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 utilities 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 \( A \).

\[
P = \frac{|TP|}{|TP| + |FP|}, \quad R = \frac{|TP|}{|TP| + |FN|}, \quad F1 = \frac{2 \times P \times R}{P + R} \quad (1)
\]

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

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 data-flow 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 compilation and debugging features for code analysis; however, its internal heuristics (i.e., pattern database) for function detection remain proprietary and thus unknown. 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 Learning 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 idolm 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 a function can be inclined to be part of another for performance;
Table 1: Summary of our test suite. The numbers in ( ) represent the number of binaries with a different set of a compiler and optimization.

<table>
<thead>
<tr>
<th>TestSuite</th>
<th>Count</th>
<th>Binary Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPEC2017</td>
<td>16 (120)</td>
<td>500.peri_bench_r, 502.gcc_r, 505.nco_r, 520.omega_r, 523.su bsaclebmk_r, 525.x264_r, 531.deepsjeng_r, 541.lcela_r, 557.xz_r, 508.rand_r, 510.parest_r, 511.povray_r, 519.blender_r, 526.blender_r, 538.imagick_r, and 544.nab_r</td>
</tr>
<tr>
<td>Utilities</td>
<td>4 (32)</td>
<td>nginx 1.16.1, vsftpd 3.0.3, and openssl 1.1.1f (libssl.so, libcrypto.so)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Tool</th>
<th>Train set</th>
<th>Test set</th>
</tr>
</thead>
<tbody>
<tr>
<td>ByteWeight</td>
<td>GNU utils</td>
<td>SPEC2017, Our utils</td>
</tr>
<tr>
<td>ByteWeight*</td>
<td>SPEC2017</td>
<td>SPEC2017 (10-fold), Our utils</td>
</tr>
<tr>
<td>Shin:RNN</td>
<td>SPEC2017</td>
<td>Our utils</td>
</tr>
<tr>
<td>IDA Pro 7.2</td>
<td>N/A</td>
<td>SPEC2017, Our utils</td>
</tr>
<tr>
<td>Ghidra 9.1.2</td>
<td>N/A</td>
<td>SPEC2017, Our utils</td>
</tr>
<tr>
<td>Nucleus</td>
<td>N/A</td>
<td>SPEC2017, Our utils</td>
</tr>
</tbody>
</table>

1. 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; 2. 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 (OO-03), 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. re-interpretation 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 binaries, 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 solely focus on x64 binaries compiled with gcc. Table 4 shows 10 different groups utilized in ByteWeight for 10-fold cross validation.

Listing 1: Example of an identical function pair after normalization.
Table 3: Comparison of the existing works for function boundary detection. (*) indicates the work that has been included for our evaluation.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Tool</th>
<th>Artifacts</th>
<th>Year</th>
<th>Dataset</th>
<th>Arch</th>
<th>Type</th>
<th>Compiler</th>
<th>OptLevel</th>
<th># Binaries</th>
<th>Compared To</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-ML</td>
<td>Nucleus*</td>
<td>Y</td>
<td>2017</td>
<td>SPEC2006, nagix, lighttpd, openssl, vsftpd, exam GNU Utils, coreutils</td>
<td>ELF</td>
<td>x86_64</td>
<td>clang/VS</td>
<td>0-0-3</td>
<td>476</td>
<td>Dyninst, ByteWeight, IDA</td>
</tr>
<tr>
<td>Function interface</td>
<td>[Andreasse et al., 2017]</td>
<td>N</td>
<td>2017</td>
<td>SPEC2006, GLIBC GNU Utils</td>
<td>ELF</td>
<td>x86_64</td>
<td>clang/gcc</td>
<td>0-0-3</td>
<td>2,488</td>
<td>ByteWeight, Shin/RNN</td>
</tr>
<tr>
<td>[Quan and Sekar, 2017]</td>
<td>[Alves-Foss and Sone, 2019]</td>
<td>Y</td>
<td>2019</td>
<td>SPEC2017, Chrome</td>
<td>ELF</td>
<td>x86_64</td>
<td>clang/gcc</td>
<td>0-0-3</td>
<td>2,860</td>
<td>ByteWeight, Shin/RNN, IDA Free, Ghidra, Nucleus</td>
</tr>
<tr>
<td>ML/DL</td>
<td>Nathan:CRF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Rosenblum et al., 2008]</td>
<td>N</td>
<td>2007</td>
<td>Unknown</td>
<td>ELF</td>
<td>x86_64</td>
<td>ebcuse/S</td>
<td>Unknown</td>
<td>1,171</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>ByteWeight*</td>
<td>[Bao et al., 2014]</td>
<td>Y</td>
<td>2014</td>
<td>GNU Unix</td>
<td>ELF</td>
<td>x86_64</td>
<td>clang/gcc</td>
<td>0-0-3</td>
<td>2,200</td>
<td>Dyninst, ByteWeight, BAP, IDA</td>
</tr>
<tr>
<td>Shin:RNN</td>
<td>[Shin et al., 2015]</td>
<td>N</td>
<td>2015</td>
<td>GNU Unix</td>
<td>ELF</td>
<td>x86_64</td>
<td>clang/gcc</td>
<td>0-0-3</td>
<td>2,200</td>
<td>ByteWeight</td>
</tr>
<tr>
<td>FID</td>
<td>[Wang et al., 2017]</td>
<td>N</td>
<td>2017</td>
<td>GNU coreutils</td>
<td>ELF</td>
<td>x86_64</td>
<td>clang/gcc</td>
<td>0-0-3</td>
<td>4,240</td>
<td>IDA, ByteWeight</td>
</tr>
<tr>
<td>LEMMA*</td>
<td>[Guo et al., 2018]</td>
<td>Y</td>
<td>2018</td>
<td>GNU Unix</td>
<td>ELF</td>
<td>x64</td>
<td>gcc</td>
<td>0-0-3</td>
<td>2,200</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 4: 10 Groups for 10-fold cross validation for ByteWeight.

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

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.

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

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%. LEMMA 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, LEMMA 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, which 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-01. This function takes a single argument (i.e., p_sess), which plays a role in branching out into multiple call invocations depending

*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.

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

Listing 2: Example of a function and its disassembly after optimization. The function cmd_process_pasmv_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-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). 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.

```
MagickExport ImageInfo *AcquireImageInfo(void) {
  ImageInfo *image_info;
  image_info=(ImageInfo *) AcquireMagickMemory(sizeof(*image_info));
  if (image_info == (ImageInfo *) NULL)
    ThrowFatalException(ResourceLimitFatalError,"MemoryAllocationFailed");
  GetImageInfo(image_info);
  return(image_info);
}
```

Listing 3: Example of a non-continuous 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_pasmv_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 atoll(const char *__nptr)
0x9C8A20 xor esi, esi
0x9C8A22 mov edx, 0Ah
0x9C8A27 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.
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 along). 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 recap’s 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,

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

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.

<table>
<thead>
<tr>
<th>Tool</th>
<th>P</th>
<th>T</th>
<th>F</th>
<th>R</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ghidra</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>clang</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gcc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ByteWeight</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shin:RNN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IDAPro</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nucleus</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

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


