A Look Back on a Function Identification Problem

Hyungjoon Koo
Sungkyunkwan University
Suwon, South Korea
kevin.koo@skku.edu

Soyeon Park
Georgia Institute of Technology
Atlanta, Georgia, USA
spark720@gatech.edu

Taesoo Kim
Georgia Institute of Technology
Atlanta, Georgia, USA
taesoo@gatech.edu

ABSTRACT

A function recognition problem serves as a basis for further binary analysis and many applications. Although common challenges for function detection are well known, prior works have repeatedly claimed a noticeable result with a high precision and recall. In this paper, we aim to fill the void of what has been overlooked or misinterpreted by closely looking into the previous datasets, metrics, and evaluations with varying case studies. Our major findings are that i) a common corpus like GNU utilities is insufficient to represent the effectiveness of function identification, ii) it is difficult to claim, at least in the current form, that an ML-oriented approach is scientifically superior to deterministic ones like IDA or Ghidra, iii) the current metrics may not be reasonable enough to measure varying function detection cases, and iv) the capability of recognizing functions depends on each tool's strategic or peculiar choice. We perform re-evaluation of existing approaches on our own dataset, demonstrating that not a single state-of-the-art tool dominates all the others. In conclusion, a function detection problem has not yet been fully addressed, and we need a better methodology and metric to make advances in the field of function identification.

CCS CONCEPTS

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

KEYWORDS

Function Identification, Function Recognition, ML-oriented, Lookback, Binary

1 INTRODUCTION

Function identification (or recognition) serves as a basis for reversing executable binaries because a function plays a pivol role of a logical chunk to understand the high-level semantics from a low-level binary. In this regard, a majority of binary analysis tools (e.g., BAP [7], BitBlaze [6], angr [25], radare [22], IDA Pro [16], Ghidra [9], rev.ng [11]) often necessitate detecting function boundaries for further analysis by default. Likewise, a number of other tasks can be performed on a function level including but not limited to control flow integrity (CFI), binary similarity analysis, binary instrumentation such as code randomization or re-optimization, type inference, and vulnerability detection.

Obtaining a function boundary on the availability of symbols or debugging information is trivial because the information readily contains the location and size of the function. However, it becomes drastically challenging when those information is stripped off, which is more common than not in practice.

A function identification problem begins with accurately recognizing all machine instructions. Once such a disassembly process is complete, one can seek a function by maintaining a database of (heuristically) known signatures like function prologues and epilogues. However, these two outwardly simple procedures are both challenging. First, accurate and robust disassembly is nontrivial. The predominant approach of disassembly is twofold. One popular way is to linearly disassemble all code (e.g., `objdump`), which is straightforward by mechanically converting byte codes into human-readable instructions. However, it suffers from robustness when code and data are intermixed (e.g., raw data in a code region, code pointer in a data region) because data may be mistakenly converted into code, and vice versa. Another means is a recursive traversal from an entry point of a binary that follows a direct control flow transfer until no new code region is discovered. Yet, indirectly reachable (or unreachable) functions may not always be statically identified. Inaccurate disassembly hampers applying function signature matching. Second, varying known compiler optimization techniques can blur the signature of a function even after successful disassembly including i) a non-returning function makes a function epilogue unclear, ii) a multi-entry function allows for jumping into the middle of a function, iii) a non-contiguous function may span multiple locations, or iv) compiler-specific heuristics may render a function boundary opaque. Besides, maintaining a signature database to support a new compiler and optimization faces another challenge as well as lack of a predefined pattern for a highly optimized function.

Despite the aforementioned challenges, many prior works have repeatedly demonstrated remarkable results with a high precision and recall (i.e., mostly 93% or above). Recent advances harness a machine learning technique (e.g., RNN), which claims to achieve even higher accuracy [14, 24]. In this paper, we revisit previous approaches including recent advances from a different angle in the domain of function identification. Note that our objective is neither to verify the correctness of prior evaluations nor to rank the existing work by comparison because there is no doubt about empirical results that are accurate and reproducible. Instead, we attempt to fill the void of what may have been overlooked or misinterpreted by closely looking into the previous corpus for evaluation, metrics, and evaluation results with the following five research perspectives in mind: i) appropriateness of the previous corpus (e.g., GNU utilities), ii) re-interpretation of the prior evaluations, iii) reasonableness of

ACSAC '21, December 6–10, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8579-4/21/12…$15.00
https://doi.org/10.1145/3485832.3488018

the current metrics, iv) effectiveness of ML-oriented techniques, and v) faithfulness of the existing tools for function identification.

First, we thoroughly assess the appropriateness of GNU utilites because most subsequent works have employed them for their evaluations after the initial release by ByteWeight [5]. We have fully quantified the bias of the dataset (first claimed by Nucleus [4]), and found that a large number of redundant functions are inevitably inserted due to a common static library during compilation. In particular, function identification with machine learning techniques often takes a *normalization* process (i.e., pre-processing) to feed instructions to a model. This renders an overfitting problem unavoidable with too many redundant functions (e.g., solely 12.1% remains unique after normalization). Second, we re-interpret that the accuracy of re-implementation of Shin's RNN [24] in LEMNA [14], unveiling that it comes from a different metric (i.e., a series of true negatives per each following byte rather than per each function). Third, we investigate varying case studies to determine the correctness of a function boundary that the current metrics cannot reasonably cover, necessitating that a better metric be explored for a fair comparison. Fourth, our re-evaluation with a different dataset demonstrates that it is difficult to claim that the current form of ML-oriented approaches surpasses rule-based ones like IDA Pro [16] or Ghidra [9]. Fifth, we conduct a comparison of function recognition results between different tools. Our finding shows that the outcome may considerably relies on the peculiarity of each tool; e.g., IDA Pro seeks functions with a recursive traversal, which does not report unused or unreachable ones on purpose (under reporting), whereas Ghidra utilizes FDE information to explore more functions, which sometimes leads over reporting. Likewise, the performance of ML-based approaches may highly vary depending on a learning dataset.

In conclusion, a function identification problem has not been fully resolved yet even with existing machine learning techniques, necessitating a better methodology and metrics. The following summarizes the key findings by looking back on prior function identification approaches.

- A common corpus (GNU utilities) is insufficient to represent the effectiveness of function identification.
- We investigate various edge cases that the current metrics may not cover.
- Our re-evaluation with a new dataset demonstrates that it is difficult to confidently claim that the effectiveness of ML-oriented approach is indeed superior to rule-based approaches.
- Revisiting existing approaches, not a single tool dominates all the others.

We will release our dataset [1] to foster future research of function identification.

2 REVISITING FUNCTION IDENTIFICATION

An executable binary that is stripped off valuable information presents difficulties in understanding underlying behaviors. Analyzing malware or any off-the-shelf binaries is commonly encountered. For demystifying the intent of a program, a binary analyst digs into a wealth of information through static code (e.g., instructions, basic blocks, functions), data (e.g., global and local variables, strings), and structure (e.g., control flow graphs, call graphs, jump tables). Meanwhile, dynamic analysis allows one to capture a wide range of interactions with an operating system by running the code.

In particular, identifying a function can be fruitful to roughly sketch the behavior and intent of a program to begin with because it often represents a logical chunk that performs a meaningful task. In this section, we briefly describe the definition of a function detection problem, and common evaluation criteria, followed by outlining prior approaches.

2.1 Problem Definition

A function recognition problem aims to discover a set of (binary) functions in case that no symbol or debugging information is readily available, which includes i) function starts and ii) function boundaries (both starts and ends). Formally, let each byte be a b_i where the entire code of a binary consists of n bytes; then a set of consecutive bytes in a code region would be $\{b_1, b_2, ..., b_n\}$. Suppose that the code region contains k functions whose start and end can be represented as (s_i, e_i).

Under this setting, as with prior work [2, 5], we define the following two tasks: i) function starts identification; find a set of the beginning of all functions: $\{s_1, s_2, ..., s_k\}$, and ii) function boundaries identification; find a set of the boundaries of all functions: $\{(s_1, e_1), ..., (s_k, e_k)\}$. As a function may be split into multiple areas (non-contiguous), a more general expression of finding a function F_i would be: $\{(s_i, b_1, ..., b_v), ...(b_w, b_{w+1}, ..., e_i)\}$ where the middle indexes of v and w may indicate arbitrary positions.

2.2 Evaluation Metrics

Let a set of true positives and true negatives (i.e., aligned with ground truths), false positives (i.e., identified as a function where it is not), and false negatives (i.e., missed a function where it is) be TP, TN, FP, and FN, respectively. The following defines a precision (P), recall (R), $F1$ score, and accuracy (A).

$$P = \frac{|TP|}{|TP| + |FP|},\ R = \frac{|TP|}{|TP| + |FN|},\ F1 = \frac{2PR}{P + R} \quad (1)$$

$$A = \frac{|TP| + |TN|}{|TP| + |TN| + |FN| + |FP|} \quad (2)$$

P and R are defined as in Equation 1: a high precision means the rate of incorrectly identified functions (FP) is low, and a high recall means the rate of missing functions (FN) is low. The $F1$ represents a single metric with the harmonic mean of P and R. Accuracy can be computed as the rate of ground truths out of all cases (Equation 2). For all evaluation metrics, the higher values represent better performance.

2.3 Prior Approaches

We classify the existing function detection techniques into two categories: a rule-based approach (i.e., using algorithms or known signatures) and a machine learning based approach (i.e., using stochastic models inferred from a given dataset).

Rule-based Approach. UNSTRIP [18] introduces semantic descriptors (i.e., system calls and concrete argument values) that represent library functions as a fingerprint for further function

[1] https://github.com/SecAI-Lab/func-identication

Table 1: Comparison of the existing works for function detection. (*) indicates the work that has been included for our evaluation.

Tool	Artifacts	Year	Dataset	Arch	Type	Compiler	OptLevel	# Binaries	Compared With	F1
Nucleus* [4]	Y	2017	SPEC2006, ngnix, lighttpd, openssh, vsftpd, exim	x86/x64	ELF/PE	clang/VS	O0-O3	476	Dyninst, ByteWeight, IDA	0.970
Function-interface [21]	N	2017	GNU Utils, SPEC2006, GLIBC	x86/x64	ELF	clang/gcc	O0-O3	2,488	ByteWeight, Shin:RNN	0.985
Jima [2]	Y	2019	GNU Utils, SPEC2017, Chrome	x86/x64	ELF	clang/gcc/icc	O0-O3	2,860	ByteWeight, Shin:RNN, IDA Free, Ghidra, Nucleus	0.997
Nathan:CRF [23]	N	2007	Unknown	x86	ELF/PE	gcc/icc/VS	Unknown	1,171	N/A	N/A
ByteWeight* [5]	Y	2014	GNU Utils	x86/x64	ELF/PE	clang/gcc	O0-O3	2,200	Dyninst, ByteWeight, BAP, IDA	0.929
Shin:RNN* [24]	N	2015	GNU Utils	x86/x64	ELF/PE	clang/gcc	O0-O3	2,200	ByteWeight	0.983
FID [28]	N	2017	GNU coreutils	x86/x64	ELF	clang/gcc/icc	O0-O3	4,240	IDA, ByteWeight	0.930
LEMNA* [14]	Y	2018	GNU Utils	x64	ELF	gcc	O0-O3	2,200	N/A	N/A

identification. It aims to generate smart patterns to detect even unknown functions rather than fully deterministic byte-level signatures. Nucleus [4] presents a function detection algorithm in a compiler agnostic fashion. With linearly disassembled code, Nucleus detects basic blocks and builds an inter-procedural control flow graph (ICFG) in the beginning. Once a direct call invocation over the ICFG reveals function entry blocks, Nucleus discovers either unreachable or indirectly reachable functions (isolated from the initial ICFG) via intra-procedural control flow analysis. Qiao et al. [21] develop another means based on static analysis. Similar to Nucleus, it collects function candidates that cannot be directly reachable, followed by checking whether they are associated with a function interface, including stack discipline, control-flow properties, and data-flow properties (i.e., parameter passing). Jima [2] is a tool suite that incorporates a series of analysis algorithms for function boundary detection, including an exception handling routine, jump pointer, tail call chain, and missing function detection (i.e., gaps between functions).

As a commercial tool, IDA Pro [16] offers powerful functionalities and tools for reversing, which is equipped with disassembly, decompilation and debugging features for better code analysis; however, its internal heuristics (i.e., pattern database) for function detection remain proprietary and thus unknown. It is noteworthy mentioning that IDA ships with a known function identification algorithm, dubbed FLIRT [15] that maintains a signature database of each function for a standard library, however, it cannot be applied to general function identification. Similarly, Ghidra [9] is an emerging open-source disassembler that offers a suite of reversing tools and decompiler. It provides a few built-in function analyzers such as `FunctionStartAnalyzer`. The analyzers begin with identifying every address referenced by a call instruction as the beginning of a function, and then utilizes a static signature database that records a known function start pattern according to a compiler and architecture [10]. Likewise, other binary analysis tools provide a comparable mechanism (i.e., manually generated signatures) for function detection such as BAP [7], BitBlaze [6], angr [25], radare [22], and rev.ng [11].

Machine Leaning Based Approach. One of the early works [23] based on machine learning adopts a model with a conditional random field (CRF) for identifying function entry points (FEPs). The model takes both idiom features (i.e., instruction sequences) and structure features (i.e., control flow) into account to classify FEPs in a binary code. Byteweight [5] builds a weighted prefix tree to recognize function starts using a precomputed signature at training time. The prefix tree holds a likelihood of a function constructed from a training data set where each node represents either a byte or an instruction, e.g, learning the probability of an FEP from a sequence of instructions (i.e., path from the root to the given node). Note that BAP [7] currently incorporates the Byteweight model as a plugin [13]. FID [28] proposes the combination of symbolic execution and machine learning, mostly focusing on identifying an FEP block. It has the internal representations of each basic block semantics with assignment formulas (e.g., stack registers) and memory access behavior (e.g., memory read), converting them into numeric feature vectors for a classifier. FID takes three learning algorithms (i.e., LinearSVC, AdaBoost, and GradientBoosting) for better prediction.

Meanwhile, Shin et al. [24] utilizes a deep learning approach for the first time, which leverages a bidirectional recurrent neural networks (RNN) model with a single hidden layer to tackle both function starts and boundary identification. Despite the absence of clear explanations for the underlying mechanism of the model, the empirical results consistently demonstrate a very high precision and recall for binaries in different formats (e.g., ELF and PE), and different architectures (e.g., x86 and x86_64 binaries). Recently, LEMNA [14] introduces the first explainable model for deep learning based security applications. It integrates fused lasso [27] for handling a feature dependency problem with a mixture regression model [19] that achieves an accurate approximation for a local decision boundary. In particular, LEMNA applies the model to function start detection for explaining the reasons for various false positive and false negative cases. We employ LEMNA's reimplementation of the Shin et al.'s approach for our evaluation as the original implementation is unavailable.

Summary of Two Approaches. Table 1 summarizes a comparison of previous approaches in the area of function identification at a glance with varying criteria including the availability of a tool, publication year, dataset information (architecture, file format, compiler, optimization level, number of binaries for testing), compared work, and reported F1 values (for `x86_64` binaries). All work target either `x86` or `x86_64` architecture, mostly focusing on ELF. Interestingly, after the first public release of the dataset (i.e., GNU Utilities) from ByteWeight [8], it becomes a de-facto standard for evaluation; every work (but Nucleus [4]) includes that corpus. Note that the

F1 values are for recognizing the beginning of functions (not function boundaries) on average. Most tools reported noticeable results, ranging from 0.929 to 0.997.

3 CHALLENGES OF FUNCTION IDENTIFICATION IN AN EXECUTABLE

A binary function that resides in a code section differs from a human-written function that conveys semantics in a source code. In a nutshell, every binary function originates from a function that is i) defined by a user (e.g., source code written by a developer), ii) generated by a compiler (e.g., stack canary check, control flow integrity routines), or iii) inserted by a linker (e.g, CRT or C RunTime functions). This section discusses common challenges for identifying functions in a stripped binary with a motivating example.

Motivating Example. IDA Pro [16] is one of the most popular reversing tools for many years. It incorporates a variety of different techniques to accurately analyze a given binary.

```
1  0x4338F0  push  r14
2  0x4338F2  push  rbx
3  0x4338F3  push  rax
4  0x4338F4  mov   rbx, rsi
5  0x4338F7  mov   r14, rdi
6  0x4338FA  mov   rax, [rsi+10h]
7  0x4338FE  cmp   rax, 2FFh
8  0x4340E2  mov   rdi, r14 [jumptable 0x433A0B case 1]
9  0x4340E5  mov   rsi, rbx
10 0x4340E8  add   rsp, 8
11 0x4340EC  pop   rbx
12 0x4340ED  pop   r14
13 0x4340EF  jmp   sub_434620
14 ; Wrong Subroutine: Case (I)
15 0x4340F4  mov   rdi, r14  [DATA XREF: .rodata:0x4A7C18]
16 0x4340F7  mov   rsi, rbx
17 0x4340FA  xor   edx, edx
18 0x4340FC  add   rsp, 8
19 0x434100  pop   rbx
20 0x434101  pop   r14
21 0x434103  jmp   sub_434BE0
22 ; End of Case (I)
23 ; Wrong Subroutine: Case (II)
24 0x434108  mov   rdi, r14  [DATA XREF: .rodata:0x4A7C78]
25 0x43410B  mov   rsi, rbx
26 0x43410E  add   rsp, 8
27 0x434112  pop   rbx
28 0x434113  pop   r14
29 0x434115  jmp   sub_434630
30 ; End of Case (II)
31 0x43411A  mov   esi, offset
32 0x43411F  mov   rdi, r14  [CODE XREF: sub_4338F0+177]
33 0x434122  mov   rdx, rbx
34 0x434125  add   rsp, 8
35 0x434129  pop   rbx
36 0x43412A  pop   r14
37 0x434156  mov   rdi, r14  [CODE XREF: sub_4338F0+857]
38 0x434159  mov   rsi, rbx
39 0x43415C  add   rsp, 8
40 0x434160  pop   rbx
41 0x434161  pop   r14
42 0x434163  jmp   sub_434830
43 ; Termination of a Function
```

Listing 1: Example of mitstakenly identifying functions from `randlib-amd64-clang-01` with IDA Pro.

Listing 1 shows that IDA Pro mistakenly reports two regions as the beginning of functions whereas the ground truth is that the entire chunk of disassembled code consists of a single function. In this example, IDA Pro successfully discovers a jump table (Line 8), cross references in a data section (Line 15 and 24), and those in a code section (Line 32 and 37) after a control flow analysis. Two subroutines (Line 15-21 and 24-29) have been determined as separate functions probably by acknowledging that it clears the stack (line 18 and 26), followed by popping two registers (line 19-20 and 27-28) with a unconditional jump (Line 21 and 29) at the end. Indeed, the sequences of such instructions are indistinguishable from the actual function termination (Line 39-42), which must be making IDA Pro confusing. This structure can be often observed in case of early returning if a certain condition is met within a function (and possibly a control flow is transferred by a code pointer). As shown in this instance, it is challenging to precisely detect a function even with a cutting-edge disassembly tool.

Challenges. The common challenges for function detection in an excutable binary are well-known, mainly due to compiler optimizations and code regions intermixed by code and data. First, optimizated code often blurs a clear signature of a function prologue and epilogue, rendering its boundary detection less straightforward because of the following reasons.

- It is common that a function becomes part of another function (i.e., function inlining); it enhances program performance by eliminating a burden on both function prologue and epilogue.
- A call invocation happens at the end of a procedure (i.e., tail call); oftentimes compiler optimization replaces it with a single jump (instead of `pop` and `ret`) for boosting performance instead of returning to an original caller.
- A single routine may be split into multiple locations (i.e., non-contiguous function).
- Different function symbols can point to the same address when they are under identical implementation; it often occurs when a derived class inherits a method from a parent class in object oriented programming by containing the same code pointer in a virtual table.
- Compiler-generated code or compiler-specific heuristics may render a function identification process opaque, including non-returning functions ending with a call or multi-entry functions where a call invokes the middle of a function.

Second, another challenge arises from an incomplete disassembly process that complicates further function signature matching. Simply put, a linear disassembly suffers from robustness due to indistinguishability of code and data in a code region whereas a recursive traversal cannot handle indirectly reachable or unused functions. Third, code from a manually written assembly can make a function unrecognizable because there are many non-standard way of writing an assembly language. One may insert an overlapping instruction or dead one that would be never executed (i.e., anti-disassembly techniques) with the freedom of code representation. Similarly, if a function is not explicitly declared, the entire object file may be considered as a single function, which also makes a function boundary obscure.

4 A CLOSE LOOK BACK ON FUNCTION IDENTIFICATION

In this section, we look back on a function identification problem mainly focusing on five research questions. Note that we have carefully investigated plentiful cases to support our claims.

Table 2: 10 Groups for 10-fold cross validation for ByteWeight.

Group	Files	Funcs	Set	Group	Files	Funcs	Set
Group 1	57	19,996	train	Group 6	49	12,236	train
Group 2	55	9,475	train	Group 7	48	12,197	train
Group 3	51	18,442	train	Group 8	46	12,324	train
Group 4	57	13,779	train	Group 9	46	20,680	test
Group 5	55	13,481	train	Group 10	52	13,519	train

4.1 Research Questions

We revisit prior approaches by closely looking into the previous datasets and evaluations by defining the following research questions that focus on ① appropriateness of dataset, ② reinterpretation of prior evaluations, ③ effectiveness of ML techniques, ④ rethinking of metrics, and ⑤ faithfulness of a tool for function identification.

- **RQ1.** Is the previous dataset (e.g., GNU utilities) appropriate for the effectiveness of a function detection technique?
- **RQ2.** Has a function detection problem been (almost) resolved as reported with a very high F1 or accuracy?
- **RQ3.** Is the current metric (i.e., precision, recall and F1) fair enough to measure function identification in general?
- **RQ4.** Are recent advances with an ML-centered approach (e.g., deep learning) superior to a rule-based one?
- **RQ5.** Is there a tool's own characteristic (i.e., idiosyncrasy that arises from a unique algorithm or behavior)? There may be a hurdle to simply pick the best function identification tool that dominates all others.

Final Goal. Our objective is neither to re-verify the correctness of prior work nor to rank the existing approaches by comparison. Rather, we attempt to fill the gaps that may have been overlooked or misinterpreted by answering the above research questions, and then eventually to seek whether a function recognition problem has been fully addressed. To this end, we conduct our own experiments with a new dataset in §5.

4.2 Appropriateness of Corpus

This section delves into one of the most popular corpus, GNU utilities that a majority of previous works in the field of function identification employs for evaluation. The GNU utilities consist of 16 `binutils`, 104 `coreutils` and nine `findutils`. In particular, all subsequent works but Nucleus [4] use the same corpus for their evaluations since the first release of ByteWeight's GNU utilities [8] as in Table 6. The publicly available dataset, in general, is quite beneficial to foster the future work because the common corpus allows for a fair comparison from different approaches. However, it is important to examine that the dataset should be moderately representative to assess a function identification problem without a bias.

Nucleus has first claimed that the released dataset of GNU utilities are too biased to be generalized with a limited assessment. To confirm such a claim, we have quantified the bias of the whole dataset, 2,200 binaries, adopted by ByteWeight. Note that, for simplicity, we solely focus on x64 binaries compiled with `gcc`. ByteWeight internally performs instruction normalization as a preprocessing step before generating a weighted tree, which converts both an immediate value and the target of a call/jump instruction into a generalized value [2], toward effectiveness and efficiency. Although the normalization step is essential to apply a machine learning technique, the problem is that only 17.6K (12.1%) out of the whole 146K functions remain unique normalized functions (NFs) once a normalization process is complete. Too many redundant data unavoidably faces an *overfitting* problem. Table 2 shows 10 different groups utilized in ByteWeight [8] with k-fold cross validation ($k = 10$). Indeed, 19.8K NFs (91.4%) in a test set have been discovered in a train set when selecting Group 9 as a test set in Table 2. We also discovered that 10K NFs are shown at least more than twice across the dataset, indicating that a severe overfitting is highly likely at all times.

```
1  ; // binutils - ar
2  ; void yyset_lineno(int line_number) {
3  ; yylineno = line_number;
4  ; }
5
6  0x432273:  push   rbp
7  0x432274:  mov    rbp,rsp
8  0x432277:  mov    DWORD PTR [rbp-0x4],edi
9  0x43227a:  mov    eax,DWORD PTR [rbp-0x4]
10 0x43227d:  mov    DWORD PTR [rip+0x378a81],eax
11 0x432283:  pop    rbp
12 0x432284:  ret
13
14 ; // binutils - as
15 ; static void set_allow_index_reg (int flag) {
16 ; allow_index_reg = flag;
17 ; }
18
19 0x4049c8:  push   rbp
20 0x4049c9:  mov    rbp,rsp
21 0x4049cc:  mov    DWORD PTR [rbp-0x4],edi
22 0x4049cf:  mov    eax,DWORD PTR [rbp-0x4]
23 0x4049d2:  mov    DWORD PTR [rip+0x30fbd8],eax
24 0x4049d8:  pop    rbp
25 0x4049d9:  ret
```

Listing 2: Example of an identical function pair after normalization.

The redundancy mainly arises from a static library in common during compilation: over a hundered of binaries from `coreutils` take a shared library of `libcoreutils.a`, including 776 common functions from 257 object files. Because the granularity of consolidation at link time is an object file, the entire functions inside the object become part of a final executable when even a single function would be in use, which inevitably introduces a substantial number of duplicate functions across the corpus.

Another interesting finding is that there are a considerable number of NFs even between different functions from different binaries. For instance, Listing 2 depicts two binary functions that are identical after normalization. The source code of those functions (line 1-4, 14-17) similarly takes a single integer as a parameter and then assigns it into a local variable. In this example, there are 16 identical NFs across six binaries.

[2]For verification, we normalize an immediate with a single value whereas ByteWeight has a few different ones (i.e., zero, positive, negative), however, it does not significantly change the final outcome.

4.3 Re-interpretation of Prior Evaluations

In this section, we revisit prior evaluations that may lead to a misinterpretation that the function detection problem has been solved despite myriad hurdles described in Section 3. ByteWeight reports an F1 value of 98.8% for ELF x64, and similarly the RNN model proposed by Shin et al.achieves 98.3%. LEMNA has re-implemented Shin's RNN model for function identification and reported a result comparable to the original one (F1 of 99.4%). In particular, LEMNA achieves an extremely high accuracy, 99.99%, across all optimization levels. In the same vein, other works consistently showcase a remarkable outcome (Table 1).

It is not questioned that the empirical results are accurate and reproducible, however, as discussed in Section 4.2, we claim that one reason for a high detection rate partially stems from an inappropriate corpus. To this end, we further carry out several experiments to support our claim. First, we employ a relatively new standard dataset, SPEC2017, to confirm that the signature of ByteWeight works well in general. Table 8 shows F1 is close to 61.7, which is far beyond the reported value. After retraining the ByteWeight model with SPEC2017, we obtain an F1 of 78.0. Second, we attempt to reproduce the accuracy of Shin's RNN model (source unavailable) with our dataset from the LEMNA's open source implementation, obtaining 94.5 and 86.1 as a precision and recall (See Table 8), respectively. Indeed, we are able to obtain an overly high accuracy as claimed, but it turns out that the accuracy comes from the means of counting true negatives. As Shin's bidirectional RNN model determines if the next byte is a function start upon a given sequence of bytes (i.e., input of n bytes as a hyperparameter), it results in a series of decisions per each following byte. If the size of a binary is s, $s-n$ decisions would produce a large number of true negatives because a majority of bytes do not represent the beginning of a function, which makes accuracy reach 99.99%, according to Equation 2.

```
1  MagickExport ImageInfo *AcquireImageInfo(void) {
2    ImageInfo *image_info;
3    image_info=(ImageInfo *) AcquireMagickMemory(sizeof(*
       image_info));
4    if (image_info == (ImageInfo *) NULL)
5    ThrowFatalException(ResourceLimitFatalError,"
       MemoryAllocationFailed");
6    GetImageInfo(image_info);
7    return(image_info);
8  }
9
10 ; ImageInfo *__cdecl AcquireImageInfo()
11 0x4C6BC0   push    rbx
12 0x4C6BC1   mov     edi, 4198h   ; size
13 0x4C6BC6   call    AcquireMagickMemory
14 0x4C6BCB   test    image_info, image_info
15 0x4C6BCE   jz      loc_4C6BE0
16 0x4C6BD0   mov     rbx, image_info
17 0x4C6BD3   mov     rdi, image_info ; image_info
18 0x4C6BD6   call    GetImageInfo
19 0x4C6BDB   mov     rax, image_info
20 0x4C6BDE   pop     image_info
21 0x4C6BDF   retn
22 0x4C6BE0   call    AcquireImageInfo.part.2
23 ...
24 ; ImageInfo *__cdecl AcquireImageInfo.part.2()
25 0x402554   push    rbx
26 0x402555   sub     rsp, 40h
27 0x402559   mov     rdi, rsp ; exception
28 ...
29 0x4025C4   call    DestroyExceptionInfo
30 0x4025C9   call    MagickCoreTerminus
31 0x4025CE   mov     edi, 1  ; status
32 0x4025D3   call    __exit
```

Listing 3: Example of a non-continous function and its disassembly after optimization.

4.4 Rethinking of Current Metrics

This section expands our concern (both unsuitable dataset and evaluation that may lead the misinterpretation of a result) that the current metrics (i.e., precision, recall, and F1 shown in Equation 1) may not be fair as a means to measure the effectiveness of function identification. We provide several case studies to rethink the suitability of the current metrics for function detection.

Non-continuous Functions. Listing 3 shows the code snippet (Line 1-8) and its disassembly from `imagick_r-amd64-gcc-03`. A compiler optimization takes an exception handler apart (Line 24-32), holding two separate binary functions as a ground truth (i.e., `AcquireImageInfo` and `AcquireImageInfo.part.2`[3]). Although it takes up a small portion of entire functions (2,997 functions or 0.38% in our dataset), such margins may lead an unfair precision and recall because it is difficult to say either side (i.e., counting a non-continuous function as one or two) is inaccurate from a reversing perspective for binary analysis. In a similar vein, going back to Listing 5, the decision that those branch functions have been reasonable in terms of function boundary correctness is questionable. Interestingly, the register `rbx` at lines 36 and 37 holds a `p_sess` value instead of a base pointer to invoke the corresponding call. It means missing the boundary of the seemingly inlined (albeit separated) function does not hamper conducting further reversing in case that such a missing function (`cmd_process_pasv_cleanup`) is both semantically and tightly coupled with its caller.

Ground Truth from Debugging Information. It is very common to extract a ground truth of a function boundary from debugging information in a non-stripped binary because debugging sections contain function positions and sizes in a DWARF structure. However, such a structure can be found even in a stripped binary, that is, an `.eh_frame` section. It follows a DWARF format by default, storing call frame information (CFI) for an exception handling routine[4]. The CFI contains two entry forms: i) a common information entry (CIE) that corresponds to a single object and ii) a frame description entry (FDE) that contains a reference to a function and its length.

```
1 ; __int64 __fastcall atol_317(const char *__nptr)
2 0x9C0A20   xor   esi, esi
3 0x9C0A22   mov   edx, 0Ah
4 0x9C0A27   jmp   _strtol
```

Listing 4: Example of an identified function by Ghidra using FDE information where a symbol table does not hold.

Ghidra, one of the state-of-the-art disassemblers, harnesses such FDEs to identify a function, sometimes resulting in discovering more functions that may not reside in a symbol table alone[5]. To exemplify, Listing 4 demonstrates a short function from `cpugcc_r-amd64-clang-O1` that has been detected by Ghidra with FDE information where a ground truth (i.e., function symbol)

[3]The symbol name ending with "`.part.{num}`" has been generated by `gcc`. It is a compiler-specific behavior because `clang` (i.e., `imagick_r-amd64-clang-03`) holds a single function symbol.

[4]The `.eh_frame` section supports exceptions in C++, but System V ABI for AMD64 [17] mandates to have the section in a stripped binary (even written in C code) by convention.

[5]The GNU `binutils` such as `objdump` or `nm` reads function symbols from a symbol table (`.symtab` and `.dynsym`) by default rather than parsing entire debugging sections.

does not hold. We discovered that there are 13, 380 such functions in the above binary, which significantly increases the number of false positives for Ghidra and Nucleus. Although the function symbols are absent in a debugging section, those functions are indeed true positives. Under the current scheme of precision and recall, the F1 value of both Ghidra and Nucleus (96.0 and 90.4 in Table 8) may be distorted because the function in Listing 4 should be viewed as an actual binary function. Considering the functions that can be found in FDEs, the recalculated F1 of Ghidra and Nucleus would be 98.0 and 93.0, respectively, whereas that of IDA Pro drops (91.3 from 93.4), which would impact on the final ranking.

4.5 On the Effectiveness of ML Techniques

This section describes the effectiveness of ML-centric approaches including deep learning with several case studies.

```
static void
process_post_login_req(struct vsf_session* p_sess) {
  char cmd;
  /* Blocks */
  cmd = priv_sock_get_cmd(p_sess->parent_fd);
  if (tunable_chown_uploads && cmd == PRIV_SOCK_CHOWN)
  cmd_process_chown(p_sess);
  ...
  else if (cmd == PRIV_SOCK_PASV_CLEANUP)
  cmd_process_pasv_cleanup(p_sess);
  ...
  else
  die("bad request in process_post_login_req");
}

; Beginning of process_post_login_req()
0xAC10   push    rbx
0xAC11   mov     rbx, p_sess
0xAC14   mov     edi, [p_sess+180h] ; fd
0xAC1A   call    priv_sock_get_cmd
...
0xAC3F   lea     rcx, jpt_AC4D
0xAC46   movsxd  rax, ds:(jpt_AC4D - 16C38h)[rcx+rax*4]
0xAC4A   add     rax, rcx
0xAC4D   jmp     rax  ; jump table
0xAC52   pop     p_sess
0xAC53   jmp     cmd_process_get_data_sock
0xAC55   lea     rdi, aBadRequestInPr
0xAC5C   pop     p_sess
0xAC5D   jmp     die
...
0xAC80   pop     p_sess
0xAC81   jmp     cmd_process_pasv_cleanup

static void
cmd_process_pasv_cleanup(struct vsf_session* p_sess)
{
  vsf_privop_pasv_cleanup(p_sess);
  priv_sock_send_result(p_sess->parent_fd,
   PRIV_SOCK_RESULT_OK);
}

; Beginning of cmd_process_pasv_cleanup()
0xAD30   push    rbx
0xAD31   mov     rbx, p_sess
0xAD34   call    vsf_privop_pasv_cleanup
0xAD39   mov     edi, [p_sess+180h]
0xAD3F   mov     esi, 1
0xAD44   pop     p_sess
0xAD45   jmp     priv_sock_send_result
```

Listing 5: Example of a function and its disassembly after optimization. The `cmd_process_pasv_cleanup()` function has been discovered by an RNN alone over rule-based approaches.

Discovering True Functions. Shin's RNN [24] is one of early works that takes advantage of recurrent neural networks (RNN) in binary analysis, proposing a bi-directional model with RNN hidden units. We investigate that the proposed model indeed outperforms rule-based techniques and heuristics adopted by popular reversing tools like IDA Pro [16] and Ghidra [9]. Figure 1 illustrates a simple comparison between the number of true functions discovered by each approach for the utilities in Table 6. According to our experiment, the Shin's RNN model discovered 8, 380 and 1, 871 functions more than Ghidra and IDA, respectively (See Table 8 in detail). Meanwhile, Ghidra and IDA discovered 9, 453 and 10, 499 functions over the RNN. Interestingly, the RNN approach accurately found 696 unique functions that both Ghidra and IDA were missing. Although a deep learning approach demonstrates its own strength, however, it is difficult to conclude that an ML-oriented technique surpasses rule-based ones.

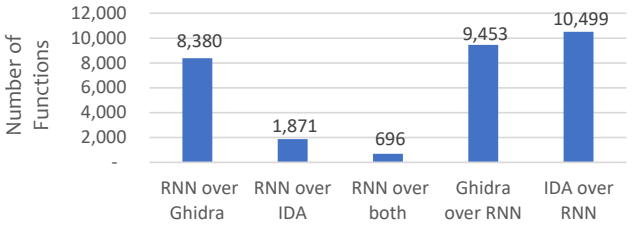

Figure 1: Comparison of the number of true functions between different tools (i.e., RNN VS rule-based approaches). The capability of discovering a function start is comparable with each other.

Table 3: Non-returning function detection rate across different tools. Rule-based approaches with heuristics demonstrate better performance than ML-oriented tools.

Tool	# of Missing	Total	Rate
IDA Pro	0	9,409	0.00%
Ghidra	54	9,409	0.57%
Nucleus	1,186	9,409	12.60%
ByteWeight	4,615	9,409	49.05%
ByteWeight*	2,024	5,125	39.49%
Shin:RNN	24	250	9.60%

Functions Discovered Solely by RNN. We further look into all 696 cases that the RNN model [24] could identify whereas rule-based techniques missed. Note that the behavior of the model lacks interpretability. Notably, 623 (90.5%) cases are from `openssl` binaries that quite a few hand-written assembly functions are included. Indeed, the prologue of those functions has deviated from known function signatures (e.g., starting with `shr` or `movdqu`). We presume that RNN deduces a hidden rule from manually written assembly functions during a learning process. 160 (23.0%) cases have relatively small functions in size (i.e., less than five instructions). Classifying with the starting instructions of missing functions by both rule-based tools, 299 cases begin with a `mov` instruction, followed by `push` (235), `jmp` (58), `lea` (39), `test` (27), and `cmp` (17).

Non-returning Functions. We also investigate a common structure that often complicates the decision of a function boundary (Listing 1 is one of good examples), that is, a non-returning function [6].

[6] We follow a particular flag (`FUNC_NORET`) that IDA Pro maintains for the analysis purpose.

Table 4: Function names (signatures) from Ghidra that do not return for ELF and PE format.

ELF (Executable and Linkable Format)	PE (Portable Executable)
exit	abort
cexit	CxxThrowException
c_exit	CxxThrowException@8
abort	CxxFrameHandler3
reboot	crtExitProcess
longjmp	ExitProcess
longjmp_chk	ExitThread
siglongjmp	exit
panic	FreeLibraryAndExitThread
stack_chk_fail	invalid_parameter_noinfo_noreturn
cxa_throw	invoke_watson
cxa_terminate	longjmp
cxa_call_unexpected	quick_exit
cxa_bad_cast	RpcRaiseException
Unwind_Resume	terminate
assert_fail	
assert_rtn	
fortify_fail	
ZSt9terminatev	
ZN10__cxxabiv111__terminateEPFvvE	
pthread_exit	

We could collect 9,409 cases (1.2%) in total that end with `call`, `jump`, or `__exit` such as Listing 5, Listing 3, and Listing 4 from our dateset. Table 3 concisely shows that ML-oriented approaches miss more functions than rule-based techniques. Interestingly, IDA Pro fully recognizes non-returning functions whereas the original ByteWeight model has the largest missing rate (i.e., almost half). Likewise, Ghidra (open source) deterministically defines a set of function names [1] that do not return as in Table 4.

Inlined Functions. We look into one of the examples in which the RNN approach has accurately captured all function starts, whereas both IDA Pro and Ghidra have failed to discover them (696 functions in Figure 1). Listing 5 illustrates the source code snippet (Line 1-14, Line 35-40) and its disassembly from `vsftpd-amd64-clang-O1`. This function takes a single argument (i.e., `p_sess`), which plays a role in branching out into multiple call invocations depending on the argument (i.e., Line 7, 10, or 13 otherwise). Although this example is slightly different from a typical function inlining case in that a function symbol resides in a symbol table, rule-based binary analysis tools regard each branch function as part of the `process_post_login_req()` function.

4.6 Faithfulness of Tools

We conduct a comparison of three function identification results with IDA Pro [16], Ghidra [9], and Shin's RNN [24] on `yes`, bundled in GNU Coreutils [12], which simply prints out a string until interrupted. The final executable contains 98 user-defined functions from 16 objects (i.e., compilation units) in total (excluding linker-inserted functions such as `_start`) when compiled with the optimization level of -O1 albeit its simple functionality. This is because even a single use of a function *consolidates all other functions* within the object that the function belongs to (or a static library) at link time, which may result in the presence of many unreachable functions in an executable binary.

Case Study: Under Reporting (IDA Pro). Table 5 summarizes a list of functions that at least one tool was unable to identify; e.g., the `hard_locale` function (at the bottom) with -O1 was not discovered by the RNN tool. A dash line means that a certain function is absent in the binary mostly due to function inlining at a high optimization level; e.g., `emit_ancillary_info` has been inlined (merged) to another function with -O2. Indeed, quite a few functions with the prefixes of `quotearg_` and `c_` have not been identified by IDA Pro. In this example, IDA Pro misses around half of the whole functions because it seeks functions with a recursive traversal [7]. This aligns with the results from Andriesse et al. [3] that disassembly tools

Table 5: Comparison of function identification results with IDA Pro (I), Ghidra (G), and Shin's RNN (R) tools. The capital letters represent that each tool could discover the beginning of a function whereas lower letters (i.e., i, g, r) represent a certain tool could not. IDA Pro purposefully does not identify unreachable (or unused) functions after constructing a control flow. Note that a dash means a function is not shown in a binary (e.g., function inlining).

Function Name	-O1	-O2	-O3
emit_ancillary_info	IGR	—	IGR
close_stdout_set_file_name	iGR	iGr	iGr
close_stdout_set_ignore_EPIPE	iGr	iGr	iGr
get_quoting_style	iGR	iGR	iGR
set_quoting_style	iGR	iGR	iGR
set_char_quoting	IGR	iGr	iGR
set_quoting_flags	iGR	iGR	iGR
set_custom_quoting	IGR	iGR	iGR
quotearg_alloc	iGR	IGR	IGR
quotearg_free	iGR	iGR	iGR
quotearg_n	iGR	iGR	iGR
quotearg_n_options	iGR	IGR	IGR
quotearg_n_mem	IGR	iGR	iGR
quotearg	iGr	iGR	iGR
quotearg_mem	iGR	iGR	iGR
quotearg_n_style	IGR	IGR	IGr
quotearg_style_mem	iGR	IGR	IGR
quotearg_char_mem	iGR	IGR	IGR
quotearg_colon_mem	iGR	IGR	IGR
quotearg_n_custom	iGR	IGR	IGR
quotearg_custom	iGR	IGR	IGR
quotearg_custom_mem	iGR	IGr	IGR
quote_n_mem	iGR	iGR	iGR
quote_mem	iGR	iGR	iGR
quote_n	iGR	iGR	iGR
quote	iGR	iGR	iGR
version_etc_ar	iGR	iGR	iGR
emit_bug_reporting_address	iGR	iGR	iGR
xnmalloc	iGR	IGR	IGR
xnrealloc	iGR	IGR	IGR
xrealloc	iGR	IGR	IGR
x2realloc	iGR	IGr	IGR
xcalloc	iGR	iGR	iGR
xstrdup	IGR	iGR	IGR
rpl_calloc	IGr	IGr	IGr
c_isalnum	iGR	—	—
c_isalpha	iGR	—	—
c_isascii	iGR	—	—
c_isblank	iGR	—	—
c_iscntrl	iGR	—	—
c_isdigit	iGr	—	—
c_isgraph	iGR	—	—
c_islower	iGR	—	—
c_isprint	iGR	—	—
c_ispunct	iGR	—	—
c_isspace	iGR	—	—
c_isupper	IGR	—	—
c_isxdigit	iGR	—	—
c_tolower	iGR	—	—
c_toupper	iGR	—	—
hard_locale	IGr	IGR	IGR

[7] IDA Pro marks unused or unreachable functions in a maroon color.

based on a control flow graph show relatively a lower accuracy than those with a linear sweep. Reporting reachable functions alone is orthogonal to the capability of recognizing functions because it is a matter of a strategic or idiosyncratic choice to help reversing.

```
; Actual function starts
0x286901 PUSH    RBP
0x286902 MOV     RBP,RSP
0x286905 MOV     qword ptr [RBP + local_20],RDI
0x286909 MOV     RAX,qword ptr [RBP + local_20]
...
0x286971 ADD     RAX,RDX
0x286974 SHL     RAX,0x2
0x286978 ADD     RAX,RDX
0x28697b XOR     RAX,qword ptr [RBP + local_10]
0x28697f POP     RBP

; Incorrect function identification: XREF 0x3d0500(*)
0x286980 c3      RET
; Actual function ends

; Frame Descriptor Entry
0x3d04f8 ddw     14h          (FDE) Length
0x3d04fc ddw     cie_003cfc68 (FDE) CIE Reference
     Pointer
0x3d0500 ddw     FUN_00286980 (FDE) PcBegin
0x3d0504 dq      160h         (FDE) PcRange
0x3d050c uleb128 0h           (FDE) Augmentation
     DataLen
```

Listing 6: Example of an erroneous function identification case with an FDE by Ghidra.

Case Study: Over Reporting (Ghida). As described in §4.4, Ghidra utilizes FDE information to explore more functions. However, it is worthwhile mentioning that FDE references may not always point to correct function locations. Listing 6 shows the case of a Ghidra's false positive with a frame descriptor entry that points to a location in the middle of a single instruction. This example comes from the `libcrypto.so-gcc-O0` binary, compiled with gcc and -O0 option. We do not confirm all individual cases, but there are a number of such cases that FDEs complicate the beginning of a function. Ghidra indeed tends to discover a little more functions than the number of ground truth. The behavior of under/over reporting well explains the reason why IDA Pro exhibits a (relatively) high precision and a low recall while Ghidra shows the opposite (Table 8).

5 EVALUATION

In this section, we revisit prior work with our dataset. The experiment has been done with a 64-bit Ubuntu 16.04 system equipped with Intel(R) Xeon(R) E5-2658 v3 CPU (with 16 2.20 GHz cores) and 64 GB RAM.

Corpus. We have collected 16 different binaries from the SPEC2017 benchmark [26] and four binaries from three utilities of our choice, and then generated 152 different x64 ELF binaries in total with two compilers (`gcc` 5.4 and `clang` 6.0.1) and four different optimization levels (O0-O3), excluding a `clang` version of `blender_r` and `parest_r` because of compilation errors (Table 6). It is worth noting that our dataset is valid after normalization because only 753 NFs (less than 1%) in a test set (80.5K) are shown in a train set (796.1K).

Results of Function Identification Tools. As shown in Table 7, we utilize three rule-based tools (i.e., IDA Pro 7.1, Ghidra 9.1.2, and Nucleus 0.65) and two ML-embedded tools (i.e., ByteWeight and LEMNA implementation for Shin et al's RNN) for recognizing function starts. Table 8 summarizes our empirical results with our own dataset as selected in Table 6. Even though we question the reasonableness of the current metrics in Section 4.4, we have used the same metrics for direct comparison with prior evaluations.

Table 6: Summary of our test suite. The numbers in a parenthesis represent the number of binaries with a different set of compilers (gcc and clang) and optimization levels (-O[0-3]).

TestSuite	Count	Binary Set
SPEC2017	16 (120)	500.perlbench_r, 502.gcc_r, 505.mcf_r, 520.omnetpp_r, 523.xalancbmk_r, 525.x264_r, 531.deepsjeng_r, 541.leela_r, 557.xz_r, 508.namd_r, 510.parest_r, 511.povray_r, 519.lbm_r, 526.blender_r, 538.imagick_r, and 544.nab_r
Utilities	4 (32)	nginx 1.16.1, vsftpd 3.0.3, and openssl 1.1.1f (libssl.so, libcrypto.so)

Table 7: Corpus for evaluating cutting-edge function detection tools. Train sets are merely for machine learning techniques. (*) represents the retrained model of ByteWeight.

Tool	Train set	Test set
Byteweight	GNU utils	SPEC2017, Our utilities
Byteweight*	SPEC2017	SPEC2017 (10-fold), Our utilities
Shin:RNN	SPEC2017	Our utilities
IDA Pro 7.2	N/A	SPEC2017, Our utilities
Ghidra 9.1.2	N/A	SPEC2017, Our utilities
Nucleus	N/A	SPEC2017, Our utilities

First, we have applied the publicly available model (the latest version as of writing) from ByteWeight [8] to our corpus. The F1 value with the released model is around 61.7. Our evaluation merely includes the binaries compiled with `gcc` because testing the binaries with `clang` is unfair that the existing model would have not learned any signature from that compiler. It indicates that GNU utilities do not offer diverse cases due to a considerable number of redundant NFs as discussed in Section 4.2. Next, we have retrained ByteWeight [5] (taking a week or so) using SPEC2017 and retested it with our dataset (both compiled with `gcc` alone). Note that three binaries of our test set have been crashed while processing, and thus are excluded. All metrics have considerably increased after retraining (78.0 on average); however, the F1 values of the newly trained model across optimized binaries (O1-3) still remain below 70 (ByteWeight* in Table 8). We employ 10-fold cross validation for the model. Besides, we have adopted LEMNA's re-implementation [14] and the hyperparameters for the Shin's RNN model because the source code of original work is currently unavailable. With the test set of our chosen utilities (32 binaries or 80.5K functions) and the training set of SPEC2017, the RNN model achieves an F1 of 90.1.

Finally, we have run the whole set (152 binaries or 796.1K functions in total) for rule-based tools including Ghidra [9], IDA Pro [16] and Nucleus [4], and obtained F1 values of 96.0, 93.4, and 90.4, respectively. Notably, IDA Pro demonstrates the highest precision (99.55) that means a false positive rate is quite low whereas Ghidra demonstrates the highest recall (98.65) that means a false negative

Table 8: Experimental results of function starts using a precision (P), recall (R), and F1 value from various tools. GT represents a ground truth discovered in a symbol table. ByteWeight* shows our empirical results after retraining with SPEC2017.

Tool		GT	TP	FP	FN	P	R	F1
ByteWeight		514,082	309,781	180,777	204,301	63.15	60.26	**61.67**
gcc		514,082	309,781	180,777	204,301	63.15	60.26	61.67
	O0	193,094	188,884	19,043	4,210	90.84	97.82	94.20
	O1	108,964	56,655	55,463	52,309	50.53	51.99	51.25
	O2	107,673	31,833	50,604	75,840	38.61	29.56	33.49
	O3	104,351	32,409	55,667	71,942	36.80	31.06	33.68
ByteWeight*		463,323	332,576	56,655	130,747	85.44	71.78	**78.02**
gcc		463,323	332,576	56,655	130,747	85.44	71.78	78.02
	O0	142,603	141,774	156	829	99.89	99.42	99.65
	O1	108,964	68,599	19,607	40,365	77.77	62.96	69.58
	O2	107,539	63,630	18,671	43,909	77.31	59.17	67.04
	O3	104,217	58,573	18,221	45,644	76.27	56.20	64.72
Shin:RNN		80,532	69,334	4,034	11,198	94.50	86.09	**90.10**
clang		41,267	35,153	1,164	6,114	96.79	85.18	90.62
	O0	11,647	11,476	52	171	99.55	98.53	99.04
	O1	11,637	9,263	346	2,374	96.40	79.60	87.20
	O2	8,998	7,194	357	1,804	95.27	79.95	86.94
	O3	8,985	7,220	409	1,765	94.64	80.36	86.91
gcc		39,265	34,181	2,870	5,084	92.25	87.05	89.58
	O0	11,657	11,477	90	180	99.22	98.46	98.84
	O1	9,349	8,351	499	998	94.36	89.33	91.77
	O2	9,305	7,304	1,137	2,001	86.53	78.50	82.32
	O3	8,954	7,049	1,144	1,905	86.04	78.72	82.22
Ghidra		796,069	785,333	54,131	10,736	93.55	98.65	**96.03**
clang		281,987	276,296	47,134	5,691	85.43	97.98	91.27
	O0	92,718	92,330	2,468	388	97.40	99.58	98.48
	O1	92,226	90,282	15,006	1,944	85.75	97.89	91.42
	O2	48,614	46,933	14,744	1,681	76.09	96.54	85.11
	O3	48,429	46,751	14,916	1,678	75.81	96.54	84.93
gcc		514,082	509,037	6,997	5,045	98.64	99.02	98.83
	O0	193,094	192,523	2,318	571	98.81	99.70	99.26
	O1	108,964	107,683	1,663	1,281	98.48	98.82	98.65
	O2	107,673	106,055	1,492	1,618	98.61	98.50	98.55
	O3	104,351	102,776	1,524	1,575	98.54	98.49	98.51
IDA Pro		796,069	699,606	3,194	96,463	99.55	87.88	**93.35**
clang		281,987	263,385	3,102	18,602	98.84	93.40	96.04
	O0	92,718	92,600	3	118	100.00	99.87	99.93
	O1	92,226	84,920	1,044	7,306	98.79	92.08	95.31
	O2	48,614	43,037	1,025	5,577	97.67	88.53	92.88
	O3	48,429	42,828	1,030	5,601	97.65	88.43	92.81
gcc		514,082	436,221	92	77,861	99.98	84.85	91.80
	O0	193,094	191,757	3	1,337	100.00	99.31	99.65
	O1	108,964	89,288	10	19,676	99.99	81.94	90.07
	O2	107,673	79,085	47	28,588	99.94	73.45	84.67
	O3	104,351	76,091	32	28,260	99.96	72.92	84.32
Nucleus		796,069	750,012	112,936	46,057	86.91	94.21	**90.42**
clang		281,987	264,819	72,945	17,168	78.40	93.91	85.46
	O0	92,718	91,872	8,810	846	91.25	99.09	95.01
	O1	92,226	82,431	21,687	9,795	79.17	89.38	83.97
	O2	48,614	45,346	21,191	3,268	68.15	93.28	78.76
	O3	48,429	45,170	21,257	3,259	68.00	93.27	78.66
gcc		514,082	485,193	39,991	28,889	92.39	94.38	93.37
	O0	193,094	188,789	8,610	4,305	95.64	97.77	96.69
	O1	108,964	104,985	7,330	3,979	93.47	96.35	94.89
	O2	107,673	95,897	11,481	11,776	89.31	89.06	89.19
	O3	104,351	95,522	12,570	8,829	88.37	91.54	89.93

rate is quite low. One plausible reason is that the way of detecting functions is different: IDA Pro tends to identify functions based on control flow graphs whereas Ghida attempts to detect as many functions as possible even with FDEs (Section 4.6). Similarly, part of the reason that IDA Pro has relatively a low recall is because it does not report unreachable (either unused or indirectly reachable) code. Another interesting finding is that IDA Pro reports a higher F1 of 96.0 for clang-generated binaries than that of 91.3 from Ghidra, however, Ghidra shows a better performance (98.9) than IDA Pro (91.8) for gcc-generated binaries.

Insights of Empirical Results. Taking a close look at the experimental results with our efforts to answer the research questions we have raised in Section 4.1, the following recaps our insights.

First, in general, state-of-the-art function detection tools work very well to which no optimization has been applied. This means a compiler toolchain emits an apparent signature like function prologues and epilogues. Second, not a single tool dominates all the others. The performance of a rule-based tool may vary depending on a signature database. It also indicates that the performance of the same tool may fluctuate according to its own version or dataset to be tested. Third, it is difficult to claim that an ML-centric approach is yet superior to rule-based approaches although the approach obviously has its own strength. Our empirical results show that both rule-based and ML-oriented approaches complement each other. For example, a deep learning technique could play a pivotal role in learning locally missing functions. Fourth, the current metrics (i.e., precision, recall, and F1 value) for function detection may not be reasonable because it is likely that they may not reflect idiosyncrasies of detection tools or various compiler optimization techniques. Fifth, the capability of identifying functions depends on each tool's strategic or peculiar choice. Overall, it is difficult to conclude that i) a function detection problem has been fully resolved, and ii) a better metric may be needed, which we leave for our future research.

6 DISCUSSION AND LIMITATIONS

This section discusses suggestions, future research and limitations.

Representativeness of Corpus. It is non-trivial to choose a sufficient number of binaries with a variety of different cases due to the nature of software diversity. To this regard, we carefully selected our test suite of SPEC2017 [26] that encompasses varying application area (e.g., interpreter, compiler, benchmark, compression, converter, and AI-centered searching algorithms), programming languages (e.g., C, C++), and sizes (e.g., ranging from 3 to 1,304 KLOC). We merely include `rate_int` suites [20] (ending with `_r`) because `speed_int` suites (ending with `_s`) consist of many duplicate functions to avoid distorting the results for our purpose. We also embody popular Web/FTP servers, and a cryptographic library with a manually-written assembly in our corpus. Although the number of binaries in our corpus is 152, the number of functions is $796K$ in our dataset, which is 80% larger than that of whole functions ($446K$) in GNU utilities [5]. However, it is possible that our investigation may have undiscovered cases, missing insights, or even biased results.

Functions in a Virtual Table. A binary written in an object-oriented language such as C++ typically contains a number of virtual tables, each of which consists of a group of function pointers. A class inheritance forms a hierarchical structure (i.e., super/parent class, sub/child class). Under this setting, by nature, different functions from separate virtual tables[8] may point to the same reference (or code region) because the implementation must be identical. Hence, it is common that a single function code can correspond to multiple function names ($1:n$ mapping). We may need a policy to handle such cases; say, what if a tool can identify F_1 but not F_2 where the two functions share the same code region?

[8] C++ employs function name mangling so that a linker can separate common name in a different name space.

Future Research Directions. We suggest several research directions in the field of function identification. First, seeking a representative binary corpus is necessary. The official corpus should cover varying code constructs and corner cases, which consists of a set of binaries compiled with different compilers (and versions if possible) and optimization levels, which must be thoroughly studied beforehand (e.g., number of redundant functions, non-returning functions, non-continuous functions, known ground truth based on debugging information). Second, defining an elaborate metric is required rather than the current scheme that an F1 value dominates alone. For example, we can conceive a sub-metric to measure the discovery of non-returning functions, unreachable functions, or overlapping functions that point to the same code region. The additional metrics would better explain the strengths and weaknesses of each tool. Third, developing a hybrid model of both rule-based and ML-oriented approaches seems a promising direction for function detection because our experimental results glimpse a complementary relationship between the two. As shown in §5, an RNN was capable of successfully finding uncommon rules for the beginning of a function from manually written assembly files.

7 CONCLUSION

In this paper, we rethink a function identification problem. with the existing rule-based and ML-centric approaches. Notable results with high F1 values from prior works seemingly convey the impression that identifying functions in a stripped binary has been (almost) addressed. To this end, we attempt to fill the void of what may have been overlooked by taking a close look at prior datasets, evaluations, common metrics, and the behavior of a tool. With varying case studies, our major findings support the followings: i) a common dataset for function detection study, GNU Utilities, is not appropriate to showcase the effectiveness of a tool, ii) it is difficult to say that ML-oriented approaches surpass rule-based ones, and iii) the capability of function recognition requires the understanding of a tool's behavior. We conclude that, based on complex binaries in the wild, the field of function recognition has a room for improvement such as better metrics and dataset for fair comparison. Moreover, we believe that machine learning plays an important role for what rule-based approaches (i.e., signature based techniques) cannot cover.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Fabian Yamaguchi, for their constructive feedback. This work was supported by the Basic Science Research Program through National Research Foundation of Korea (NRF) grant funded by the Ministry of Education of the Government of South Korea (No. 2021R1F1A105248411). Also, it was supported, in part, by the NSF award CNS-1563848 and CNS-1749711 ONR under grant N00014-18-1-2662, N00014-15-1-2162, N00014-17-1-2895, DARPA AIMEE HR00112090034 and SocialCyber HR00112190087, ETRI IITP/KEIT[2014-3-00035], and gifts from Facebook, Mozilla, Intel, VMware and Google. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsor.

REFERENCES

[1] National Security Agency. 2021. Ghidra Features. https://github.com/NationalSecurityAgency/ghidra/tree/da94eb86bd2b89c8b0ab9bd89e9f0dc5a3157055/Ghidra/Features/Base/data.
[2] Jim Alves-Foss and Jia Sone. 2019. Function Boundary Detection in Stripped Binaries. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*.
[3] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. 2016. An In-Depth Analysis of Disassembly on Full-Scale X86/X64 Binaries. In *Proceedings of the 25th USENIX Security Symposium (Security)*. Austin, TX.
[4] Dennis Andriesse, Asia Slowinska, and Herbert Bos. 2017. Compiler-Agnostic Function Detection in Binaries. In *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (EuroSP*. Paris, France.
[5] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. ByteWeight: Learning to Recognize Functions in Binary Code. In *Proceedings of the 23rd USENIX Security Symposium (Security)*. San Diego, CA.
[6] Berkeley. 2008. BitBlaze: Binary Analysis for Computer Security. http://bitblaze.cs.berkeley.edu/.
[7] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*. Snowbird, UT.
[8] ByteWeight. 2014. ByteWeight: Recognizing Functions in Binaries. http://security.ece.cmu.edu/byteweight/.
[9] NSA's Research Directorate. 2019. Ghidra. https://ghidra-sre.org/.
[10] NSA's Research Directorate. 2019. Ghidra function start signature DB. https://github.com/NationalSecurityAgency/ghidra/blob/master/Ghidra/Processors/x86/data/patterns/x86-64gcc_patterns.xml.
[11] Alessandro Di Federico, Mathias Payer, and Giovanni. 2017. rev.ng: a unified binary analysis framework to recover CFGs and function boundaries. In *Proceedings of the 2017 International Conference on Compiler Construction (CC)*. Austin, TX.
[12] Free Software Foundation. 2014. Coreutils - GNU core utilities. https://www.gnu.org/software/coreutils/.
[13] Ivan Gotovchits. 2021. BAP facility for indentifying code entry points. https://opam.ocaml.org/packages/bap-byteweight/.
[14] Wenbo Guo, Dongliang Mu5, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. 2018. LEMNA: Explaining Deep Learning based Security Applications. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*. Toronto, ON, Canada.
[15] Hex-Rays. 2005. IDA Fast Library Identification and Recognition Technology. https://www.hex-rays.com/products/ida/tech/flirt/.
[16] Hex-Rays. 2005. IDA Pro Disassembler. https://www.hex-rays.com/idapro/.
[17] Jan Hubicka, Andreas Jaeger, Michael Matz, and Mark Mitchell. 2003. System V Application Binary Interface. http://www.ucw.cz/~hubicka/papers/abi/.
[18] Emily R. Jacobson, Nathan Rosenblum, and Barton P. Miller. 2011. Labeling library functions in stripped binaries. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software (PASTE)*. Seattle, USA.
[19] Abbas Khalili and Jiahua Chen. 2007. Variable selection in finite mixture of regression models. *J. Amer. Statist. Assoc.* (2007).
[20] Ankur Limaye and Tosiron Adegbija. 2018. A Workload Characterization of the SPEC CPU2017 Benchmark Suite.
[21] Rui Qiao and R Sekar. 2017. Function Interface Analysis: A Principled Approach for Function Recognition in COTS Binaries. In *Proceedings of the 47th International Conference on Dependable Systems and Networks (DSN)*. Denver, USA.
[22] Radare2. 2009. Libre and Portable Reverse Engineering Framework. http://radare.re/n/.
[23] Nathan Rosenblum, Xiaojin Zhu, Barton Miller, and Karen Hunt. 2008. Learning to Analyze Binary Computer Code. *Association for the Advancement of Artificial Intelligence (AAAI)* (2008).
[24] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing Functions in Binaries with Neural Networks. In *Proceedings of the 24th USENIX Security Symposium (Security)*. Washington, DC.
[25] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*. San Jose, CA.
[26] SPEC Standard Performance Evaluation Corporation. 2017. SPEC CPU2017 Benchmark. https://www.spec.org/cpu2017.
[27] Robert Tibshirani, Michael Saunders, Saharon Rosset, Ji Zhu, and Keith Knight. 2005. Sparsity and Smoothness via the Fused Lasso. *Journal ofthe Royal Statistical Society: Series B (Statistical Methodology)* (2005).
[28] Shuai Wang, Pei Wang, and Dinghao Wu. 2017. Semantics-Aware Machine Learning for Function Recognition in Binary Code. In *Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME*. Shanghai, China.