ByteWeight: Learning to Recognize Functions in Binary Code

Tiffany Bao
Carnegie Mellon University
`tiffanybao@cmu.edu`

Jonathan Burket
Carnegie Mellon University
`jburket@cmu.edu`

Maverick Woo
Carnegie Mellon University
`pooh@cmu.edu`

Rafael Turner
University of Chicago
`turnersr@uchicago.edu`

David Brumley
Carnegie Mellon University
`dbrumley@cmu.edu`

Abstract

Function identification is a fundamental challenge in reverse engineering and binary program analysis. For instance, binary rewriting and control flow integrity rely on accurate function detection and identification in binaries. Although many binary program analyses assume functions can be identified a priori, identifying functions in stripped binaries remains a challenge.

In this paper, we propose ByteWeight, a new automatic function identification algorithm. Our approach automatically learns key features for recognizing functions and can therefore easily be adapted to different platforms, new compilers, and new optimizations. We evaluated our tool against three well-known tools that feature function identification: IDA, BAP, and Dyninst. Our data set consists of 2,200 binaries created with three different compilers, with four different optimization levels, and across two different operating systems. In our experiments with 2,200 binaries, we found that ByteWeight missed 44,621 functions in comparison with the 266,672 functions missed by the industry-leading tool IDA. Furthermore, while IDA misidentified 459,247 functions, ByteWeight misidentified only 43,992 functions.

1 Introduction

Binary analysis is an essential security capability with extensive applications, including protecting binaries with control flow integrity (CFI) [1], extracting binary code sequences from malware [9], and hot patching vulnerabilities [25]. Research interest in binary analysis shows no sign of waning. In 2013 alone, several papers such as CFI for COTS [34] (referred to as COTS-CFI in this paper), the Rendezvous search engine for binaries [21], and the Phoenix decompiler [28] focus on developing new binary analysis techniques.

Function identification is a preliminary and necessary step in many binary analysis techniques and applications. For example, one property of CFI is to constrain inter-function control flow to valid paths. In order to reason about such paths, however, binary-only CFI infrastructures need to be able to identify functions accurately. In particular, COTS-CFI [34], CCFIR [33], MoCFI [12], Abadi et al. [1], and extensions like XFI [15] all depend on accurate function identification to be effective.

CFI is not the only consumer of binary-level function identification techniques. For example, Rendezvous [21] is a search engine that operates at the granularity of function binaries; incorrect function identification can therefore result in incomplete or even incorrect search results. Decompilers such as Phoenix [28], Boomerang [32], and Hex-Rays [18] recover high-level source code from binary code. Naturally, decompilation occurs on only those functions that have been identified in the input binary.

Given the foundational impact of accurate function identification in so many security applications, is this problem easy and can thus be regarded as "solved"? Interestingly, recent security research papers seem to have conflicting opinions on this issue. On one side, Kruegel et al. argued in 2004 that function start identification can be solved "very well" [23, §4.1] in regular binaries and even some obfuscated ones. On the other side, Perkins et al. described static function start identification as "a complex task in a stripped x86 executable" [25, §2.2.3] and therefore applied a dynamic approach in their ClearView system. A similar opinion is also shared by Zhang et al., who stated that "it is difficult to identify all function boundaries" [34, §3.2] and used a set of heuristics for this task.

So how good are the current tools at identifying functions from stripped, non-malicious binaries? To find out, we collected a dataset of 2,200 Linux and Windows binaries generated by GNU `gcc`, Intel `icc`, and Microsoft Visual Studio (`VS`) with multiple optimization levels. We then use our dataset to evaluate the most recent release of three popular off-the-shelf solutions for function identification: (i) IDA (v6.5 at submission), used in CodeSurfer/x86 [2], Choi et al.'s work on statically determining binary similarity [11], BinDiff [4], and BinNavi [5]; (ii) the CMU Binary Analysis Platform

(BAP, v0.7), used in the Phoenix decompiler [28] and the vulnerability analysis tool Mayhem [10]; and (iii) the `unstrip` utility in Dyninst (dated 2012-11-30), used in BinSlayer [7], Sharif et al.'s work on dynamic malware analysis [29], and Sidiroglou et al.'s work on software recovery navigation [30].

Our finding is that while IDA performs better than BAP and Dyninst on our dataset, its result can still be quite alarming—in our experiment, IDA returned 521,648 true positives (41.81%), 266,672 false negatives (21.38%), and 459,247 false positives (36.81%). While there is no doubt that such failures can have a negative impact on downstream security analyses, a real issue is in setting the right expectation on the subject within the security research community. If there is a publicly-available function identification solution where both its mechanism and limitations are well-understood by researchers, then researchers may be come up with creative strategies to cope with the limitations in their own projects. The goal of this paper is to explain our process of developing such a solution and to establish its quality through evaluating it against the aforementioned solutions.

We draw inspirations from how BAP and Dyninst perform function identification since their source code is available. Both solutions rely on fixed, manually-curated signatures. Dyninst, at the version we tested, uses the byte signature `0x55` (`push %ebp` in assembly) to recognize function starts in ELF x86 binaries [14]. BAP v0.7 uses a more complex signature, but it is also manually generated. Unfortunately, the process of manually generating such signatures do not scale well. For example, each new compiler release may introduce new idioms that require new signatures to capture. The myriad of different optimization settings, such as omit frame pointers, may also demand even more signatures. Clearly, we cannot expect to manually catch up.

One approach to recognizing functions is to automatically learn key features and patterns. For example, seminal work by Rosenblum et al. proposed binary function start identification as a supervised machine learning classification problem [27]. They model function start identification as a Conditional Random Field (CRF) in which binary offsets and a number of selected idioms (patterns) appear in the CRF. Since standard inference methods for CRF on large, highly-connected graphs are expensive, Rosenblum et al. adopted feature selection and approximate inference to speed up their model. However, using hardware available in 2008, they needed 150 compute-days just for the feature selection phase on 1,171 binaries.

In this paper, we propose a new automated analysis for inferring functions and implemented it in our BYTEWEIGHT system. A key aspect of BYTEWEIGHT is the ability to learn signatures for new compilers and optimizations at least one order of magnitude faster than as reported by Rosenblum et al. [27], even after generously accounting for CPU speed increase since 2008. In particular, we avoid using CRFs and feature selection, and instead opt for a simpler model based on learning prefix trees. Our simpler model is scalable using current computing hardware: we finish training 2,064 binaries in under 587 compute-hours. BYTEWEIGHT also does not require compiler information of testing binaries, which makes the tool more powerful in practice. In the interest of open science, we also make our tools and datasets available to seed future improvements.

At a high level, we learn signatures for function starts using a weighted prefix tree, and recognize function starts by matching binary fragments with the signatures. Each node in the tree corresponds to either a byte or an instruction, with the path from the root node to any given node representing a possible sequence of bytes or instructions. The weights, which can be learned with a single linear pass over the data set, express the confidence that a sequence of bytes or instructions corresponds to a function start. After function start identification, we then use value set analysis (VSA) [2] with an incremental control flow recovery algorithm to find function bodies with instructions, and extract function boundaries.

To evaluate our techniques, we perform a large-scale experiment and provide empirical numbers on how well these tools work in practice. Based on 2,200 binaries across operating systems, compilers and optimization options, our results show that BYTEWEIGHT has a precision and recall of 97.30% and 97.44% respectively for function *start* identification. BYTEWEIGHT also has a precision and recall of 92.84% and 92.96% for function *boundary* identification. Our tool is adaptive for varying compilers and therefore more general than current pattern matching methods.

Contributions. This paper makes the following contributions:

- We enumerate the challenges we faced and implement a new function start identification algorithm based on prefix trees. Our approach is automatic and does not require a priori compiler information (see §4). Our approach models the function start identification problem in a novel way that makes it amenable to much faster learning algorithms.
- We evaluate our method on a large test suite across operating systems, compilers, and compiling optimizations. Our model achieves better accuracy than previously available tools.
- We make our test infrastructure, data set, implementation, and results public in an effort to promote open science (see §5).

```c
#include <stdio.h>
#include <string.h>
#define MAX 10
void sum(char *a, char *b)
{
    printf("%s + %s = %d\n",
           a, b, atoi(a) + atoi(b));
}
void sub(char *a, char *b)
{
    printf("%s - %s = %d\n",
           a, b, atoi(a) - atoi(b));
}
void assign(char *a, char *b)
{
    char pre_b[MAX];
    strcpy(pre_b, b);
    strcpy(b, a);
    printf("b is changed from %s to %s\n",
           pre_b, b);
}
int main(int argc, char **argv)
{
    void (*funcs[3])(char *x, char *y);
    int f;
    char a[MAX], b[MAX];
    funcs[0] = sum;
    funcs[1] = sub;
    funcs[2] = assign;
    scanf("%d %s %s", &f, a, b);
    (*funcs[f])(a, b);
    return 0;
}
```

(a) Source code

```
        00400660 <assign>:
        mov     %rbx,-0x10(%rsp)
        mov     %rbp,-0x8(%rsp)
        sub     $0x28,%rsp
        mov     %rdi,%rbp
        lea     0xf(%rsp),%rdi
        ...
        004006b0 <sub>:
        mov     %rbx,-0x18(%rsp)
        mov     %rbp,-0x10(%rsp)
        mov     %rsi,%rbx
        mov     %r12,-0x8(%rsp)
        xor     %eax,%eax
        sub     $0x18,%rsp
        ...
        00400710 <sum>:
        mov     %rbx,-0x18(%rsp)
        mov     %rbp,-0x10(%rsp)
        mov     %rsi,%rbx
        mov     %r12,-0x8(%rsp)
        xor     %eax,%eax
        sub     $0x18,%rsp
        ...
```

(b) Assembly compiled by `gcc -O3`

Figure 1: Example C Code. IDA fails to identify functions `sum`, `sub`, and `assign` in the compiled binary.

2 Running Example

We start with a simple example written in C, shown in Figure 1. In this program, three functions are stored as function pointers in the array `funcs`. When the program is run, input from the user dictates which function gets called, as well as the function arguments. We compiled this example code on Linux Debian 7.2 x86-64 using `gcc` with `-O3`, and stripped the binary using the command `strip`. We then used IDA to disassemble the binary and perform function identification. Many security tools use IDA in this way as a first step before performing additional analysis [9, 20, 24]. Unfortunately, for our example program IDA failed to identify the functions `sum`, `sub`, and `assign`.

IDA's failure to identify these three critical functions has significant implications for security analyses that rely on accurate function boundary identification. Recall that the CFI security policy dictates that runtime execution must follow a path of the static control flow graph (CFG). In this case, when the CFG is recovered by first identifying functions using IDA, any call to `sum`, `sub`, or `assign` would be incorrectly disallowed, breaking legitimate program behavior. Indeed, any indirect jump to an unidentified or mis-identified function will be blocked by CFI. The greater the number of functions missed, the more legitimate software functionality incorrectly lost. Secondly, suppose we are checking code for potential security-critical bugs. In our sample program, the function `assign` is vulnerable to a buffer overflow attack, but is not identified by IDA as a function. For tools like ClearView [25] that operate on binaries at the function level, missing functions can mean missing vulnerabilities.

In our analysis of 1,171 binaries, we observed that that IDA failed to identify 266,672 functions. BYTEWEIGHT improves on this number, missing only 44,621. BYTEWEIGHT also makes fewer mistakes, incorrectly identifying functions 43,992 times compared to 459,247 with IDA. While these results are not perfect, they demonstrate that our automated machine learning approach can outperform years of manual hand-tuning that has gone into IDA.

3 Problem Definition and Challenges

The goal of function identification is to faithfully determine the set of functions that exist in binary code. Determining what functions exist and which bytes belong to which functions is trivial if debug information is present.

For example, "unstripped" Linux binaries contain a symbol table that maps function names to locations in a binary, and Microsoft program database (PDB) information contains similar information for Windows binaries. We start with notation to make our problem definition precise and then formally define three function identification problems. We then describe several challenges to any approach or algorithm that addresses the function identification problems. In subsequent sections we provide our approach.

3.1 Notation and Definitions

A binary program is divided into a number of sections. Each section is given a type, such as code, data, read-only data, and so on. In this paper we only consider executable code, which we treat as a binary string.

Let B denote a binary string. For concreteness, think of this as a binary string from the `.text` section in a Linux executable. Let $B[i]$ denote the i^{th} byte of a binary string, and $B[i:i+j]$ refer to the list of contiguous bytes $B[i], B[i+1], \ldots, B[i+j-1]$. Thus, $B[i:i+j]$ is j-bytes long (with $j \geq 0$).

Each byte in an executable is associated with an *address*. The address of byte i is calculated with respect to a fixed section offset, i.e., if the section offset is ω, the address of byte i is $i+\omega$. For convenience, we omit the offset, and refer to i as the i^{th} address. Since the real address can always be calculated by adding the fixed offset, this can be done without loss of generality.

A function F_i in a binary B is a list of addresses corresponding to statements in either a function from the original compiled language or a function introduced directly by the compiler, denoted as

$$F = \{B[i], B[j], \ldots, B[k]\}$$

Note that function bytes need not be a set of contiguous addresses. We elaborate in §3.3 on real optimizations that result in high-level functions being compiled to a set of non-contiguous intervals of instructions.

Towards our goal of determining which bytes of a binary belong to which functions, we define the *set of functions* in a binary

$$\mathbf{FUNCS}(B) = \{F_1, F_2, \ldots, F_k\}.$$

Note that functions may share bytes, i.e., it may be that $F_1 \cap F_2 \neq \emptyset$. We give examples in §3.3 where this is the case.

We call the lowest address of a function F_i the *function start* address s_i, i.e., $s_i = \min(F_i)$. The *function end* address e_i is the maximum byte in a function body, i.e., $e_i = \max(F_i)$. We define the *function boundary* (s_i, e_i) as the function start and end addresses for F_i.

In order to evaluate function identification algorithms, we define ground truth in terms of oracles, which may have a number of implementations:

Function Oracle. $\mathbf{O}_{\text{func}}$ is an oracle that, given a binary B, returns a list of functions $\mathbf{FUNCS}(B)$ where each F_i is a set of bytes representing higher-level function i, as defined above.

Boundary Oracle. $\mathbf{O}_{\text{bound}}$ is an oracle that, given B, returns the set of function boundaries $\{(s_1,e_1),(s_2,e_2),\ldots,(s_k,e_k)\}$.

Start Oracle. $\mathbf{O}_{\text{start}}$ is an oracle that, given B, returns the set of function start addresses $\{s_1, s_2, \ldots, s_k\}$.

These oracles are successively less powerful. For example, implementing a boundary oracle $\mathbf{O}_{\text{bound}}$ from a function oracle $\mathbf{O}_{\text{func}}$ requires simply taking the minimum and maximum element of each F_i. Similarly, a start oracle $\mathbf{O}_{\text{start}}$ can be implemented from either $\mathbf{O}_{\text{func}}$ or $\mathbf{O}_{\text{bound}}$ by finding the minimum element of each F_i.

We do not restrict ourselves to a specific oracle implementation, as realizable oracles may vary across operating system and compiler. For example, the boundary oracle can be implemented by retaining debug information for Windows or Linux binaries. The function oracle can be implemented by instrumenting a compiler to output a list of instruction addresses included in each compiled function.

3.2 Problem Definition

With the above definitions, we are now ready to state our problem definitions. We start with the least powerful identification (function start) and build up to the most difficult one (entire function).

Definition 3.1. The *Function Start Identification* (FSI) problem is to output the complete list of function starts $\{s_1, s_2, \ldots, s_k\}$ given a binary B compiled from a source with k functions.

Suppose there is an algorithm $\mathscr{A}_{\mathsf{FSI}}(B)$ for the FSI problem which outputs $S = \{s_1, s_2, \ldots, s_k\}$. Then:

- The set of true positives, TP, is $S \cap \mathbf{O}_{\text{start}}(B)$.
- The set of false positives, FP, is $S - \mathbf{O}_{\text{start}}(B)$.
- The set of false negatives, FN, is $\mathbf{O}_{\text{start}}(B) - S$.

We also define precision and recall. Roughly speaking, precision reflects the number of times an identified function start is really a function start. A high precision means that most identified functions are indeed functions, whereas a low precision means that some sequences are incorrectly identified as functions. Recall is the measurement describing how many functions were identified within a binary. A high recall means an algorithm detected most functions, whereas a low recall means most functions were missed. Mathematically, they can be expressed as

$$\text{Precision} = \frac{|\text{TP}|}{|\text{TP}| + |\text{FP}|}$$

and

$$\text{Recall} = \frac{|\text{TP}|}{|\text{TP}| + |\text{FN}|}.$$

A more difficult problem is to identify both the start *and* end addresses for a function:

Definition 3.2. The *Function Boundary Identification* (FBI) problem is to identify the start and end bytes (s_i, e_i) for each function i in a binary, i.e., $S = \{(s_1,e_1),(s_2,e_2),\ldots,(s_k,e_k)\}$, given a binary B compiled from a source with k identified functions.

Suppose there is an algorithm $\mathscr{A}_{\mathsf{FBI}}(B)$ for the FBI problem which outputs $S = \{(s_1,e_1),(s_2,e_2),\ldots,(s_k,e_k)\}$. We then define true positives, false positives, and false negatives similarly to above with the additional requirement that *both* the start and end addresses must match the output of the boundary oracle, i.e., for oracle output (s_{gt}, e_{gt}) and algorithm output $(s_{\mathscr{A}}, e_{\mathscr{A}})$, a positive match requires $s_{gt} = s_{\mathscr{A}}$ *and* $e_{gt} = e_{\mathscr{A}}$. A false negative occurs if either the start or end address is wrong. Precision and recall are defined analogously to the FSI problem.

Finally, we define the general function identification problem:

Definition 3.3. The *Function Identification* (FI) problem is to output a set $\{F_1, F_2, \ldots, F_k\}$ where each F_i is a list of bytes corresponding to high-level function i given a binary B with k identified functions.

We define true positives, false positives, false negatives, precision, and recall for the FI problem in the same ways as FSI and FBI but add the requirement that all bytes of a function must be matched between agorithm and oracle.

The above problem definitions form a natural hierarchy, where function start identification is the easiest and full function identification is the most difficult. For example, an algorithm $\mathscr{A}_{\mathsf{FBI}}$ for function boundaries can solve the function start problem by returning the start element of each tuple. Similarly, an algorithm for the function identification problem needs only return the maximum and minimum element to solve the function boundary identification problem.

3.3 Challenges

Identifying functions in binary code is made difficult by optimizing compilers, which can manipulate functions in unexpected ways. In this section we highlight several challenges posed by the behavior of optimizing compilers.

Not every byte belongs to a function. Compilers may introduce extra instructions for alignment and padding between or within a function. This means that not every instruction or byte must belong to a function. For example, suppose we have symbol table information for a binary B. One naive algorithm is to first sort symbol-table entries by address, and then ascribe each byte between entry f_i and f_{i+1} as belonging to function f_i. This algorithm has appeared in several binary analysis platforms used in security research, such as versions of BAP [3] and BitBlaze [6]. This heuristic is flawed, however. For example, in Figure 2 lines 7–8 are not owned by any function.

```
1    <_func1>:
2    100000e20:   push   %rbp
3    100000e21:   mov    %rsp,%rbp
4    100000e24:   lea    0x69(%rip),%rdi
5    100000e2b:   pop    %rbp
6    100000e2c:   jmpq   100000e5e <_puts$stub>
7    100000e31:   nopl   0x0(%rax)
8    100000e38:   nopl   0x0(%rax,%rax,1)
9    <func2>:
```

Figure 2: Unreachable function example: source code and assembly.

Functions may be non-contiguous. Functions may have gaps. The gaps can be jump tables, data, or even instructions for completely different functions [26]. As noted by Harris and Miller [19], function sharing code can also lead to non-contiguous functions. Figure 3 shows code that starts out with the function `ConvertDefaultLocale`. Midway through the function at lines 17–21, however, the compiler decided to include a few lines of code for `FindNextFileW` as an optimization. Many binary analysis platforms, such as BAP [3] and BitBlaze [6], are not able to handle non-contiguous functions.

Functions may not be reachable. A function may be dead code and never called, but nonetheless appear in the binary. Recognizing such functions is still important in many security scenarios. For example, suppose two malware samples both contain a unique, identifying, yet uncalled function. Then the two malware samples are likely related even though the function is never called. One consequence of this is that techniques based solely on recursive disassembling from program start are not well-suited to solve the function identification problem. A recursive disassembler only disassembles bytes that occur along some control flow path, and thus by definition will miss functions that are not called.

Unreachability may occur for several reasons, including compiler optimizations. For example, Figure 4 shows a function for computing factorials called `fac`. When compiled by `gcc -O3`, the result of the call to `fac` is precomputed and inlined. Although the code of `fac` appears, it is never called in the binary code.

Security policies such as CFI and XFI must be aware of all low-level functions, not just those in the original code.

```
1    <ConvertDefaultLocale>
2      7c8383ff:   mov    %edi,%edi
3      7c838401:   push   %ebp
4      ...
5      7c83840c:   jz     7c848556
6      7c838412:   test   %eax, %eax
7      7c838414:   jz     7c83965c
8      7c83841a:   mov    $1024,%ecx
9      7c83841f:   cmp    %ecx,%eax
10     7c838421:   jz     7c83965c
11     7c838427:   test   $252,%ah
12     7c83842a:   jnz    7c838442
13     7c83842c:   mov    %eax,%edx
14     ...
15     7c838442:   pop    %ebp
16     7c838443:   ret    4
17     ; chunk of different function FindNextFileW
18     7c838446:   push   6
19     7c838448:   call   sub_7c80935e
20     7c83844d:
21     ; end of chunk
22     ...
23     7c83965c:   call   GetUserDefaultLCID
24     7c890661:   jmp    7c838442
25     ...
26     7c848556:   mov    $8,%eax
27     7c84855b:   jmp    7c838442
```

Figure 3: Lines 17–21 show code from `FindNextFileW` included in the middle of `ConvertDefaultLocale`.

Functions may have multiple entries. High-level languages use functions as an abstraction with a single entry. When compiled, however, functions may have multiple entries as a result of specialization. For example, the `icc` compiler with `-O1` specialized the `chown_failure_ok` function in GNU LIBC. As shown in Figure 5, a new function entry `chown_failure_ok.` (note the period) is added for use when invoking `chown_failure_ok` with `NULL`. The compiled binary has both symbol table entries. Unlike shared code for two functions that were originally separate, the compiler here has introduced shared code via multiple entries as an optimization.

Identifying both functions is necessary in many security scenarios, e.g., CFI needs to identify each function entry point for safety, and realize that both are possible targets. More generally, any binary rewriting for protection (e.g., memory safety, control safety, etc.) would need to reason about both entry points.

Functions may be removed. Functions can be removed by function inlining, especially small functions. Compilers perform function-inlining to reduce function call overhead and expose more optimization opportunities. For example, the function `utimens_symlink` is inlined into the function `copy_internal` when compiled by `gcc` with `-O2`. The source code and assembly code are shown in Figure 6. Note that function inlining does not have to be explicitly declared with `inline` annotation in source code. Many compilers inline functions by default unless explicitly disabled with options such as `-fno-deault-inline` [17]. This indicates that for those binary analysis techniques which need function information, even though source code is accessible, a robust function identification technique should still operate on the program binary. If using source code, function identification may be less precise due to functions that are inlined during compilation.

Each compilation is different. Binary code is not only heavily influenced by the compiler but also the compiler version and specific optimizations employed. For example, `icc` does not pre-compute the result of `fac` in Figure 4, but `gcc` does. Even different versions of a compiler may change code. For example, traditionally `gcc` (e.g., version 3) would only omit the use of the frame pointer register `%ebp` when given the `-fomit-frame-pointer` option. Recent versions of `gcc` (such as version 4.2), however, opportunistically omit the frame pointer when compiled with `-O1` and `-O2`. As a result several tools that identified functions by scanning for `push %ebp` break. For example, Dyninst, used for instrumentation in several security projects, relies on this heuristic to identify functions and breaks on recent versions of `gcc`.

4 BYTEWEIGHT

In this section, we detail the design and algorithms used by BYTEWEIGHT to solve the function identification problems. We first start with the FSI problem, and then move to the more general function identification problem.

We cast FSI as a machine learning classification problem where the goal is to label each byte of a binary as either a function start or not. We use machine learning to automatically generate literal patterns so that BYTEWEIGHT can handle new compilers and new optimizations without relying on manually generated patterns or heuristics. Our algorithm works with both byte sequences and disassembled instruction sequences.

Our overall system is shown in Figure 7. Like any classification problem, we have a training phase followed by a classification phase. During training, we first compile a reference corpus of source code to produce binaries where the start addresses are known. At a high level, our algorithm creates a weighted prefix tree of known function start byte or instruction sequences. We *weight* vertices in the prefix tree by computing the ratio of true positives to the sum of true and false positives for each sequence in the reference data set. We have designed and implemented two variations of BYTEWEIGHT: one working with raw bytes and one with normalized disassembled instructions. Both use the same overall algorithm and data structures. We show in our evaluation that the normalization approach provides higher precision and recall, and costs less time (experiment 5.2).

In the classification phase, we use the weighted prefix tree to determine whether a given sequence of bytes or

```
int fac(int x)
{
  if (x == 1) return 1;
  else return x * fac(x - 1);
}

void main(int argc, char **argv)
{
  printf("%d", fac(10));
}
```

(a) Source code

```
080483f0 <fac>:
...
08048410 <main>:
...
movl   $0x375f00,0x4(%esp)
movl   $0x8048510,(%esp)
call   8048300  ;call printf without fac
xor    %eax,%eax
add    $0x8,%esp
pop    %ebp
ret
```

(b) Assembly compiled by `gcc -O2`

Figure 4: Unreachable code: source code and assembly.

```
extern bool
chown_failure_ok (struct cp_options const *x)
{
  return ((errno == EPERM || errno == EINVAL)
          && !x->chown_privileges);
}
```

(a) Source Code

```
<chown_failure_ok>:
804f544:        mov    0x4(%esp),%eax
<chown_failure_ok.>:
804f548:        push   %esi
804f549:        push   %esi
804f54a:        push   %esi
...
```

(b) Assembly compiled by `icc -O1`

Figure 5: `chown_failure_ok` is specialized: source code and assembly.

instructions corresponds to a function start. We say that a sequence corresponds to a function start if the corresponding *terminal node* in the prefix tree has a weight value larger than the threshold t. In the case where the sequence exactly matches a path in the prefix tree, the terminal node is the final node in this path. If the sequence does not exactly match a path in the tree, the terminal node is the last matched node in the sequence.

Once we identify function starts, we infer the remaining bytes (and instructions) that belong to a function using a CFG recovery algorithm. The algorithm incrementally determines the CFG using a variant of VSA [2]. If an indirect jump depends on the value of a register, then we over-approximate a solution to the function identification problem by adding edges that correspond to locations approximated using VSA.

4.1 Learning Phase

The input to the learning phase is a corpus of training binaries $\mathbb{T}$, and a maximum sequence length $\ell > 0$. ℓ serves as a bound on the maximum tree height.

In ByteWeight, we first generate the oracle $\mathbf{O}_{\text{bound}}$ by compiling known source using a variety of optimization levels while retaining debug information. The debug information gives us the required (s_i, e_i) pair for each function i in the binary.

In this paper, we consider two possibilities: learning over raw bytes and learning over normalized instructions. We refer to both raw bytes and instructions as a sequence of elements. The sequence length ℓ determines how many raw sequential bytes or instructions we consider for training.

Step 1: Extract first ℓ elements for each function (Extraction). In the first step, we iterate over all (s_i, e_i) pairs and extract the first ℓ elements. If there are fewer than ℓ elements in the function, we extract the maximum number of elements. For raw bytes, this is $B[s : s+\ell]$ bytes, and for instructions, it is the first ℓ instructions disassembled linearly starting from $B[s]$.

Step 2: Generate a prefix tree (Tree Generation). In step 2, we generate a *prefix tree* from the extracted sequences to represent all possible function start sequences up to ℓ elements.

A prefix tree, also called a trie, is a data structure enabling efficient information retrieval. In the tree, each non-root node has an associated byte or instruction. The sequence for a node n is represented by the elements that appear on the path from the root to n. Note that the tree represents all strings up to ℓ elements, not just exactly ℓ elements.

Figure 8a shows an example tree on instructions, where node `callq 0x43a28` represents the instruction sequence:

```
push    %ebp            ;saved stack pointer
mov     %esp,%ebp       ;establish new frame
callq   0x43a28         ;call another function
```

If the sequence is over bytes, the prefix tree is calculated directly, although our experiments indicate that a prefix tree calculated over normalized instructions fairs

```
static inline int
utimens_symlink (char const *file,
                 struct timespec const *timespec)
{
  int err = lutimens (file, timespec);
  if (err && errno == ENOSYS)
    err = 0;
  return err;
}

static bool
copy_internal (char const *src_name,
               char const *dst_name,
               ...)
{
  ...
  if ((dest_is_symlink
      ?utimens_symlink (dst_name,
                        timespec)
      :utimens (dst_name, timespec))
      != 0)
  ...
}
```

(a) Source Code

```
<_copy_internal>:
100003170:      push   %rbp
100003171:      mov    %rsp,%rbp
100003174:      push   %r15
100003176:      push   %r14
...
10000468c:      test   %r14b,%r14b
10000468f:      je     100005bfd
100004695:      lea    -0x738(%rbp),%rsi
10000469c:      mov    -0x750(%rbp),%rdi
1000046a3:      callq  10000d020 <_lutimens>
1000046a8:      test   %eax,%eax
1000046aa:      mov    %eax,%ebx
...
```

(b) Assembly compiled by `gcc -O2`

Figure 6: Example of function being removed due to function inlining optimization.

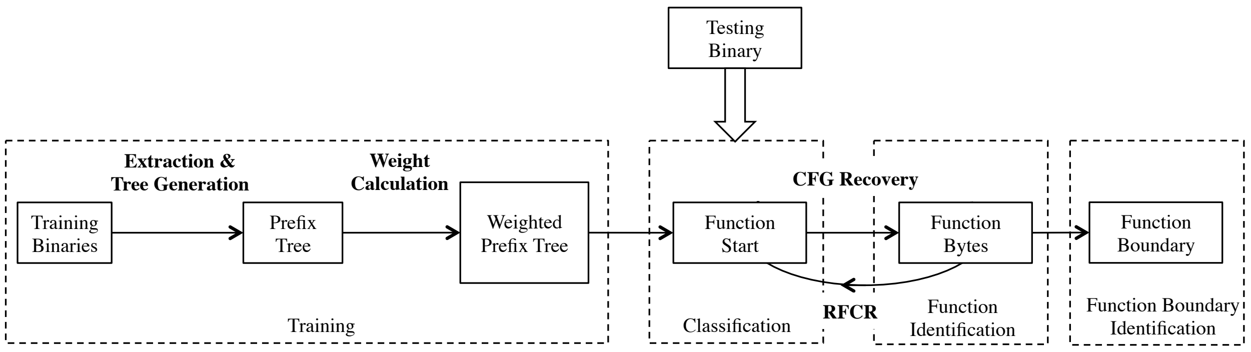


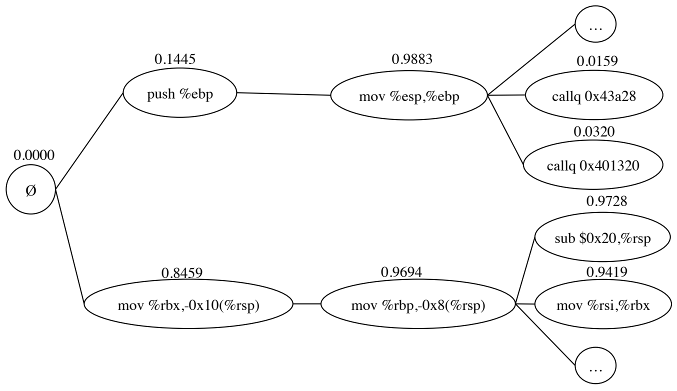
Figure 7: The BYTEWEIGHT function boundary inference approach.

better. We perform two types of normalization: immediate number normalization and call & jump instruction normalization. As shown in Table 1, normalization takes an instruction as input and generalizes it so that it can match against very similar, but not identical instructions. These two types of normalization help us improve recall at the cost of a little precision (Table 2). In our running example, only the function `assign` is recognized as a function start when matched against the unnormalized prefix tree (Figure 8a), while functions `assign`, `sub`, and `sum` can all be recognized when matched against the normalized prefix tree (Figure 8b).

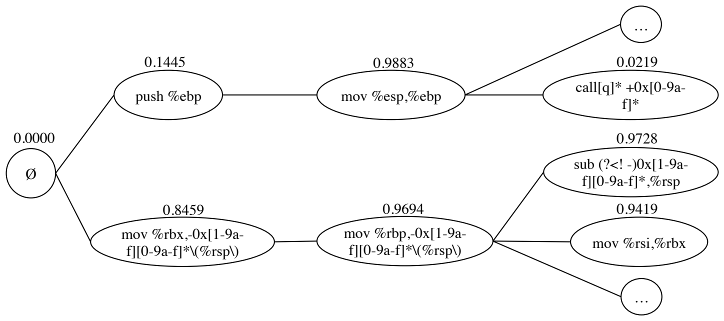
Step 3: Calculate tree weights (Weight Calculation). The prefix tree represents possible function start sequences up to ℓ elements. For each node, we assign a weight that represents the likelihood that the sequence corresponding to the path from the root node to this node is a function start in the training set. For example, according to Figure 8, the weight of node `push %ebp` is 0.1445, which means that during training, 14.45% of all sequences with prefix of `push %ebp` were truly function starts, while 85.55% were not.

To calculate the weight, we first count the number of occurrences $\mathbb{T}_+$ in which each prefix in the tree matches a true function start with respect to the ground truth $\mathbf{O}_{\text{start}}$ for the entire training set $\mathbb{T}$.

Second, we lower the weight of a prefix if it occurs in a binary, but is not a function start. We do this by performing an exhaustive disassembly starting from every address that is *not* a function start [23]. We match each exhaustive disassembly sequence of ℓ elements against the tree. We call these *false matches*. The number of false matches $\mathbb{T}_-$ is the number of times a prefix represented



(a) Unnormalized


0.0000
Ø
0.1445
push %ebp
0.9883
mov %esp,%ebp
…
0.0219
call[q]* +0x[0-9a-f]*
0.8459
mov %rbx,-0x[1-9a-f][0-9a-f]*\(%rsp\)
0.9694
mov %rbp,-0x[1-9a-f][0-9a-f]*\(%rsp\)
0.9728
sub (?<! -)0x[1-9a-f][0-9a-f]*,%rsp
0.9419
mov %rsi,%rbx
…

(b) Normalized

Figure 8: Example of unnormalized (a) and normalized (b) prefix tree. Weight is shown above its corresponding node.

in the tree is not a function start in the training set $\mathbb{T}$. The weight for each node n is then the ratio of true positives to overall matches

$$W_n = \frac{\mathbb{T}_+}{\mathbb{T}_+ + \mathbb{T}_-}. \tag{1}$$

Since the prefix tree can end up being quite large, it is beneficial to prune the tree of unnecessary nodes. For each node in the tree, we remove all its child nodes if the value of $\mathbb{T}_-$ for this node is 0. For any child node, the value of $\mathbb{T}_-$ is never negative and never larger than the value of $\mathbb{T}_-$ for the parent node. Hence, if $\mathbb{T}_-$ is 0 for a parent node, then the value must be 0 for all of the child nodes as well. The intuition here is that if a child node matches a sequence that is not a function start, then so must the parent node. Thus, if the parent node does not have any false matches, then neither can a child node. Based on Equation 1, if $\mathbb{T}_- = 0$ and $\mathbb{T}_+ > 0$, then the weight of the node is 1. Since the child nodes of such a node also have a $\mathbb{T}_-$ value of 0 and are not included in the tree if $\mathbb{T}_+ = 0$, they must also have a weight of 1. As discussed more in Section 4.2, child nodes with identical weights are redundant and can safely be removed without affecting classification.

This pruning optimization helps us greatly reduce the space needed by the tree. For example, pruning reduced the number of nodes in the prefix tree from 2,483 to 1,447 for our Windows x86 dataset. Moreover, pruning increases the speed of matching, since we can determine the weight of test sequences after traversing fewer nodes in the tree.

4.2 Classification Phase Using a Weighted Prefix Tree

The output of the learning phase is a weighted prefix tree (e.g., Figure 8). The input to the classification step is a binary B, the weighted prefix tree, and a weight threshold t.

To classify instructions, we perform exhaustive disassembly of the input binary B and match against the tree. Matching is done by tokenizing the disassembled stream, performing normalization as done during learning, and walking the tree. To classify bytes rather than instructions, we again start at every offset but instead match the raw bytes instead of normalized instructions.

The weight of a sequence is determined by last matching node (the *terminal* node) during the walk. For example, given the tree in Figure 8a, and our running example with sequences

```
mov    %rbx,-0x10(%rsp)
mov    %rbp,-0x8(%rsp)
sub    %0x28,%rsp
```

Type		Unnormalized Signature	Normalized Signature
Immediate	all	`mov $0xaa,%eax`	`mov \$-*0x[0-9a-f]+,%eax`
		`mov %gs:0x0,%eax`	`mov %gs:-*0x[0-9a-f]+,%eax`
		`mov 0x80502c0,%eax`	`mov -*0x[0-9a-f]+,%eax`
	zero	`mov $0xaa,%eax`	`mov \$-*0x[1-9a-f][0-9a-f]*,%eax`
		`mov %gs:0x0,%eax`	`mov %gs:0x0+,%eax`
		`mov 0x80502c0,%eax`	`mov -*0x[1-9a-f][0-9a-f]*,%eax`
	positive	`mov $0xaa,%eax`	`mov \$(?<! -)0x[1-9a-f][0-9a-f]*,%eax`
		`mov %gs:0x0,%eax`	`mov %gs:-0x[0-9a-f]+|0x0+,%eax`
		`mov 0x80502c0,%eax`	`mov (?<! -)0x[1-9a-f][0-9a-f]*,%eax`
	negative	`mov $0xaa,%eax`	`mov \$(?<! -)0x[0-9a-f]+,%eax`
		`mov %gs:0x0,%eax`	`mov %gs:(?<! -)0x[0-9a-f]+,%eax`
		`mov 0x80502c0,%eax`	`mov (?<! -)0x[0-9a-f]+,%eax`
		`movzwl -0x6c(%ebp),%eax`	`movzl -0x[1-9a-f][0-9a-f]*\(%ebp\),%eax`
	npz	`mov $0xaa,%eax`	`mov \$(?<! -)0x[1-9a-f][0-9a-f]*,%eax`
		`mov %gs:0x0,%eax`	`mov %gs:0x0+,%eax`
		`mov 0x80502c0,%eax`	`mov (?<! -)0x[1-9a-f][0-9a-f]*,%eax`
		`movzwl -0x6c(%ebp),%eax`	`movzl -0x[1-9a-f][0-9a-f]*\(%ebp\),%eax`
Call & Jump		`call 0x804cf32`	`call[q]* +0x[0-9a-f]*`

For immediate normalization, we generalize immediate operands. There are five kinds of generalization: all, zero, positive, negative, and npz. For jump and call instruction normalization, we generalize callee and jump addresses.

Table 1: Normalizations in signature.

the matching node will be `mov%rbp,-0x8(%rsp)`, giving a weight of 0.9694. However, for another sequence

```
push   %ebp
and    $0x2,%esp
```

we would have weight 0.1445. We say the sequence is the beginning of a function if the output weight w is not less than the threshold t.

4.3 The Function Identification Problem

At a high level, we address the function identification problem by first determining the start addresses for functions, and then performing static analysis to recover the CFG of instructions that are reachable from the start. Direct control transfers (e.g., direct jumps and calls) are followed using recursive disassembly. Indirect control transfers, e.g., from indirect calls or jump tables, are enumerated using VSA [2]. The final CFG then represents all instructions (and corresponding bytes) that are owned by the function starting at the given address.

CFG recovery starts at a given address and recursively finds new nodes that are connected to found nodes. The process ends when no more vertices are added into graph. Starting at the addresses classified for FSI, CFG recovery recursively adds instructions that are reachable from these starts. A first-in-first-out vertex array is maintained during CFG recovery.

At the beginning, there is only one element – the start address in the array. In each round, we process the first element by exploring new reachable instructions. If the new instruction is not in the array, it will be appended to the end. Elements in the array are handled accordingly until all elements have been processed and no more instructions are added.

If the instruction being processed is a branch mnemonic, the reachable instruction is the branch reference. If it is a call mnemonic, the reachable instructions include both the call reference and the instruction directly following the call instruction. If it is an exit instruction, there will be no new instruction. For the rest of mnemonics, the new instruction is the next one by address. We handle indirect control transfer instruction by VSA: we infer a set that over-approximates the destination of the indirect jump and thus over-approximate the function identification problem.

Note that functions can exit by calling a no-return function such as `exit`. This means that some call instructions in fact never return. To detect these instances, we check the call reference to see if it represents a known no-return function such as `abort` or `exit`.

4.4 Recursive Function Call Resolution

Pattern matching can miss functions; for example, a function that is written directly in assembly may not obey calling conventions. To catch these kinds of missed functions, we continue to supplement the function start list during CFG recovery. If a call instruction has its callee in the `.text` section, we consider the callee to be a function start. We then do CFG recovery again, starting at the new function start until there are no more functions added into the function start list. We will refer to this strategy as recursive function call resolution (RFCR). In §5.3, we discuss the effectiveness of this technique in function start identification.

4.5 Addressing Challenges

In this section, we describe how ByteWeight addresses the challenges raised in §3.3.

First, ByteWeight recovers functions that are unreachable via calls because it does not depend on calls to identify functions. In particular, ByteWeight recovers any function start that matches the learned weighted prefix tree as described above. Similarly, our approach will also learn functions that have multiple entries, provided a similar specialization occurs in the training set. This seems realistic in many scenarios since the number of compiler optimizations that create multiple entry functions are relatively few and can be enumerated during training.

ByteWeight also deals with overlapping byte or instruction sequences provided that there is a unique start address. Consider two functions that start at different addresses, but contain the same bytes. During CFG recovery, ByteWeight will discover that both functions use the same bytes, and attribute the bytes to both functions. ByteWeight can successfully avoid false identification for inlined functions when inlined function does not behave like an empirical function start (does not weighted over threshold in training).

Finally, note that ByteWeight does not need to attribute every byte or instruction to a function. In particular, only bytes (or instructions) that are reachable from the recovered function entries will be owned by a function in the final output.

5 Evaluation

In this section, we discuss our experiments and performance. ByteWeight is a cross-platform tool which can be run on both Linux and Windows. We used BAP [3] to construct CFGs. The rest of the implementation consists of 1988 lines of OCaml code and 222 lines of shell code. We set up ByteWeight on one desktop machine with a quad-core 3.5GHz i7-3770K CPU and 16GB RAM. Our experiments aimed to address three questions:

1. Does ByteWeight's pattern matching model perform better than known models for function start identification? (§5.2)
2. Does ByteWeight perform function start identification better than existing binary analysis tools? (§5.3)
3. Does ByteWeight perform function boundary identification better than existing binary analysis tools? (§5.4)

In this section, we first describe our data set and ground truth (the oracle), then describe the results of our experiments. We performed three experiments answering the above three questions. In each experiment, we compared ByteWeight against existing tools in terms of both accuracy and speed.

Because ByteWeight needs training, we divided the data into training and testing sets. We used standard 10-fold validation, dividing the element set into 10 sub-sets, applying 1 of the 10 on testing, and using the remaining 9 for training. The overall precision and recall represent the average of each test.

5.1 Data Set and Ground Truth

Our data set consisted of 2,200 different binaries compiled with four variables:

Operating System. Our evaluation used both Linux and Windows binaries.

Instruction Set Architecture (ISA). Our binaries consisted of both x86 and x86-64 binaries. One reason for varying the ISA is that the calling convention is different, e.g., parameters are passed by default on the stack in Linux on x86, but in registers on x86-64.

Compiler. We used GNU `gcc`, Intel `icc`, and Microsoft `VS`.

Optimization Level. We experimented with the four optimization levels from no optimization to full optimization.

On Linux, our data set consisted of 2,064 binaries in total. The data set contained programs from `coreutils`, `binutils`, and `findutils` compiled with both `gcc` 4.7.2 and `icc` 14.0.1. On Windows, we used `VS` 2010, `VS` 2012, and `VS` 2013 (depending on the requirements of the program) to compile 68 binaries for x86 and x86-64 each. These binaries came from popular open-source projects: putty, 7zip, vim, libsodium, libetpan, HID API, and pbc (a library for protocol buffers). Note that because Microsoft Symbol Server releases only public symbols which do not contain information of private functions, we were unable to use Microsoft Symbol Server for ground truth and include Windows system applications in our experiment.

We obtained ground truth for function boundaries from the symbol table and PDB file for Linux and Windows binaries, respectively. We used `objdump` to parse symbol tables, and `Dia2dump` [13] to parse PDB files. Additionally, we extracted "thunk" addresses from PDB files. While most tools do not take thunks into account, IDA considers thunks in Windows binaries to be special functions. To get a fair result, we filtered out thunks from IDA's output using the list of thunks extracted from PDB files.

5.2 Signature Matching Model

Our first experiment evaluated the signature matching model for function start identification. We compared BYTEWEIGHT and Rosenblum et al.'s implementation in terms of both accuracy and speed. In order to equally evaluate the signature matching models, recursive function call resolution was not used in this experiment.

The implementation of Rosenblum et al. is available as a matching tool with 12 hard-coded signatures for `gcc` and 41 hard-coded signatures for `icc`. Their learning code was not available, nor was their dataset. Although they evaluated `VS` in their paper, the version of the implementation that we had did not support `VS` and was limited to x86. Each signature has a weight, which is also hard-coded. After calculating the probability for each sequence match, it uses a threshold of 0.5 to filter out function starts. Taking a binary and a compiler name (`gcc` or `icc`), it generates a list of function start addresses. To adapt to their requirements, we divide Linux x86 binaries into two groups by compiler, where each group consists of 516 binaries. We did 10-fold cross validation for BYTEWEIGHT, and use the same threshold as Rosenblum et al.'s implementation.

We also evaluated another two varieties of our model: one without normalization, and one with a maximum tree height of 3, which is same as the model used by Rosenblum et al. and BYTEWEIGHT (3), respectively.

Table 2 shows precision, recall, and runtime for each compiler and each function start identification model. From the table we can see that Rosenblum et al.'s implementation had an accuracy below 70%, while both BYTEWEIGHT-series models achieved an accuracy of more than 85%. Note that BYTEWEIGHT with 10-length and normalized signatures (the last row in table) performed particularly well, with an accuracy of approximately 97%, a more than 35% improvement over Rosenblum et al.'s implementation.

Table 2 also details the accuracy and performance differences among BYTEWEIGHT with different configurations. Comparing against the full configuration model (BYTEWEIGHT), the model with a smaller maximum signature length (BYTEWEIGHT (3)) performs slightly faster (3% improvement), but sacrifices 7% in accuracy. The model without signature normalization (BYTEWEIGHT (no-norm)) has only 1% higher precision but 6.68% lower recall, and the testing time is ten times longer than that of the normalized model.

5.3 Function Start Identification

The second experiment evaluated our full function start identification against existing static analysis tools. We compared BYTEWEIGHT (no-RFCR)—a version without recursive function call resolution, BYTEWEIGHT, and the following tools:

IDA. We used IDA 6.5, build 140116 along with the default FLIRT signatures. All function identification options were enabled.

BAP. We used BAP 0.7, which provides a `get_function` utility that can be invoked directly.

Dyninst. Dyninst offers the tool *unstrip* [31] to identify functions in binaries without debug information.

Naive Method. This matched simple `0x55` (`push %ebp` or `push %rbp`) and `0xc3` (`ret` or `retq`) signatures only.

We divided our data set into four categories: ELF x86, ELF x86-64, PE x86, and PE x86-64. Unlike the previous experiment, binaries from various compilers but the same target were grouped together. Overall, we had 1032 ELF x86 and ELF x86-64 binaries, and 68 PE x86 and PE x86-64 binaries. We evaluated these categories separately, and again applied 10-fold validation. During testing, we used the same score threshold $t = 0.5$ as in the first experiment.

Note that not every tool in our experiment supports all binary targets. For example, Dyninst does not support ELF x86-64, PE x86, or PE x86-64 binaries. We use "-" to indicate when the target is not supported by the tool. Also, we omitted 3 failures in BYTEWEIGHT, and 10 failures in Dyninst during this experiment. Due to a bug in BAP, BYTEWEIGHT failed in 3 `icc` compiled ELF x86-64 binaries: *ranlib* with `-O3`, *ld_new* with `-O2`, and *ld_new* with `-O3`. Dyninst failed in 8 `icc` compiled ELF x86-64 binaries and 2 `gcc` compiled ELF x86-64 binaries. The results of our experiment are shown in Table 3.

As evident in Table 3, BYTEWEIGHT achieved a higher precision and recall than BYTEWEIGHT without recursive function call resolution. BYTEWEIGHT performed above 96% in Linux, while all other tools all performed below 90%. In Windows, we have comparable performance to IDA in terms of precision, but improved results in terms of recall.

Interestingly, we found that the naive method was not able to identify any functions in PE x86-64. This is mainly because `VS` does not use `push %rbp` to begin a function; instead, it uses move instructions.

5.4 Function Boundary Identification

The third experiment evaluated our function boundary identification against existing static analysis tools. As in the last experiment, we compared BYTEWEIGHT, BYTEWEIGHT (no-RFCR), IDA, BAP, and Dyninst, classified binaries by their target, and applied 10-fold validation on each of the classes. The results of our experiment are shown in Table 4.

Our tool performed the best in Linux, and was comparable to IDA in Windows. In particular, for Linux binaries, BYTEWEIGHT and BYTEWEIGHT (no-RFCR) have both precision and recall above 90%, while IDA is below 73%.

	GCC			ICC		
	Precision	Recall	Time(sec)	Precision	Recall	Time(sec)
Rosenblum et al.	0.4909	0.4312	1172.41	0.6080	0.6749	2178.14
BYTEWEIGHT (3)	0.9103	0.8711	1417.51	0.8948	0.8592	1905.34
BYTEWEIGHT (no-norm)	0.9877	0.9302	19994.18	0.9727	0.9132	20894.45
BYTEWEIGHT	0.9726	0.9599	1468.75	0.9725	0.9800	1927.90

Table 2: Precision/Recall of different pattern matching models for function start identification.

	ELF x86	ELF x86-64	PE x86	PE x86-64
Naive	0.4217/0.3089	0.2606/0.2506	0.6413/0.4999	0.0000/0.0000
Dyninst	0.8877/0.5159	–	–	–
BAP	0.8910/0.8003	–	0.3912/0.0795	–
IDA	0.7097/0.5834	0.7420/0.5550	0.9467/0.8780	0.9822/0.9334
BYTEWEIGHT (no-RFCR)	0.9836/0.9617	0.9911/0.9757	0.9675/0.9213	0.9774/0.9622
BYTEWEIGHT	0.9841/0.9794	0.9914/0.9847	0.9378/0.9537	0.9788/0.9798

Table 3: Precision/Recall for different function start identification tools.

For Windows binaries, IDA achieves better results than BYTEWEIGHT with x86-64 binaries, but is slightly worse for x86 binaries.

5.5 Performance

Training. We compare BYTEWEIGHT against Rosenblum et al.'s work in terms of time required for training. Since we do not have access to either their training code or their training data, we instead compare the results based on the performance reported in paper. There are two main steps in Rosenblum et al.'s work. First, they conduct feature selection to determine the most informative idioms – patterns that either immediately precede a function start, or immediately follow a function start. Second, they train parameters of these idioms using a logistic regression model. While they did not provide the time for parameter learning, they did describe that feature selection required 150 compute *days* for 1,171 binaries. Our tool, however, spent only 586.44 compute *hours* to train on 2,064 binaries, including overhead required to setup cross-validation.

Testing. We list the performance of BYTEWEIGHT, IDA, BAP, and Dyninst for testing. As described in section 4, BYTEWEIGHT has three steps in testing: function start identification by pattern matching, function boundary identification by CFG and VSA, and recursive function call resolution (RFCR). We report our time performance separately, as shown in Table 5.

IDA is clearly the fastest tool for PE files. For ELF binaries, it takes a similar amount of time to use IDA and BYTEWEIGHT to identify function starts, however our measured times for IDA also include the time required to run other automatic analyses. BAP and Dyninst have better performance on ELF x86 binaries, mainly because they match fewer patterns than BYTEWEIGHT and do not normalize instructions. This table also shows that function boundary identification and recursive function call resolution are expensive to compute. This is mainly because we use VSA to resolve indirect calls during CFG recovery, which costs more than typical CFG recovery by recursive disassembly. Thus while BYTEWEIGHT with RFCR enabled has improved recall, it is also considerably slower.

6 Discussion

Recall that our tool considers a sequence of bytes or instructions to be a function start if the weight of the corresponding terminal node in the learned prefix tree is greater than 0.5. The choice to use 0.5 as the threshold was largely dictated by Rosenblum et al., who also used 0.5 as a threshold in their implementation. While this appears to be a good choice for achieving high precision and recall in our system, it is not necessarily the optimal value. In the future, we plan to experiment with different thresholds to better understand how this affects the accuracy of BYTEWEIGHT.

While there are similarities betwen Rosenblum et al.'s approach and ours, there are also several key differences that are worth highlighting:

- Rosenblum et al. considered sequences of bytes or instructions immediately preceding functions, called prefix idioms, as well the entry idioms that start a function. Our present model does not include prefix idioms. Rosenblumet al.'s experiments show prefix

	ELF x86	ELF x86-64	PE x86	PE x86-64
Naive	0.4127/0.3013	0.2472/0.2429	0.5880/0.4701	0.0000/0.0000
Dyninst	0.8737/0.5071	–	–	–
BAP	0.6038/0.6300	–	0.1003/0.0219	–
IDA	0.7063/0.5653	0.7284/0.5346	0.9393/0.8710	0.9811/0.9324
BYTEWEIGHT (no-RFCR)	0.9285/0.9058	0.9317/0.9159	0.9503/0.9048	0.9287/0.9135
BYTEWEIGHT	0.9278/0.9229	0.9322/0.9252	0.9230/0.9391	0.9304/0.9313

Table 4: Precision/Recall for different function boundary identification tools.

	ELF x86	ELF x86-64	PE x86	PE x86-64
Dyninst	2566.90	–	–	–
BAP	1928.40	–	3849.27	–
IDA*	5157.85	5705.13	318.27	371.59
BYTEWEIGHT-Function Start	3296.98	5718.84	10269.19	11904.06
BYTEWEIGHT-Function Boundary	367018.53	412223.55	54482.30	87661.01
BYTEWEIGHT-RFCR	457997.09	593169.73	84602.56	97627.44

* For IDA, performance represents the total time needed to complete disassembly and auto-analysis.

Table 5: Performance for different function identification tools (in seconds).

idioms increase accuracy in their model. In the future, we plan to investigate whether adding prefix matching to our model can increase its accuracy as well.

- Rosenblum et al.'s idioms are limited to at most 4 instructions [27, p. 800] due to scalability issues with forward feature selection. With our prefix tree model, we can comfortably handle longer instruction sequences. At present, we settle on a length of 10. In the future, we plan to optimize the length to strike a balance between training speed and recognition accuracy.
- Rosenblum et al.'s CRF model considers both positive and negative features. For example, their algorithm is designed to avoid identifying two function starts where the second function begins within the first instruction of the first function (the so-called "overlapping disassembly"). Although we consider both positive and negative features as well, in constrast the above outcome is feasible with our algorithm.

While our technique is not compiler-specific, it is based on supervised learning. As such, obtaining representative training data is key to achieving good results with BYTEWEIGHT. Since compilers and optimizations do change over time, BYTEWEIGHT may need to be retrained in order to accurately identify functions in this new environment. Of course, the need for retraining is a common requirement for every system based on supervised learning. This is applicable to both BYTEWEIGHT and Rosenblum et al.'s work, and underscores the importance of having a computationally efficient training phase.

Despite our tool's success, there is still room for improvement. As shown in Section 5, over 80% of BYTEWEIGHT failures are due to the misclassification of the end instruction for a function, among which more than half are functions that do not return and functions that call such no-return functions. To mitigate this, we could backward propagate information about functions that do not return to the functions that call them. For example, if function `f` always calls function `g`, and `g` is identified as a no-return function, then `f` should also be considered a no-return function. We could also use other abstract domains along with the strided intervals of VSA to increase the precision of our indirect jump analysis [2], which can in turn help us identify more functions more accurately.

One other scenario where BYTEWEIGHT currently struggles is with Windows binaries compiled with hot patching enabled. With such binaries, functions will start with an extra `mov %edi,%edi` instruction, which is effectively a 2-byte `nop`. A training set that includes binaries with hot patching can reduce the accuracy of BYTEWEIGHT. Because the extra instruction `mov %edi,%edi` is treated as the function start in binaries with hot patching, any subsequent instructions are treated as false matches. Thus, any sequence of instructions that would normally constitute a function start but now follows a `mov %edi,%edi` is considered to be a false match. Consider a hypothetical dataset where all functions start with `push %ebp; mov %esp,%ebp`, but half of the binaries are compiled with hot patching and thus start functions with an extra `mov %edi,%edi`. Half of the time, the sequence `push %ebp; mov %esp,%ebp` will be treated as a function start, but in the other half it will not be treated as such, thus leaving the sequence with a weight of 0.5 in our prefix tree. In order to deal with this compiler peculiarity, we would need give special consideration to `mov %edi,%edi`, treating both this instruction and the instruction following it as a function start for the sake of training.

Although training BYTEWEIGHT for function start identification is relatively fast, training for function *boundary* identification is still quite slow. Profiling reveals that most of the time is spent building CFGs, and in particular resolving indirect jumps using VSA. In future work, we plan to explore alternative approaches that avoid VSA altogether.

Finally, obfuscated or malicious binaries which intentionally obscure function start information are out of scope of this paper.

7 Related Work

In addition to the already discussed Rosenblum et al. [27], there are a variety of existing binary analysis platforms tackle the binary identification problem. BitBlaze [6] assumes debug information. If no debug information is present, it treats the entire section as one function. BitBlaze also provides an interface for incorporating Hex Rays function identification information.

Dyninst [19] also offers tools, such as *unstrip* [31], to identify functions in binaries without debug information. Within the Dyninst framework, potential functions in the `.text` section are identified using the hex pattern `0x55` representing `push %ebp`. First, Dyninst will start at the entry address and traverse inter- and intra-procedural control flow. The algorithm will scan the gaps between functions and check if `push %ebp` is present. This does not preform well across different optimizations and operating systems.

IDA using proprietary heuristics and FLIRT [16] technique attempts to help security researchers recover procedural abstractions. However, updating the signature database requires an amount of manual effort that does not scale. In addition, because FLIRT uses a pattern matching algorithm to search for signatures, small variations in libraries such as different compiler optimizations or the use of different compiler versions, prevent FLIRT from recognizing important functions in a disassembled program. The Binary Analysis Platform (BAP) also attempts to provide a reliable identification of functions using custom-written signatures [8].

Kruegel et al. perform exhaustive disassembly, then use unigram and bigram instruction models, along with patterns, to identify functions [23]. Jakstab uses two predefined patterns to identify functions for x86 code [22, §6.2].

8 Conclusion

In this paper, we introduce BYTEWEIGHT, a system for automatically learning to identify functions in stripped binaries. In our evaluation, we show on a test suite of 2,200 binaries that BYTEWEIGHT outperforms previous work across two operating systems, two compilers, and four different optimizations. In particular, BYTEWEIGHT misses only 44,621 functions in comparison with the 266,672 functions missed by the industry-leading tool IDA. Furthermore, while IDA misidentifies 459,247 functions, BYTEWEIGHT misidentifies only 43,992 functions. To seed future improvements to the function identification problem, we are making our tools and dataset available in support of open science.

Acknowledgments

This material is based upon work supported by DARPA under Contract No. HR00111220009. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA.

References

[1] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity—principles, implementations, and applications. *ACM Transactions on Information and System Security 13*, 1 (2009), 1–40.

[2] BALAKRISHNAN, G. *WYSINWYX: What You See Is Not What You Execute*. PhD thesis, University of Wisconsin-Madison, 2007.

[3] BAP: Binary analysis platform. `http://bap.ece.cmu.edu/`.

[4] BinDiff. `http://www.zynamics.com/bindiff.html`.

[5] BinNavi. `http://www.zynamics.com/binnavi.html`.

[6] BitBlaze: Binary analysis for computer security. `http://bitblaze.cs.berkeley.edu/`.

[7] BOURQUIN, M., KING, A., AND ROBBINS, E. BinSlayer: Accurate comparison of binary executables. In *Proceedings of the 2nd ACM Program Protection and Reverse Engineering Workshop* (2013), ACM.

[8] BRUMLEY, D., JAGER, I., AVGERINOS, T., AND SCHWARTZ, E. J. BAP: A binary analysis platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification* (2011), Springer, pp. 463–469.

[9] CABALLERO, J., JOHNSON, N. M., MCCAMANT, S., AND SONG, D. Binary code extraction and interface identification for security applications. In *Proceedings of the 17th Network and Distributed System Security Symposium* (2010), The Internet Society.

[10] CHA, S. K., AVGERINOS, T., REBERT, A., AND BRUMLEY, D. Unleashing mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (2012), IEEE, pp. 380–394.

[11] CHOI, S., PARK, H., LIM, H.-I., AND HAN, T. A static birthmark of binary executables based on API call structure. In *Proceeding of the 12th Asian Computing Science Conference* (2007), Springer, pp. 2–16.

[12] DAVI, L., DMITRIENKO, A., EGELE, M., FISCHER, T., HOLZ, T., HUND, R., STEFAN, N., AND SADEGHI, A.-R. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Proceedings of the 19th Network and Distributed System Security Symposium* (2012), The Internet Society.

[13] Dia2dump Sample. http://msdn.microsoft.com/en-us/library/b5ke49f5.aspx.

[14] Dyninst API. http://www.dyninst.org/.

[15] ERLINGSSON, U., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. C. XFI: Software guards for system address spaces. In *Proceedins of the 7th Symposium on Operating Systems Design and Implementation* (2006), USENIX, pp. 75–88.

[16] IDA FLIRT Technology. https://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml.

[17] GCC—Function Inline. http://gcc.gnu.org/onlinedocs/gcc/Inline.html.

[18] GUILFANOV, I. Decompilers and beyond. In *BlackHat USA* (2008).

[19] HARRIS, L. C., AND MILLER, B. P. Practical analysis of stripped binary code. *ACM SIGARCH Computer Architecture News 33*, 5 (2005), 63–68.

[20] HU, X., CHIUEH, T.-C., AND SHIN, K. G. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (2009), ACM, pp. 611–620.

[21] KHOO, W. M., MYCROFT, A., AND ANDERSON, R. Rendezvous: A search engine for binary code. In *Proceedings of the 10th IEEE Working Conference on Mining Software Repositories* (2013), IEEE, pp. 329–338.

[22] KINDER, J. *Static Analysis of x86 Executables*. PhD thesis, Technische Universität Darmstadt, 2010.

[23] KRUEGEL, C., ROBERTSON, W., VALEUR, F., AND VIGNA, G. Static disassembly of obfuscated binaries. In *Proceedings of the 13th USENIX Security Symposium* (2004), USENIX, pp. 255–270.

[24] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (2012), IEEE, pp. 601–615.

[25] PERKINS, J. H., KIM, S., LARSEN, S., AMARASINGHE, S., BACHRACH, J., CARBIN, M., PACHECO, C., SHERWOOD, F., SIDIROGLOU, S., SULLIVAN, G., WONG, W.-F., ZIBIN, Y., ERNST, M. D., AND RINARD, M. Automatically patching errors in deployed software. In *Proceedings of the ACM 22nd Symposium on Operating Systems Principles* (2009), ACM, pp. 87–102.

[26] ROSENBLUM, N. The new Dyninst code parser: Binary code isn't as simple as it used to be, 2006.

[27] ROSENBLUM, N. E., ZHU, X., MILLER, B. P., AND HUNT, K. Learning to analyze binary computer code. In *Proceedings of the 23rd National Conference on Artificial Intelligence* (2008), AAAI, pp. 798–804.

[28] SCHWARTZ, E., LEE, J., WOO, M., AND BRUMLEY, D. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Proceedings of the 22nd USENIX Security Symposium* (2013), USENIX, pp. 353–368.

[29] SHARIF, M., LANZI, A., GIFFIN, J., AND LEE, W. Impeding malware analysis using conditional code obfuscation. In *Proceedings of the 16th Network and Distributed System Security Symposium* (2008), Internet Society.

[30] SIDIROGLOU, S., LAADAN, O., KEROMYTIS, A. D., AND NIEH, J. Using rescue points to navigate software recovery. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy* (2007), IEEE, pp. 273–280.

[31] Unstrip. http://www.paradyn.org/html/tools/unstrip.html.

[32] VAN EMMERIK, M. J., AND WADDINGTON, T. Using a decompiler for real-world source recovery. In *Proceedings of the 11th Working Conference on Reverse Engineering* (2004), IEEE, pp. 27–36.

[33] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., AND ZOU, W. Practical control flow integrity & randomization for binary executables. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (2013), IEEE, pp. 559–573.

[34] ZHANG, M., AND SEKAR, R. Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX Security Symposium* (2013), pp. 337–352.