98.47/95.52	98.80/92.87
Ghidra 9.0.1	100.00/97.35	65.72/64.68	75.31/73.88	99.98/98.44	97.94/96.46	98.94/97.43
Nucleus 0.65	91.20/95.58	93.18/89.40	91.94/92.24	97.68/97.33	93.79/91.40	95.59/94.16
Shin* et al.	99.56/97.75	99.06/95.34	99.31/96.53	98.80/94.85	97.80/89.91	98.30/92.32
Jima	99.21/97.93	99.46/98.19	99.33/98.05	99.81/99.30	99.38/98.86	99.59/99.07
Jima % Improvement	-0.79/-0.43	0.40/2.85	0.02/1.02	-0.11/0.86	0.91/2.40	0.65/1.64

Table 6: Summary of average Function Start/Function Boundary detection F1 values for all optimization levels by compiler for the Unix utilities dataset. Jima improvement is compared to best of previous 3 tools. Byteweight* and Shin* data are self-reported values included for reference.

	UNIX Utilities Data Set % F1					
	ELF x86			ELF x86-64		
	clang	gcc	icc	clang	gcc	icc
BAP 1.5.0	64.09/30.91	97.28/80.42	N/A	56.80/38.02	90.49/71.38	83.63/68.67
Ghidra 9.0.1	69.98/68.79	99.26/99.14	56.60/53.63	99.30/98.51	99.36/99.22	98.14/94.50
Nucleus 0.65	97.15/96.29	96.31/97.36	82.44/83.13	85.11/95.57	92.52/88.63	85.63/91.85
Jima	99.96/99.90	99.78/99.36	99.96/99.90	99.88/99.76	99.96/99.87	98.93/97.60
Jima % Improvement	2.81/3.61	0.52/0.22	17.55/16.77	0.58/1.25	0.60/0.65	1.79/3.10
Byteweight*	98.17/92.53			98.80/92.87		
Shin* et al.	99.31/96.53			98.30/92.31		

Table 7: Summary of average Function Start/Function Boundary detection F1 values for all optimization levels by compiler for the SPEC 2017 CPU dataset, and Chrome. Jima improvement is compared to best of other tools. IDA was only tested on the gcc compiled version and is included for reference, BAP failed to complete for these.

	SPEC CPU 2017 % F1						Chrome % F1
	ELF x86			ELF x86-64			ELF x86-64
	clang	gcc	icc	clang	gcc	icc	clang
IDA 7.0 (Free) 1.5.0		82.42/75.57			80.59/74.69		88.67/82.63
Ghidra 9.0.1	88.39/80.33	99.16/96.58	48.39/42.54	97.30/92.43	99.18/96.28	83.61/68.87	99.97/99.51
Nucleus 0.65	74.94/35.42	72.26/34.16	72.67/64.74	91.93/87.50	97.61/95.55	65.71/63.00	95.01/52.92
Jima	98.72/98.46	99.80/99.21	97.59/88.27	99.94/99.90	99.99/99.98	99.27/92.54	99.04/97.25
Jima % Improvement	10.33/18.13	0.64/2.63	24.92/23.53	2.64/7.47	0.81/3.70	15.66/23.37	-0.93/-2.26

binaries had low numbers of functions, so a small number of errors could result in a relatively large percentage decrease. We therefore plotted the F1 values for all of the icc compiled binaries for Jima in Figures 1 and 2. These values are sorted by optimization level and then by the number of functions[9]. There are some obvious artifacts in the results for O2 and O3 optimizations. We can see a slope up as percentages improve as the number of functions increases, which shows that a few errors in function detection can affect the overall results. The existence of two distinct slopes in these regions is due to the nature of the data sets. The lower slope is primarily the coreutils dataset, and due to some common functions among those utilities were compiled in a way that Jima analyzed incorrectly. The scattered lower values, specifically in Figure 2 are due to some binaries in the SPEC CPU dataset. Jima did not accurately detect the end of the functions, creating shorter functions compared to ground truth, and also creating some extra false positives. This is mostly due to NOPs padding the end of some functions, but counted by the icc compiler as part of the function, but not counted by Jima. We decided to not correct for this when reporting results here, because we could not find evidence that prior art ignored these NOPs, and still gives us a lower bound on correctness.

[9]These graphs were created to highlight the differences, therefore the y-axis scale starts around the minimum value.

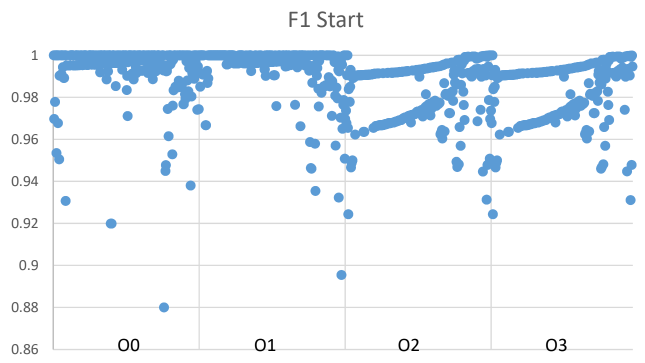

Figure 1: Jima F1 Start values for icc compiled binaries sorted by optimization level and number of functions.

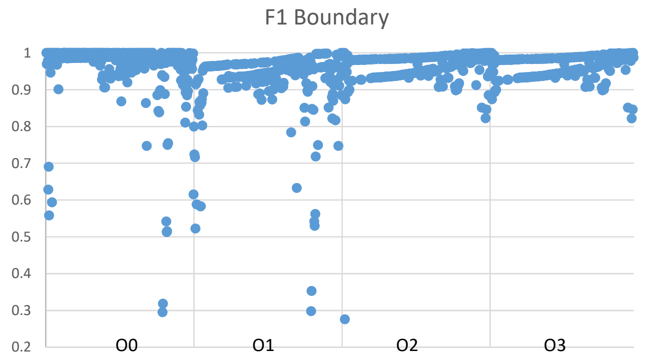

Figure 2: Jima F1 Boundary values for icc compiled binaries sorted by optimization level and number of functions.

Table 8: Execution Time and Space for Analysis of Chrome

Tool	Run Time (sec)	Max Mem (GB)
Nucleus	245.36	10,980
Jima	666.46	27,510
Ghidra	6,066.59	2,512

5.4 Computation Time and Space

The most accurate and time efficient tools appear to be Ghidra, Nucleus and Jima, and therefore we compared their performance on Chrome. We wanted to know how well they perform on a 99MB executable with a large number of functions. The results are shown in Table 8. Nucleus [12] runs quite efficiently and with some control for memory overhead; the posted code being specifically used for just function boundary detection. Ghidra runs slower but much more beneficially in terms of RAM usage, and performs a range of analysis – although we tried to limit the analysis for our experiments to limit it to function boundary detection. Not included on this table: IDA Free ran twice as slow as Jima, and BAP ran much slower and took more memory – crashing on analysis of Chrome.

In a review of the performance of Jima, we found that the majority of time is taken in disassembly and pre-processing. Since Jima uses `objdump`, there is additional overhead in reading the textual output and converting it into the Jil internal representation. A direct integration of a disassembler would greatly speed up this part of the program, and is slated for future work.

6 DISCUSSION

6.1 Averages vs. Details

The results summarized in Tables 6 and 7 are averages over many different test programs, but miss details such as that seen in Figures 1 and 2. For example, if we look at the x86 gcc data for Unix utilities, Jima compares favorably to Ghidra (Table 6). However, when we plot the difference in F1 function boundary values for each executable, we can see details that averages miss (see Figure 3). In this figure, binaries are sorted by utility group (binutils, coreutils, findutils), then by optimization levels O0-O3 and then alphabetically by tool name. This grouping shows that Jima does worse on function boundary detection on some of the binaries, specifically those compiled with coreutils O2 and O3 optimizations, or findutils when compared to Ghidra. However, for some cases Jima does significantly better than Ghidra. When plotted this way, we see that the results are quite mixed and that there are some trade-offs between approaches, indicating a need for further investigation.

To get a better feel for the overall results, instead of just relying on averages, we calculated the number of binaries where the F1 values for function start or function boundary detection were better, equal or worse for Jima with respect to Ghidra (Table 9) and Nucleus (Table 10). Here it is much more apparent that Jima provides better results in almost all cases except with function boundaries in gcc compared to Ghidra.

6.2 Error Analysis

There are several things that limit the effectiveness of our approach, some of which are discrepancies with ground truth. There are times our algorithm correctly finds all executable instructions of a function, but the listing in the unstripped binary includes additional instructions or padding NOPS. In addition there are limitations to our algorithm. The first is the incompleteness of the jump table detection process. Although our heuristics are pretty accurate, they are not able to calculate all possible values of jump table pointers. Second is the use of jump pointers that are not part of a jump table, and call pointers, that may not be accurately detected. Third is the use of call pointers and tables of function addresses, Jima does not address these at all.

6.3 Algorithmic Comparison to Prior Art

In the previous sections we have discussed how our algorithm works, and compared our results to that of prior art. Naturally, this is the part of the paper where we explain why it works better than prior art. That may be harder to do than it appears on the outset.

The machine learning [6] and neural network [14] approaches are based on automated learning of patterns in the code. These techniques both require a large amount of training, and extensive, representative datasets. However, they attempt to learn to match

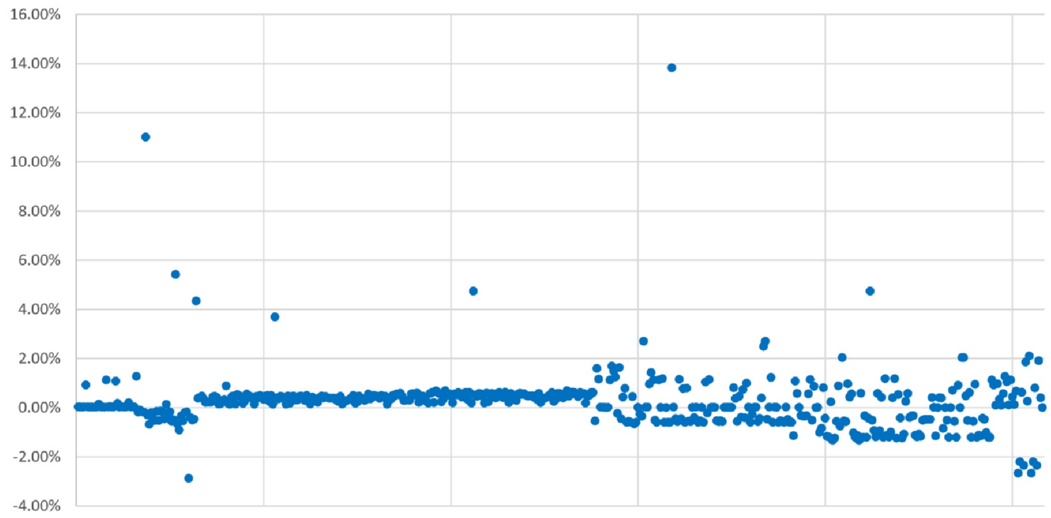

Figure 3: Jima improvement over Ghidra, comparison of F1 function boundary for Unix dataset, sorted by utility group, then optimization level.

Table 9: Count of Binaries for Jima Improvement/Equal/Loss over Ghidra for F1 Metric of Function Starts and Boundaries by Compiler for x86 and x86-64 Unix Utilities and SPEC CPU 2017

	Jima vs Ghidra					
	gcc		icc		clang	
	Start	Bound.	Start	Bound.	Start	Bound.
# Binaries where Jima F1 is Higher	1,116	944	1,269	1,266	1,228	1,245
# Binaries where Jima F1 is Equal	156	21	0	0	10	0
# Binaries where Jima F1 is Lower	0	306	1	4	9	2
Jima's Average F1 Score Improvement	0.5%	0.8%	23.36%	24.62%	13.65%	15.57%

Table 10: Count of Binaries for Jima Improvement/Equal/Loss over Nucleus for F1 Metric of Function Starts and Boundaries by Compiler for x86 and x86-64 Unix Utilities and SPEC CPU 2017

	Jima vs Nucleus					
	gcc		icc		clang	
	Start	Bound.	Start	Bound.	Start	Bound.
# Binaries where Jima F1 is Higher	1,266	1,221	1,232	1,206	1,237	1,231
# Binaries where Jima F1 is Equal	6	3	18	3	10	0
# Binaries where Jima F1 is Lower	0	48	20	61	0	16
Jima's Average F1 Score Improvement	2.3%	3.4%	25.18%	27.92%	4.71%	10.71%

patterns of code at the end of a function. Although they are independent of the semantics of the code, this also means they can not take advantage of those semantics. One major issue would be in evaluation of code with exception handlers. The compilers we investigated appended exception handlers to the end of a function, after the normal "function termination" code; a pattern matching technique will often miss these. As a benefit, these approaches can handle data embedded within the instructions, since they tend to overlook internals of functions.

Tools that focus primarily on recursive descent disassembly, such as that used by BAP [7] and Ida-Pro [11] have two major issues. We found with BAP, that if a compiler optimization replaced a function call with a jump, which occurs when the call is the last instructions of the function, the tool may assume the jump target and all of the intermediate code is part of the same function. We have found in BAP that sometimes the tool actually reported bytes as being part of two different functions. The tools may also miss some function starts or jump pointer targets. Combining the concepts of recursive descent along with linear disassembly and some selective heuristics gave us a better approach.

As for Nucleus [3] and Ghidra [2], they both seem to use a more algorithmic approach to function detection. Nucleus uses a graph-based approach to bring basic blocks of code together into a functional unit. This worked well on gcc and clang, however, Nucleus struggled on icc compiled data sets. We also found test cases that Nucleus could not handle, which can be traced back to the inability of the Capstone [8] disassembler to correctly disassemble the code. We are unsure of the algorithmic approach used by Ghidra, however, it seems to be optimized for gcc and struggles with clang and icc.

When we look at the compilers, instead of the analysis algorithms, we see that most tools performed well on gcc. We believe this is due to the fact that gcc is very commonly used compiler and therefore is the focus of much examination and research. Also, the generated code is a bit more structured and straight-forward compared to the other compilers. The Intel compiler, icc, gave all tools the most difficulty, which is partially due to the inclusion of embedded data in included library code. We also found that icc generated code for function pointer calculation was not as standardized as in the other compilers.

Jima uses a combination of approaches to detect function boundaries, as described in Section 3. Jima first uses the linear disassembly from `objdump` and parses that to get a list of call targets. This is used as the initial set of functions. If the file contains exception handler tables, Jima then gathers information from those tables, mapping section handlers to function code, grouping it all together as a first pass at function boundaries. Jima then takes the initial function start addresses and performs a recursive descent on those functions, gathering sections of code into the same function as it progresses. If Jima encounters a jump pointer, it tries to calculate the start and end of the jump table in memory, and the resulting jump addresses. This approach is similar to the approach taken by Zhang and Sekar [17], although developed independently. During analysis, Jima also looks for calls to terminal functions. This heuristic allows the tool to treat these calls as exist conditional for the recursive descent. Finally, Jima detects unassigned instructions, those not mapped to a function, as new possible instruction starts. Jima takes these new starts and the newly detected terminal functions as feedback into the process, and reruns the recursive descent analysis as needed.

This combination of linear disassembly and analysis, recursive descent analysis, jump pointer and terminal function detection with feedback allows Jima to develop a reasonable semantic model of the control flow of the program under analysis.

7 CONCLUSION

This paper presented the function detection algorithm implemented in Jima, a tool developed for binary vulnerability analysis and repair. Using static analysis, and limited behavioral analysis, the algorithm is able to consistently detect function starts and bounds better than existing tools or prior publications. The run-time performance of the solution is very good, allowing for quick analysis of programs. There are still several areas of future improvement for Jima and the function detection algorithm:

- We can improve performance by integrating disassembly into Jima and not rely on parsing textual output from `objdump`.
- There are some edge cases for PIC code on jump tables that need to be addressed. This involves improvement in the limited symbolic execution subroutines.
- The definition of ground truth needs to be improved. We plan to analyze binaries to look for extra function labels inserted by the optimizing compiler, but are never called. We also plan to look at the listed ground truth for function length, detecting trailing NOPs or unreachable code. The ultimate goal is to get function starts and boundaries that can be best used for control flow integrity [1]. Functions that are never called, and code that is never executed, should be excluded from the set of authorized locations with a binary.
- The current tool only works on Linux ELF binaries. We plan to port the tool to work on Windows PE binaries.

8 ACKNOWLEDGMENTS AND AVAILABILITY

This work was funded in part by the Idaho State Board of Education through grant IGEM17-001 and also qualifying phase prize money from the DARPA Cyber Grand Challenge. All results presented here are the opinions of the authors and not the sponsors. Jima is being released under the GPL license at our github site https://github.com/CenterForSecureAndDependableSystems/FunctionBoundary. Datasets, scripts and the Jima executable are available as ACSAC submitted artifacts.

REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, , and J. Ligatti. 2008. Control-flow integrity—principles, implementations, and applications. *ACM Transactions on Information and System Security* 13, 1 (2008), 1–40.
[2] National Security Agency. 2017. Ghidra Reverse Engineering Tool. (2017). https://www.nsa.gov/resources/everyone/ghidra/
[3] Dennis Andriesse, Asia Slowinska, and Herbert Bos. 2017. Compiler-Agnostic Function Detection in Binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. 177–189.
[4] T. Avgerinos, D. Brumley, J. Davis, R. Goulden, T. Nighswander, A. Rebert, and N. Williamson. 2018. The Mayhem Cyber Reasoning System. *IEEE Security & Privacy* 16, 2 (2018), 52–60.
[5] Tiffany Bao and David Brumley. 2014. ByteWeight: Recognizing Functions in Binary Code. (2014). http://security.ece.cmu.edu/byteweight/
[6] Tiffany Bao, Jonathan Burket, Maverick Woa, Rafael Turner, and David Brumley. 2014. ByteWeight: Learning to Recognize Functions in Binary Code. In *Proc. USENIX Security Symposium*. 845–860.
[7] BAP 2019. BAP: Binary analysis platform. (2019). http://bap.ece.cmu.edu/
[8] Capstone 2019. Capstone: The Ultimate Disassembler. (2019). http://www.capstone-engine.org/
[9] SPEC Corp. 2017. SPEC Benchmarks. (2017). http://www.spec.org
[10] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. 2017. REV.NG: A Unified Binary Analysis Framework to Recover CFGs and Function Boundaries. In *52nd Annual IEEE Carnahan Conference on Security Technology*. 131–141. https://doi.org/10.1145/3033019.3033028
[11] IDA 2019. Hex-Rays IDA. (2019). https://www.hex-rays.com/products/ida/
[12] Nucleus 2018. Nucleus source code. (2018). https://www.vusec.net/projects/function-detection
[13] N. E. Rosenblum, X. Zhu, B. P. Miller, and K. Hunt. 2008. Learning to analyze binary computer code.. In *National Conference on Artificial Intelligence*. 798—-804.
[14] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing Functions in Binaries with Neural Networks. In *Proc. USENIX Security Symposium*. 611–626.
[15] Jia Song and Jim Alves-Foss. 2015. The DARPA Cyber Grand Challenge: A Competitor's Perspective, Part 1. *IEEE Security & Privacy* 13, 6 (2015), 72–76.
[16] Jia Song and Jim Alves-Foss. 2016. The DARPA Cyber Grand Challenge: A Competitor's Perspective, Part 2. *IEEE Security & Privacy* 14, 1 (2016), 76–81.
[17] M. Zhang and R. Sekar. 2013. Control flow integrity for COTS binaries. In *22nd USENIX Security Symposium*. 337—-352.

APPENDICIES

A.1 FORMAL DEFINITIONS

We present the formal definitions of precision, recall and F1 value for those not familiar with the terms, as well as a formalism of the task definition.

A.1.1 Notation

When evaluating our solution compared to previously published work [3, 6, 14] we use the following metrics: *precision*, *recall* and *F1 score*. Each of these is reported for both function start and function boundary results. When Jima returns a list of function starts and boundaries for a binary, these metrics are defined as follows:

Precision When looking at the functions reported by the various tools, we will have correct results, *true positives* (**tp**) and incorrect results, *false positives* (**fp**). Precision is defined as the ratio of **tp** to total reported positive results:

$$\frac{tp}{tp + fp}$$

Recall When looking at the functions reported compared to ground truth, we will have our correct results, **tp**, and the functions that are missed, the *false negatives* (**fn**). Recall is defined as the ratio of **tp** to the total correct positives:

$$\frac{tp}{tp + fn}$$

F1 score The F1 score is a weighted average of precision and recall and can be used to compare the overall effectiveness of the full solution:

$$\frac{2 * precision * recall}{precision + recall}$$

Note that precision focuses on the lack of false positives, where recall focuses on lack of false negatives. If we report every address as a possible function start, we will have 100% recall, but practically 0% precision. We want both numbers to be high, and this is where the F1 score is useful, informing us of the combined effectiveness of both precision and recall.

For analysis of binaries, there will be a correlation between these numbers. If the tool reports too short of a function boundary, it may then assume subsequent instructions are part of a new function. Therefore a false positive report in a function boundary can result in a false positive in a function start report. Conversely, if tool detects too long of a function boundary, it can accidentally merge two functions, and now the false positive boundary report can generate a false negative function start report. We conjecture that if function start detection is perfect and there are no false positives and false negatives then the tool can also accurately calculate function boundaries.

A.1.2 Task Definition

Assume the tool has access to the stripped binary code C of a program which contains n functions, $f_1, \ldots, f_n$. For each function i we define (s_i, e_i) as the memory addresses of the first and last byte in the function. As with prior work [6, 14] we define the following tasks:

- Function start identification: Given C, find $\{s_1, \ldots, s_n\}$, the address of the first executable byte of all functions. This is typically the single entry point for a function, although some compilers and languages will allow multiple function entry points (See Section 3.3 and Fig. 4 in Bao et al. [6] for a discussion of this.)
- Function end identification: Given C, find $\{e_1, \ldots, e_n\}$, the address of the last executable byte of each function.
- Function boundary identification: Given C, find

  $$\{(s_1, e_1), \ldots, (s_n, e_n)\}$$

  the pairing of the addresses of the first and last executable bytes in all functions, assuming that each function is contiguous in memory.
- General Function identification: Given C, find

  $$\{(b_{1,1}, b_{1,2}, ..b_{1,l1}), \ldots, (b_{n,1}, b_{n,2}, ..b_{n,ln})\}$$

  all bytes of all functions, not necessarily unique or contiguous.

As stated by Shin et al. [14], function boundary identification is a superset of function start and end identification, and general function identification is a super set of all other tasks. As with prior work [6, 14] we leave general function identification to later work. In addition, our work solves function boundary identification and then extracts function start (and end) information from that analysis.

A.2 GROUND TRUTH

The following notes are intended to help clarify the analysis to enable comparison with similar work.

Ground Truth. Ground truth is derived from an unstripped ELF file. Using `objdump` and `grep` we generating a listing of function starting bytes and their lengths. The result generates a listing (see Listing 5), where the first column is the function start address and the fifth column is function length. The following interpretations were used when comparing our work with prior work:

1. *Zero-length functions.* Some functions are listed with 0 length, (such as the first few seen in Listing 5). These functions are included in the analysis of function start addresses detection, but not function boundary detection. A manual inspection of our outputs confirmed that our tool finds the correct function boundaries.
2. *Function aliases.* Some functions have multiple names for the same function entry point (such as the last few entries seen in Listing 5). This is true for both icc and gcc compilers and is an artifact of the source code. Our generation of ground truth removed these duplicates. A review of the source code provided by Bao et al. [5] shows that they do not remove these duplicates and therefore count statistics for duplicate functions more than once. Overall duplicates represent a small percentage of the total functions, and therefore the impact on statistics is relatively small.
3. *Function Boundaries.* The `icc` compiler has an odd artifact in function size, in that several functions end with a NOP that is included in the overall function size. Jima does not count terminal NOPs and therefore reports shorted function

```
test@debian: /usr/bin/objdump -t -f gcc_binutils_64_O4_size | grep 'F .text' | sort
Function Start Addr                        Function Length               Function Name
0000000000401b70 g     F .text   0000000000000000             _start
0000000000401b9c l     F .text   0000000000000000             call_gmon_start
0000000000401bc0 l     F .text   0000000000000000             deregister_tm_clones
0000000000401bf0 l     F .text   0000000000000000             register_tm_clones
0000000000401c30 l     F .text   0000000000000000             __do_global_dtors_aux
0000000000401c50 l     F .text   0000000000000000             frame_dummy
0000000000401c7c l     F .text   0000000000000046             ptr_align
0000000000401cc2 l     F .text   000000000000010a             emit_ancillary_info
0000000000401dcc l     F .text   000000000000003e             emit_try_help
0000000000401e0a l     F .text   0000000000000062             io_blksize
0000000000401e6c g     F .text   0000000000000105             usage
...
00000000004cd080 g     F .text   000000000000000f             .hidden __fstat
00000000004cd080  w    F .text   000000000000000f             .hidden fstat
00000000004cd090 g     F .text   0000000000000010             .hidden __lstat
00000000004cd090  w    F .text   0000000000000010             .hidden lstat
```

Listing 5: Generating Ground Truth

```
08082f4e <fix_syms>:
8082f4e:     pushl   %esi
8082f4f:     pushl   %ebx
8082f50:     pushl   %ebp
8082f51:     movl    0x10(%esp),%ebx
8082f55:     movzbl  0xc(%ebx),%eax
8082f59:     cmpl    $0x3,%eax
```

Listing 6: ICC Compiler Optimization O1

```
80b41c0 <fix_syms>:
80b41c0:          movl    0x4(%esp),%eax
80b41c4:          movl    0x8(%esp),%edx

080b41c8 <fix_syms.>:
80b41c8:     pushl   %esi
80b41c9:     pushl   %edi
80b41ca:     pushl   %ebx
80b41cb:     pushl   %ebp
80b41cc:     subl    $0xc,%esp
80b41cf:     movl.s  %eax,%ebp
80b41d1:     movl.s  %edx,%esi
80b41d3:     movzbl  0xc(%ebp),%eax
80b41d7:     cmpl    $0x3,%eax
```

Listing 7: ICC Compiler Optimization O2

lengths. An examination of the code by Bao et al. [5] shows that they determined a function boundary was correct if the calculated length was less than or equal to the ground truth length.

(4) *New Functions.* Compilers may create multiple function entry points, or even multiple functions from a single source code function, listing each as a separate function in the ELF file. We have seen this occur with compilers creating a function preface to map stack parameters into registers, allowing optimized function calls that just use registers to bypass the preface. We have also seen compilers take a function that has distinct execution paths, through a switch or if-then-else statement predicated upon a parameter, and create multiple separate functions. The optimizing compiler can determine the value of the parameter at call time and then call the correct new function.

For this work, we use all the functions as reported in the ELF file, except when there are multiple names for the same function. The code in Listings 6-7 demonstrates this. The icc compiler with optimization level O1 generated the assembly fragment in Listing 6 for the start of function "`fix_syms`". However, for optimization level O2, the compiler generates the code fragment in Listing 7, and generates a second function symbol "`fix_syms.`". This second symbol could be used as an entry point for a function that passed arguments using registers and not the stack. However, in this example, the generated binary never called this second entry point, therefore we have to question if it is a true function and part of the ground truth, or should be ignored.