Revisiting Function Identification with Machine Learning

Hyungjoon Koo, **Soyeon Park** and **Taesoo Kim**
Georgia Institute of Technology
{hkoo37, soyeon, 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 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 function detection in general, iv) 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 our community first has to seek a better metric for fair comparison in order to make advances in the field of function identification.

1 Introduction

Function identification (or recognition) serves as a basis for reversing executable binaries and for many applications, including control flow integrity (CFI), binary similarity analysis, binary transformation (i.e., randomization, re-optimization), type inference, and vulnerability detection. Likewise, a majority of binary analysis tools (i.e., BAP [Brumley *et al.*, 2011], angr [Shoshitaishvili *et al.*, 2016], radare [Radare2, 2009], IDA Pro [Hex-Rays, 2005b], Ghidra [Directorate, 2019a], rev.ng [Federico *et al.*, 2017]) often begin with function detection for further analysis by default because a binary function provides a logical unit to understand and analyze the high-level semantics of a low-level binary. Obtaining a function boundary upon the availability of symbol or debugging information is trivial. However, it becomes drastically challenging when the information is stripped off, which is more common than not in practice.

A simple means of function recognition is to linearly disassemble all code (e.g., `objdump`), followed by applying function signature matching such as a function prologue and epilogue available. However, it suffers from robustness either when a predefined pattern lacks such a signature (i.e., highly optimized functions) or when code and data are intermixed. 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. However, indirectly reachable (or unreachable) functions may not always be statically identified. Despite such challenges, many prior works have repeatedly demonstrated remarkable results with high precision and recall (i.e., mostly 96% or above). Recent advances harness a machine learning technique (i.e., RNN), which claims to achieve even higher accuracy [Shin *et al.*, 2015; Guo *et al.*, 2018].

In this paper, we (re-)evaluate and challenge recent advances in function identification from a different angle, particularly focusing on proposals that utilize machine learning techniques. Note that our objective is neither to verify the correctness of prior evaluations nor to rank the existing approaches 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 datasets, metrics, and evaluations with the following four research perspectives in mind: i) appropriateness of the previous datasets (e.g., GNU utilities), ii) re-interpretation of the prior evaluations, iii) effectiveness of ML-oriented techniques, and iv) reasonableness of the current metrics, for function identification.

The following summarizes the key contribution of our paper. First, we investigate GNU utilites because all subsequent works (but Nucleus) have employed them for their evaluations after the initial release by ByteWeight [Bao *et al.*, 2014]. With normalization, we have discovered quite a few redundant functions (sorely 12.1% remains unique), which cannot prevent overfitting. Although Nucleus first asserted the bias of the dataset with a limited assessment, we have fully quantified the claim. Second, our finding shows that the accuracy of LEMNA [Guo *et al.*, 2018] (re-implementation of Shin's RNN [Shin *et al.*, 2015]) comes from a different metric (i.e., a series of true negatives per each following byte).Third, the evaluation with our own dataset shows that not a single tool dominates all the others. Although an ML-oriented approach has its own strength; e.g., automating the implementation of function identification algorithms, the existing proposals still lack scientific outcomes to confidently claim that they indeed are superior to deterministic and popular approaches like IDA or Ghidra. Fourth, we discover a handful of cases to determine the correctness of function boundary that the current metrics cannot reasonably cover, necessitating that a better metric be explored for more a fair comparison. Overall, our thorough evaluation with our own dataset, which will be publicly available upon publication, shows that a function identification problem requires further study.

2 Background and Related Work

2.1 Problem Definition of Function Identification

A function recognition problem aims to discover a set of functions in case no symbol or debugging information is readily available, which includes both i) function starts and ii) function boundaries (both starts and ends). Analyzing malware or binaries that have stripped off such information is common.

2.2 Evaluation Metrics

Let a set of true positives (i.e., aligned with a ground truth), 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, FP, and FN, respectively. The following defines a precision (P), recall (R), $F1$ score, and accuracy (A).

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

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

Note that a high precision means the rate of incorrectly identified functions (FP) is low, whereas 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.

2.3 Related Work

Deterministic Approach. UNSTRIP [Jacobson *et al.*, 2011] generates semantic descriptors (i.e., system calls and concrete argument values) that represent library functions as a fingerprint for further function identification. Nucleus [Andriesse *et al.*, 2017] 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 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 and Sekar, 2017] 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 dataflow properties (i.e., parameter passing). Jima [Alves-Foss and Sone, 2019] is a tool suite that incorporates a series of analysis algorithms for function boundary detection, including exception handling, jump pointer, tail call chain, and missing function detection (i.e., gaps between functions). IDA Pro [Hex-Rays, 2005b] is a very popular disassembly tool equipped with both decompilation and debugging features for code analysis; however, its internal heuristics (i.e., pattern database) for function detection remain proprietary and thus unknown[1]. Ghidra [Directorate, 2019a] 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 [Directorate, 2019b].

Machine Leaning Based Approach. One of the early works [Rosenblum *et al.*, 2008] 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 binary code. Byteweight [Bao *et al.*, 2014] 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). FID [Wang *et al.*, 2017] 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 (i.e., stack registers) and memory access behavior (i.e., memory read), converting them into numeric feature vectors for a classifier. Meanwhile, [Shin *et al.*, 2015] 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 demonstrate a very high precision and recall. Recently, LEMNA [Guo *et al.*, 2018] introduces the first explanation model for a deep learning based security application (i.e., using an RNN model). It integrates fused lasso [Tibshirani *et al.*, 2005] for handling a feature dependency problem with a mixture regression model [Khalili and Chen, 2007] that achieves an accurate approximation for a local decision boundary.

3 Challenges of Function Identification

A binary function that resides in a code section differs from a human-written function that conveys semantics. Every binary function originates from a function i) defined by a user (i.e., source code), ii) generated by a compiler (i.e., stack canary check), or iii) inserted by a linker (i.e., CRT function).

The common challenges for function detection are well-known, mainly due to compiler optimizations and code regions intermixed by code and data. First, code optimization often blurs a clear signature of a function prologue and epilogue, rendering its boundary detection less straightforward because ① a function can be inlined to be part of another for performance;

[1]Note that IDA ships with a known function identification algorithm, dubbed FLIRT [Hex-Rays, 2005a] that maintains a signature database of each function for a standard library, however, it cannot be applied to general function identification.

Table 1: Summary of our test suite. The numbers in () represent the number of binaries with a different set of a compiler and optimization.

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 2: Summary of cutting-edge function detection tools. (*) represents the retrained model of ByteWeight.

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

② a call invocation happens at the end of a procedure (i.e., tail call), replacing it with a single jump (instead of `pop` and `ret`) without returning to an original caller; ③ a single routine may be split into multiple locations (non-contiguous function); ④ different function symbols can point to the same address (i.e., identical implementation); and ⑤ compiler-generated code or compiler-specific heuristics may render function identification opaque. Second, a compiler can mix a jump table as data within a function for indirect transfer that complicates a linear disassembly task (commonly seen in ARM or Windows binaries). Other reasons include multi-entry functions (i.e., calls to the middle of a function), non-returning functions (i.e., ending with a call), and code from manually written assembly.

4 Rethinking Function Detection Problem

In this section, we describe the function identification problem mainly focusing on four research questions. We aim neither to simply rank the existing tools by comparison nor to verify the correctness of prior evaluations. Instead, we attempt to fill the gaps that may have been overlooked or misinterpreted.

Test Suite. We have collected 16 different binaries from the SPEC2017 benchmark [Standard Performance Evaluation Corporation, 2017] 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 1). Note that the binaries ending with `_s` in SPEC2017 are ruled out due to almost identical function list.

Function Identification Tool. As shown in Table 2, we utilize three deterministic tools (IDA, Ghidra, Nucleus) and two ML-embedded tools (ByteWeight, LEMNA implementation of Shin et al's RNN) for recognizing function starts.

4.1 Research Questions

We revisit prior approaches to answer the following research questions that focus on ① appropriateness of dataset, ② reinterpretation of prior evaluations, ③ effectiveness of ML techniques, and ④ rethinking of metrics, for function identification. We also conduct extra experiments if required.

- **RQ1.** Is the previous dataset (i.e., 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.** Are recent advances with an ML-centered approach (i.e., deep learning) superior to a deterministic one?
- **RQ4.** Is the current metric (i.e., precision, recall and F1) fair enough to measure function identification in general?

4.2 Appropriateness of Dataset

Table 3 shows a comparison of prior approaches for function identification at a glance. After the first release of ByteWeight's GNU utilities [ByteWeight, 2014] (16 `binutils`, 104 `coreutils`, 9 `findutils`), all subsequent works but Nucleus employ the same dataset for their evaluations. Nucleus has first claimed that they are too biased to be generalized with a limited assessment of the assertion.

We have quantified the bias of the dataset, 129 GNU utilities, adopted by ByteWeight. For simplicity, we sorely focus on x64 binaries compiled with `gcc`. Table 4 shows 10 different groups utilized in ByteWeight for 10-fold cross validation.

```
; // binutils - ar
; void yyset_lineno(int line_number) {
; yylineno = line_number;
; }
0x432273:  push   rbp
0x432274:  mov    rbp,rsp
0x432277:  mov    DWORD PTR [rbp-0x4],edi
0x43227a:  mov    eax,DWORD PTR [rbp-0x4]
0x43227d:  mov    DWORD PTR [rip+0x378a81],eax
0x432283:  pop    rbp
0x432284:  ret
; // binutils - as
; static void set_allow_index_reg (int flag) {
; allow_index_reg = flag;
; }
0x4049c8:  push   rbp
0x4049c9:  mov    rbp,rsp
0x4049cc:  mov    DWORD PTR [rbp-0x4],edi
0x4049cf:  mov    eax,DWORD PTR [rbp-0x4]
0x4049d2:  mov    DWORD PTR [rip+0x30fbd8],eax
0x4049d8:  pop    rbp
0x4049d9:  ret
```

Listing 1: Example of an identical function pair after normalization.

ByteWeight performs normalization of a function as a preprocessing step before generating a weighted tree; that is, it converts both an immediate number and target of a call/jump instruction into a generalized value to improve a recall. After normalization[2], we found that only 17.6K (12.1%) out of the whole 146K functions remained unique normalized functions (NFs). For example, 19.8K NFs (91.4%) have been discovered in a train set (20.7K functions) when selecting Group 9 as a test set in Table 4. This indicates cross validation cannot avoid an overfitting because of too many redundant data. The redundancy mainly arises from a static library in common during compilation: the `coreutils` consists of 106 small binaries that employ `libcoreutils.a`, including 776 common functions from 257 object files. Another interesting finding is that there are a considerable number of NFs even between different functions from different binaries. For instance, Listing 1

[2]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.

Table 3: Comparison of the existing works for function boundary detection. (*) indicates the work that has been included for our evaluation.

Approach	Tool	Artifacts	Year	Dataset	Arch	Type	Compiler	OptLevel	# Binaries	Compared To
Non-ML	Nucleus* [Andriesse *et al.*, 2017]	Y	2017	SPEC2006, ngnix, lighttpd, opensshd, vsftpd, exim	x86/x64	ELF/PE	clang/VS	O0-O3	476	Dyninst, ByteWeight, IDA
	Function-interface [Qiao and Sekar, 2017]	N	2017	GNU Utils, SPEC2006, GLIBC	x86/x64	ELF	clang/gcc	O0-O3	2,488	ByteWeight, Shin:RNN
	Jima [Alves-Foss and Sone, 2019]	Y	2019	GNU Utils, SPEC2017, Chrome	x86/x64	ELF	clang/gcc/icc	O0-O3	2,860	ByteWeight, Shin:RNN, IDA Free, Ghidra, Nucleus
ML/DL	Nathan:CRF [Rosenblum *et al.*, 2008]	N	2007	Unknown	x86	ELF/PE	gcc/icc/VS	Unknown	1,171	N/A
	ByteWeight* [Bao *et al.*, 2014]	Y	2014	GNU Utils	x86/x64	ELF/PE	clang/gcc	O0-O3	2,200	Dyninst, ByteWeight, BAP, IDA
	Shin:RNN* [Shin *et al.*, 2015]	N	2015	GNU Utils	x86/x64	ELF/PE	clang/gcc	O0-O3	2,200	ByteWeight
	FID [Wang *et al.*, 2017]	N	2017	GNU coreutils	x86/x64	ELF	clang/gcc/icc	O0-O3	4,240	IDA, ByteWeight
	LEMNA* [Guo *et al.*, 2018]	Y	2018	GNU Utils	x64	ELF	gcc	O0-O3	2,200	N/A

Table 4: 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

depicts two binary functions that are identical after normalization. The source code of those functions (line 1-5, 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. 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).

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 showcase a remarkable outcome compared to the existing works (Table 3).

We believe the reported empirical results are accurate and reproducible, but, as discussed in Section 4.2, we claim that one reason for a high detection rate partially stems from an inappropriate corpus. 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 6 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 6), 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.

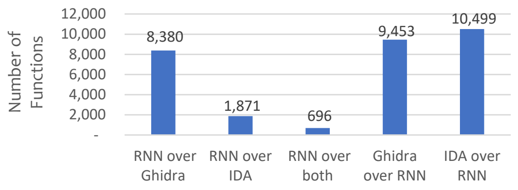

Figure 1: Comparison of the number of true functions between different tools (i.e., RNN VS deterministic approaches).

Table 5: Non-returning function detection across different 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%

4.4 On the Effectiveness of ML Techniques

This section describes the effectiveness of ML-centric approaches, including deep learning (i.e, Shin's RNN). Figure 1 illustrates a simple comparison between the number of true functions discovered by each tool. For example, Shin's RNN approach discovers 8,380 and 1,871 functions more than Ghidra and IDA, respectively (See Table 6 for details). Meanwhile, Ghidra and IDA discover 9,453 and 10,499 functions more than the RNN.

Case Study: Non-returning Functions

In particular, we investigate all non-returning functions[3] (9,409 or 1.2%) from our dateset, ending with `call`, `jump`, or `__exit` such as Listing 2, Listing 3, and Listing 4. Table 5 concisely shows that ML-oriented approaches miss more functions than deterministic techniques.

Case Study: 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 2 illustrates the code snippet (line 1-14) 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

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

on the argument (i.e., line 7, 10, and 13 otherwise). Although this example is slightly different from a typical function inlining case in that a function symbol resides in a symbol table, deterministic binary analysis tools regard each branch function as part of `process_post_login_req`.

```
 1 static void
 2 process_post_login_req(struct vsf_session* p_sess) {
 3   char cmd;
 4   /* Blocks */
 5   cmd = priv_sock_get_cmd(p_sess->parent_fd);
 6   if (tunable_chown_uploads && cmd == PRIV_SOCK_CHOWN)
 7     cmd_process_chown(p_sess);
 8   ...
 9   else if (cmd == PRIV_SOCK_PASV_CLEANUP)
10     cmd_process_pasv_cleanup(p_sess);
11   ...
12   else
13     die("bad request in process_post_login_req");
14 }
15
16 ; process_post_login_req(vsf_session *p_sess)
17 0xAC10   push     rbx
18 0xAC11   mov      rbx, p_sess
19 0xAC14   mov      edi, [p_sess+180h] ; fd
20 0xAC1A   call     priv_sock_get_cmd
21 ...
22 0xAC3F   lea      rcx, jpt_AC4D
23 0xAC46   movsxd   rax, ds:(jpt_AC4D - 16C38h)[rcx+rax*4]
24 0xAC4A   add      rax, rcx
25 0xAC4D   jmp      rax  ; jump table
26 0xAC52   pop      p_sess
27 0xAC53   jmp      cmd_process_get_data_sock
28 0xAC55   lea      rdi, aBadRequestInPr
29 0xAC5C   pop      p_sess
30 0xAC5D   jmp      die
31 ...
32 0xAC80   pop      p_sess
33 0xAC81   jmp      cmd_process_pasv_cleanup
34 ...
35 ; cmd_process_pasv_cleanup(vsf_session *p_sess)
36 0xAD30   push     rbx
37 0xAD31   mov      rbx, p_sess
38 0xAD34   call     vsf_privop_pasv_cleanup
39 0xAD39   mov      edi, [p_sess+180h]
40 0xAD3F   mov      esi, 1
41 0xAD44   pop      p_sess
42 0xAD45   jmp      priv_sock_send_result
```

Listing 2: Example of a function and its disassembly after optimization. The function `cmd_process_pasv_cleanup` has been discovered by an RNN alone over deterministic approaches.

4.5 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 scientific means to measure the effectiveness of function identification. We provide a handful of case studies to rethink the suitability of the current metrics for function detection.

Case Study: Non-continuous Functions

Listing 3 shows the code snippet (line 1-8) and its disassembly from `imagick_r-amd64-gcc-O3`. 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`[4]). 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.

```
 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.

In a similar vein, going back to Listing 2, 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. Likewise, an `._eh_frame` section (even in a stripped binary) follows a DWARF format by default, storing call frame information (CFI) for an exception handling routine. 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.

A state-of-the-art disassembler such as Ghidra harnesses such FDEs to identify a function, sometimes resulting in discovering more functions that may not reside in a symbol table

[4]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-O3`) holds a single function symbol.

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) does not hold. We have found that there are 13,380 such functions in the above binary, which significantly increases the number of false positives for Ghidra and Nucleus. Under the current scheme of precision and recall, the F1 value of both Ghidra and Nucleus (96.0 and 90.4 in Table 6) 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 impacts on the final ranking. It is worthwhile to mention that referring FDE may point to an incorrect function location (i.e., an FDE pointing to a location in the middle of a single instruction).

5 Evaluation

Table 6 summarizes our empirical results with our own dataset as selected in Table 1. Even though we question the reasonableness of the current metrics in Section 4.5, we have used the same metrics for direct comparison with prior evaluations. We have applied the publicly available model from ByteWeight to our utility dataset. The F1 value is around 61.7 (our evaluation merely includes the binaries compiled with `gcc` because the existing model has not learned any signature from `clang`). It indicates that GNU utilities do not offer diverse cases due to a considerable number of redundant NFs as discussed in Section 4.2. We have retrained ByteWeight (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 (78.0 on average); however, the F1 values of the newly trained model across optimized binaries (O1-3) still remain below 70. Besides, we have adopted LEMNA's re-implementation and its hyperparameters for Shin et al.'s RNN model because the 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 deterministic tools including Ghidra, IDA Pro and Nucleus, and obtained F1 values of 96.0, 93.4, and 90.4, respectively.

6 Discussion

Taking a close look at the experimental results with our efforts to answer the research questions we have raised, the following recaps our insights. First, in general, state-of-the-art function detection tools work very well when no optimization has been applied. Second, not a single tool dominates all the others. The performance of a deterministic tool may vary depending on a signature database. Third, it is difficult to claim that an ML-centric approach is yet superior to deterministic approaches although the approach obviously has its own strength. Fourth, the current metrics (i.e., precision, recall, and F1 value) for function detection may not be reasonable due to idiosyncrasies from various compiler optimization techniques. This necessitates a better metric, which we leave for our future research. Fifth, overall, it is difficult to conclude that a function detection problem has been fully resolved. We believe that both deterministic and ML-oriented approaches complement each other. For example, deep learning could play a pivotal role in learning locally missing functions.

[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.

Table 6: 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
IDAPro		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

7 Conclusion

In this paper, we rethink the function identification problem using both deterministic and ML-centric approaches. To this end, we have attempted to re-interpret prior datasets, evaluations, and even common metrics using varying case studies.

Open Problem. Based on our major findings, we call for seeking better metrics and dataset for fair comparison in the field of function recognition.

References

[Alves-Foss and Sone, 2019] Jim Alves-Foss and Jia Sone. Function Boundary Detection in Stripped Binaries. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2019.

[Andriesse *et al.*, 2017] Dennis Andriesse, Asia Slowinska, and Herbert Bos. Compiler-Agnostic Function Detection in Binaries. In *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (EuroSP*, Paris, France, April 2017.

[Bao *et al.*, 2014] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. ByteWeight: Learning to Recognize Functions in Binary Code. In *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, August 2014.

[Brumley *et al.*, 2011] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*, Snowbird, UT, July 2011.

[ByteWeight, 2014] ByteWeight. ByteWeight: Recognizing Functions in Binaries. http://security.ece.cmu.edu/byteweight/, 2014.

[Directorate, 2019a] NSA's Research Directorate. Ghidra, 2019. https://ghidra-sre.org/.

[Directorate, 2019b] NSA's Research Directorate. Ghidra function start signature DB. https://github.com/NationalSecurityAgency/ghidra/blob/master/Ghidra/Processors/x86/data/patterns/x86-64gcc_patterns.xml, 2019.

[Federico *et al.*, 2017] Alessandro Di Federico, Mathias Payer, and Giovanni. 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, February 2017.

[Guo *et al.*, 2018] Wenbo Guo, Dongliang Mu5, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. LEMNA: Explaining Deep Learning based Security Applications. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, October 2018.

[Hex-Rays, 2005a] Hex-Rays. IDA Fast Library Identification and Recognition Technology. https://www.hex-rays.com/products/ida/tech/flirt/, 2005.

[Hex-Rays, 2005b] Hex-Rays. IDA Pro Disassembler, 2005. https://www.hex-rays.com/idapro/.

[Jacobson *et al.*, 2011] Emily R. Jacobson, Nathan Rosenblum, and Barton P. Miller. Labeling library functions in stripped binaries. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software (PASTE)*, Seattle, USA, September 2011.

[Khalili and Chen, 2007] Abbas Khalili and Jiahua Chen. Variable selection in finite mixture of regression models. *Journal of the American Statistical Association*, 2007.

[Qiao and Sekar, 2017] Rui Qiao and R Sekar. 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, June 2017.

[Radare2, 2009] Radare2. Libre and Portable Reverse Engineering Framework. http://rada.re/n/, 2009.

[Rosenblum *et al.*, 2008] Nathan Rosenblum, Xiaojin Zhu, Barton Miller, and Karen Hunt. Learning to analyze binary computer code. *Association for the Advancement of Artificial Intelligence (AAAI)*, 2008.

[Shin *et al.*, 2015] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing Functions in Binaries with Neural Networks. In *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, August 2015.

[Shoshitaishvili *et al.*, 2016] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 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, May 2016.

[Standard Performance Evaluation Corporation, 2017] SPEC Standard Performance Evaluation Corporation. SPEC CPU2017 Benchmark. https://www.spec.org/cpu2017, 2017.

[Tibshirani *et al.*, 2005] Robert Tibshirani, Michael Saunders, Saharon Rosset, Ji Zhu, and Keith Knight. Sparsity and smoothness via the fused lasso. *Journal ofthe Royal Statistical Society: Series B (Statistical Methodology)*, 2005.

[Wang *et al.*, 2017] Shuai Wang, Pei Wang, and Dinghao Wu. 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, September 2017.