Semantics-Aware Machine Learning for Function Recognition in Binary Code

Shuai Wang, Pei Wang, and Dinghao Wu
College of Information Sciences and Technology
The Pennsylvania State University
University Park, PA 16802, USA
{szw175, pxw172, dwu}@ist.psu.edu

***Abstract*—Function recognition in program binaries serves as the foundation for many binary instrumentation and analysis tasks. However, as binaries are usually stripped before distribution, function information is indeed absent in most binaries. By far, identifying functions in stripped binaries remains a challenge. Recent research work proposes to recognize functions in binary code through machine learning techniques. The recognition model, including typical function entry point patterns, is automatically constructed through learning. However, we observed that as previous work only leverages *syntax-level* features to train the model, binary obfuscation techniques can undermine the pre-learned models in real-world usage scenarios. In this paper, we propose FID, a *semantics-based* method to recognize functions in stripped binaries. We leverage symbolic execution to generate semantic information and learn the function recognition model through well-performing machine learning techniques.**

FID extracts semantic information from binary code and, therefore, is effectively adapted to different compilers and optimizations. Moreover, we also demonstrate that FID has high recognition accuracy on binaries transformed by widely-used obfuscation techniques. We evaluate FID with over four thousand test cases. Our evaluation shows that FID is comparable with previous work on normal binaries and it notably outperforms existing tools on obfuscated code.

***Index Terms*—reverse engineering; machine learning; function recognition;**

I. Introduction

Function recognition in binary programs is critical in reverse engineering [1], [2], [3] and many binary instrumentation and analysis tasks [4], [5], [6]. For example, control-flow integrity validates control flow transfers with rules constructed before execution [7], [8], [9], and function addresses are used to define these rules. In addition, many binary similarity analysis tools launch the similarity test in the granularity of functions [10], [11], [12]. Thus, incorrectly identified functions can drastically impede the similarity test. Recent research work [13] studies the recovery of relocation information from binary code, in which function addresses are the prerequisite to identify code pointers.

Despite the fundamental role of functions in binary instrumentation and analysis applications, function information is usually absent in real world program binaries. The main reason is that to reduce application size and defeat reverse engineering from adversaries, program symbols (including function information) are usually removed from program binaries before distribution. By far, identifying functions in stripped binaries remains a challenge. Some research work has been proposed to discuss the recognition of functions in stripped binaries [14], [15]. Also, most of the widely-used binary reverse engineering and analysis tools have implemented their own methods to identify functions [16], [17], [18]. Note that most of these existing tools rely on handwritten patterns to recognize function *prologue instructions*. However, it is reported that these manually written patterns can become less effective when the input binary is highly optimized. Indeed, it has been pointed out that the industry strength reverse engineering tool IDA-Pro (version 6.5) which features function detection fails to recognize functions in a simple C program compiled by Intel `icc` compiler with the O3 optimization level [19].

Distinguished from these manually written patterns, Rosenblum et al. [20] first consider the function recognition as a machine learning problem; patterns are automatically learned from the training data, which will be installed for usage. In addition, recent work proposes advanced machine learning techniques to recognize functions with improved performance [19], [21]. In general, these machine learning methods automatically learn key features from a large set of binary code to train a detection model, and given a sequence of machine code bytes (or assembly instructions), the "learned" model is able to answer whether the given code bytes start new functions. The machine learning approaches have been evaluated to work better than handwritten pattern matching methods. However, we observe one of the limitations of these methods is that they construct the "key features" purely through machine code bytes (or assembly instructions). That means, the leveraged features only capture the *syntax-level information* in the binary code. Thus, it is reasonable to suspect that program syntax changes can potentially defeat the learned models due to different complication settings or even program obfuscations.

Program obfuscation and diversification transform programs into complex representations which are difficult to understand. Typical obfuscation techniques insert garbage code into random positions of the program, change the control flow structures, and harden control flow predicates into opaque formats. To defeat reverse engineering and analysis from adversaries, besides deleting the debug and relocation information, we assume *software can be obfuscated* before release. To provide better analysis facilities of real world program binaries, we study the function recognition problem against binary obfuscated by commonly-used techniques, which, to our best knowledge, has not been evaluated by previous work systematically.

Symbolic execution captures the semantic information of programs. The key idea of symbolic execution is to use symbolic variables to represent the input and (statically) interpret the code. After symbolic execution, for each initial input, a

```
1  <emit_try_help>:
2   push   %ebp
3   mov    %esp,%ebp
4   sub    $0x18,%esp
5   lea    0x804db36,%eax
6   mov    0x8050144,%ecx
7   mov    %eax,(%esp)
8   mov    %ecx,-0x4(%ebp)
9   call   gettext
```

(a) Original code.

```
eax = 0x804db36
ebx = reg2
ecx = mem1
edx = reg4
esi = reg5
edi = reg6
ebp = reg8
esp = reg7 - 32
```

(b) Assignment formulas corresponding to the original code.

```
1  <emit_try_help>:
2   push   %ebp
3   nop
4   mov    %esp,%ebp
5   sub    $0x18,%esp
6   lea    0x804db36,%eax
7   mov    0x8050144,%ecx
8   mov    %eax,(%esp)
9   mov    %ecx,-0x4(%ebp)
10  call   gettext
```

(c) Obfuscated code.

```
eax = 0x804db36
ebx = reg2
ecx = mem1
edx = reg4
esi = reg5
edi = reg6
ebp = reg8
esp = reg7 - 32
```

(d) Assignment formulas corresponding to the obfuscated code.

Fig. 1: A motivating example.

symbolic formula is generated to represent its output semantics. To better tackle the function recognition problem, we propose to identify functions through the combination of symbolic execution and machine learning. To this end, we first employ an open-source reverse engineering tool Uroboros [13], [22] to disassemble the input binary and recover basic blocks. We then apply symbolic execution on *each individual* basic block to generate corresponding semantics. In particular, we record the assignment formula ([11]) of each register which captures the behavior within one block as well as memory accesses during the interpretation. We select key semantics from the outputs of symbolic execution, and translate them into numeric feature vectors. We utilize well-performing machine learning techniques to learn from the acquired key features and train a recognition model. For any given basic block, the learned model can answer whether it represents a *function entry point basic block* or not, thus identifying a new function. We implement the proposed technique in a tool named FID, and we evaluate FID against a broad set of diverse program binaries produced by three compilers and four optimization levels. The evaluation results show that FID is comparable or even outperforms the state-of-the-art function recognition tools towards the broad sets of test cases we use. We also employ a widely-used program obfuscation tool, Obfuscator-LLVM [23] to measure the obfuscation resilience of FID (Obfuscator-LLVM is referred as `ollvm` later). Binaries transformed by seven obfuscation strategies are produced and evaluated in this paper (including three widely-used binary obfuscation methods and their four compositions). Our evaluation shows that while previous tools suffer from the drastically changed syntax in obfuscated binary code, the performance of FID is quite promising. In sum, this paper makes the following contributions:

- We identify the limitations of previous machine learning based techniques in function recognition, i.e., model learned from syntax-level features can be defeated easily by program syntax changes. We propose a novel technique to extract the semantics and learn a more robust model. We implement our proposed approach as a practical tool, FID.
- We evaluate a broad set of normal and obfuscated program binaries. Evaluation shows that our approach can successfully capture the semantic information across various compilers and optimization levels. Our evaluation also reports that FID can outperform previous available tools against multiple widely-used obfuscation transformations and their compositions.

The rest of the paper is organized as the following. We first present a motivating example in §II. We then give the overview of FID in §III-A, iterate critical design choices of FID in the following subsections of §III and evaluates FID in §IV. We present the discussion in §V, review related work in §VI and conclude the paper in §VII.

II. Motivating Example

We observed that previous function recognition methods can become malfunctional in front of program syntax changes. We present an example in Figure 1, in which a machine learning based function recognition tool ByteWeight [19] misidentifies a function entry point.

To set up this test, we first compile all the 32-bit GNU Coreutils binaries (version 8.23) using `LLVM` 3.6 and optimization O0. We train ByteWeight (bap-byteweight in BAP v0.99 [24]) to learn a recognition model from all these binaries. This version of ByteWeight captures informative machine code bytes to train the model. ByteWeight also has another implementation which takes assembly instructions to train the model [25]. In the rest of this paper we refer to the byte-level ByteWeight as BW-Byte while the other one as BW-Instr.

Garbage code insertion obfuscates programs by inserting meaningless instruction sequences into the code. To present a straightforward and informative example, We insert garbage code to obfuscate the syntax of one Coreutils program binary (`basename`). To this end, we disassemble the program binary of `basename`, insert one `nop` instruction at the beginning of the function `emit_try_help` (line 3 in Figure 1c), and reassemble the instrumented output into an executable. Figure 1a presents the prologue instruction sequence of `emit_try_help` before obfuscation, and Figure 1c shows the obfuscated code. We then employ BW-Byte to identify functions from both the original and the obfuscated binaries; we report that while BW-Byte can correctly recognize this `emit_try_help` function from the original binary, the same function cannot be recognized from the obfuscated code.

BW-Byte constructs weighted prefix trees to represent typical function entry point patterns, each tree node maintaining one machine byte. We consider the main reason for the misidentification is because the inserted `nop` defeats the matching towards the pre-learned tree structures. To illustrate the hidden similarity between the original and obfuscated code, we present the *assignment formula* of eight registers through symbolic execution ([11]). Assignment formulas capture the code semantics in terms of input and output relations. We initialize each register with a symbolic variable as the input

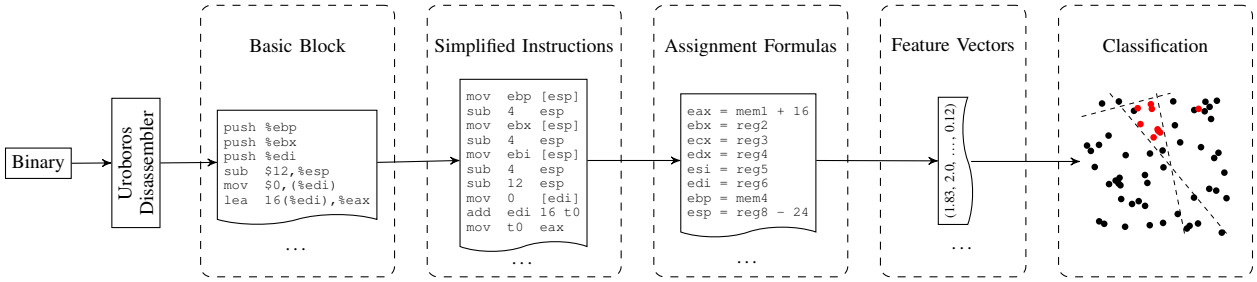

Fig. 2: The workflow of FID.

(e.g., `reg1`), and during interpretation, every memory access towards uninitialized region (e.g., line 6 in Figure 1a) creates a new symbolic variable as well (e.g., `mem1`). The interpretation outputs are shown in Figure 1b and Figure 1d; the assignment formulas of the original and obfuscated instruction sequences—as can be expected—are equivalent. Note that while most formulas have one symbolic variable, formula of stack register `esp` contains a subtraction operation of 32. Typically for programs on the x86 architecture, stack needs to grow to store local variables, which decrements the stack register. In sum, the recognition model trained from program syntax could become *unreliable* in front of even simple syntax changes, while the semantics can usually be preserved regarding such changes.

III. Design

We now outline our approach for function recognition in program binaries. To this end, we train a classifier through semantics of function entry point basic blocks. Later, for a given basic block, our classifier is able to answer whether it represents a function entry point or not, thus recognizing a new function. The extracted semantics is mainly represented as assignment formulas, which describe basic block's behavior regarding the input and output relation of registers.

FID is built on top of Uroboros, an open-source binary disassembly and instrumentation platform [13]. The input program binaries are disassembled and maintained as its internal data, and Uroboros provides utilities to perform inspection and manipulation. While some program instrumentation facilities are provided already, the analysis component of Uroboros is quite insufficient. In this paper, we extend Uroboros with multiple analysis functionalities.

Scope and Assumptions. FID is mainly designed to recognize functions inside x86 ELF binaries without debug or relocation information. We evaluate it in test cases compiled by different compilers, optimization levels and commonly-used obfuscations. Careful readers may notice that FID extracts semantics of each basic block. Thus, correct disassembling and basic block recovery are the prerequisites of FID. In this paper, we assume the disassembling and basic block recovery are mostly reliable. We discuss these assumptions in §V.

As previously mentioned, one motivation of our research is that syntax-based pattern is untenable or even misleading due to syntax changes. Therefore, in this research, besides `call` instructions identified inside the code section of the program binary, FID does not take program syntax into consideration. In addition to the trained recognition model, the destinations (i.e., the callee) of `call` instructions are used to reveal more functions. This design choice is detailed in §III-G and §V.

A. Workflow

Figure 2 shows the overall workflow of FID. FID takes a stripped binary as input and employs Uroboros to recover program control flow structures (including each basic block). FID visits each recovered basic block and launches the implemented analysis components.

As Uroboros is mainly designed to support binary instrumentation, assembly instructions maintained by it are not parsed into *analysis-friendly* formats. Therefore similar to other binary analysis tools, FID first simplifies the complex representations maintained by Uroboros and lift them into easy-to-analysis formats. This process exposes instructions with implicit memory operations into corresponding explicit expressions and simplifies composite instruction operands (e.g., indirect addressing).

In the next phase, we launch a symbolic execution engine to produce the semantics of each basic block. Note that as we are only analyzing each individual block, it is not necessary to track the intra and inter-procedure execution information. We capture the *assignment formulas* for eight general-purpose registers and also record the memory access behaviors in this step (§III-B). Given all the acquired semantics, we then select *informative* features and trim off redundancy before learning (§III-C). We emphasize this step is necessary as the deliberately-selected key features can boost the learning process and improve the performance.

We now have the representations of the semantics each basic block has; the assignment formulas and memory access behaviors describe operations a basic block will perform. However, assignment formulas are purely *syntactic*; learning directly from assignment formulas are very challenging. Therefore, we then seek to translate semantics into *numeric feature vectors*. We extract multiple numeric features from both *lexical* and *syntactic* aspects of an assignment formula (§III-D).

With all the numeric vectors collected, we then discuss how we launch the learning process and train the recognition model (§III-E). In addition, our study shows that binary compiled by different compilers may have different feature distribution, and to present a practical tool, we undertake a *pre-classification* step, determining which compiler the input binary is compiled from (§III-F). After that, for a given binary, we use the model learned from binaries compiled only by the identified compiler to predicate.

Similar with previous work [19], [26], we improve the accuracy through control flow analysis, i.e., identifying function

call instructions in the disassembled code. Motivated by the low recognition precision of previous tools (§IV-C), one of our central design choice is to present *conservative* feature selection which guarantees high precision rate, and improve the recall rate with *call instruction collection* (§III-G).[1]

B. Basic Block-level Symbolic Execution

After acquiring the simplified code, the first step is to leverage symbolic execution to interpret instruction sequences corresponding to each basic block. For each basic block, we collect the assignment formula of every 32-bit general-purpose register. We also record the memory access behavior a basic block commits during the execution. As mentioned previously, symbolic execution engine implemented in FID initializes the interpretation at the beginning of every basic block; this design choice can largely improve the practicability, as typical challenges in analyzing real-world large size binaries, such as inter-procedural analysis, are not considered. Our implementation follows the common design of a symbolic execution engine. We leverage bit vectors defined by the Z3 SMT solver [27] to construct the input value of each register. The symbolic variables initialized through Z3 allow bit-level arbitrary computation, and after aggregating assignment formulas representing the output semantics, we pass formulas to the Z3 solver to simplify the expressions before further analysis.

FID also maintains a lookup table to represent memory contents and how each position is accessed through memory addressing formulas (e.g., `4*reg1 + 4`, where `reg1` is the input variable of register `eax`). Memory read operations on stack is likely to indicate access on function parameters, which is considered as a key feature of function starting blocks (see §III-C for details). Therefore after execution, FID iterates each recorded memory read operation, and if any of the memory read formula contains the input symbolic variable of the stack register `esp`, this memory access formula will be recorded. Note that it is likely to use other registers instead of `esp` to access stack, as we statically interpret the instructions and keep track of the memory, dataflow from `esp` to other registers can be captured as well. After the interpretation, assignment formulas and memory read formulas of each basic block are dumped out for further analysis.

C. Select Informative Semantics

Before we "learn" a model from all the acquired data in §III-B, we first select the key features that are mostly informative in this research. Indeed, our preliminary test shows that by deliberately selecting a subset from all the outputs in §III-B, we could notably improve the performance of FID.

Stack Registers. We observe that most function entry point basic blocks will create new stack frame and reserve spaces to allocate local variables. That means, stack register `esp` and `ebp` are likely to be adjusted in typical function entry point basic blocks. Figure 4 presents an example, demonstrating how stack register `esp` is used in a typical function beginning basic block. As shown in Figure 4a, register `esp` is utilized to reserve 92 bytes on the stack to store local variables. Note that besides the explicit subtraction operation on `esp` (line 5 in Figure 4a), `push` opcode (line 1–4) also implicitly decrements the input value of `esp` by 16 bytes (this implicit effect has been translated into explicit statements before symbolic execution). Figure 4b presents the assignment formulas corresponding to instructions in Figure 4a. Assignment formulas of register `esp` contain a subtraction operation of 92.

In general, stack registers are majorly *informative*, and we pick their assignment formulas (formulas of register `esp` and `ebp`) to learn.

Memory Read. Although stack registers are informative, actually in a typical function call context, both caller and callee can manipulate the stack registers. To future distinguish caller and callee, we elaborate on how we select key features from memory access behaviors.

In this research, we capture the stack memory *read* operations through register `esp`, which is likely to indicate typical parameter read operations at the beginning of functions. Figure 3 presents typical memory access instructions in a function call context (this example is from a GNU Coreutils program `printf`). As shown in Figure 3d, memory positions pointed by `reg8+4` and `reg8+8` which represent the memory positions of the first two function parameters, are all visited by the callee (`reg8` is the input variable of stack register `esp`). Whereas on the caller side no memory read can be found.

In general, we assume stack memory read operations are informative in identifying function entry points, if the memory access formula follows certain addressing patterns. Recall in §III-B the symbolic execution engine has iterated all the memory read addressing formulas and dumped out formulas containing the input variable of `esp`. We then check the presence of memory addressing formulas following such pattern `reg8 + 4*n`, where n can equal to 1, 2, and 3 (suppose `reg8` is the input variable of `esp`, and we assume each stack memory fetching is 4-byte aligned). Such addressing formulas indicate memory read towards the first three potential parameters of a function. Functions with three or more parameters will have the same feature vectors (i.e., `(Present, Present, Present)`) in this step.

D. Translate Assignment Formulas into Numeric Vectors

Several previous work seeks to recognize equivalent assignment formulas through a theorem prover [28], [29]. Given two formulas, a theorem prover is able to prove the equivalence between them, thus identifying program units (e.g., two basic blocks) that are semantically equivalent. However, despite its disinformative results (a prover can only tell "match" or "unmatch"), we emphasize this equivalence-seeking approach is not suitable in our usage scenario, as we are more interested in the gradual similarity.

On the other hand, we observe that most data mining methods take *numeric vectors* to train the model. Enlightened by recent research [30], we seek to translate the acquired semantics into numeric vectors to support a forthright learning process. We also notice that some machine learning algorithms are able to use more complex representations (e.g., string and tree kernels [31]). We leave it as one future work to explore

[1] In this paper, *precision* represents the percentage of function entry points identified that are correct; *recall* is the percentage of real function entry points identified as such. We also calculate *F1 score* in evaluation sections, which is the harmonic mean of precision and recall; naturally, the higher F1 score is, the better a learned model is considered in general.

```
mov    $0x805248e,0x4(%esp)
mov    -0x10(%ebp),%eax
mov    %eax,(%esp)
call   c_strcasecmp
```

(a) Caller's basic block.

```
eax = mem1
ebx = reg2
ecx = reg3
edx = reg4
esi = reg5
edi = reg6
ebp = reg7
esp = reg8
```

(b) Assignment formulas of the caller's basic block.

```
<c_strcasecmp>:
 push   %ebp
 mov    %esp,%ebp
 push   %esi
 push   %ebx
 sub    $0x20,%esp
 mov    0x8(%ebp),%esi
 mov    0xc(%ebp),%ebx
 cmp    %ebx,%esi
 jne    0x804cf85
```

(c) Callee's basic block.

```
eax      = reg1
ebx      = mem1
ecx      = reg3
edx      = reg4
esi      = mem2
edi      = reg6
ebp      = reg8 - 4
esp      = reg8 - 44
[reg8+4] = mem3
[reg8+8] = mem4
```

(d) Assignment formulas and memory reads of the callee's basic block.

Fig. 3: Memory access behaviors in an inter-procedure control transfer.

```
1  push   %ebp
2  push   %ebx
3  push   %edi
4  push   %esi
5  sub    $0x4C,%esp
6  mov    0x68(%esp),%eax
7  mov    0x64(%esp),%edx
8  mov    0x60(%esp),%esi
9  test   %edx,%edx
```

(a) A function entry point basic block.

```
eax = mem1
ebx = reg2
ecx = reg3
edx = mem2
esi = mem3
edi = reg6
ebp = reg7
esp = reg8 - 92
```

(b) The corresponding assignment formulas.

Fig. 4: Stack register adjustments in a function entry point.

challenges in adopting such advanced models in mining symbolic formulas.

In this step, we choose to extract and combine *lexical* and *syntactic* features from assignment formulas; each feature as a *numeric value*. Most lexical features are captured directly from formulas' textual representations, while syntactic features are acquired from the parsed abstract-syntax trees (ASTs). Besides, we also capture three boolean (0/1) *stack* features based on the stack memory access behaviors. Note that each assignment formula produces 8 features, and we capture formulas of `esp` and `ebp` (§III-C). Thus, for each basic block, we construct a numeric vector with 19 elements (8*2+3).

Lexical Features. Table I shows the lexical features we extract from the textual representations. We obtain the number of operators and constants by directly analyzing the text. As for the token related feature, we employ Python library `tokenize` to calculate the total number of tokens. We are particularly interested in the subtraction operations of stack registers in typical function entry point basic blocks, and to this end, we identify two features regarding the presence of subtraction operations and their (potentially) small operands.[2]

Syntactic Features. In this step, we extract features that can be ignored in the lexical analysis. In particular, as each assignment formula can be parsed into a *syntax tree*, we extract syntactic features on top of the parsed tree. Table II presents features we utilized; We calculate the maximum levels of nested parentheses and the maximum depth of an AST as two features. Considering function prologue block delivers unique assignment formulas, it shall be accurate to assume ASTs of prologue blocks and other blocks would yield different similarity distributions when comparing to randomly selected

[2]"Small operands" refer to operands of subtraction operations that are less than a threshold. In our prototype implementation this threshold is 65536.

TABLE I: Lexical features.

Feature	Definition
numOperator/length	the number of occurrences of operators divided by the formula length of characters
numToken/length	the number of tokens divided by the formula length of characters
numConstant/length	the number of constants divided by the formula length of characters
decOperator/length	the number of subtraction operators divided by the formula length of characters
decNum/length	the number of "small operands" in subtraction operations divided by the formula length of characters

TABLE II: Syntactic features.

Feature	Definition
maxNestingDepth	the maximum levels of nested parentheses
maxDepthASTNode	the maximum depth of an AST
aveTreeDistance	the average tree edit distance between the target AST and 50 random picked ASTs

ASTs. Hence, we calculate similarity for each AST with other randomly select ASTs in our dataset. *Tree edit distance* is employed to measure the similarity, and we use Python library `zss`, which implements the Zhang-Shasha algorithm [32] to calculate the distance. Without losing generality, we randomly select 50 ASTs and calculate their tree edit distances towards one target AST.

Stack Features. As in the previous step we have checked the presences of three stack memory accesses that indicate function parameter read (§III-C), here we create three boolean (0/1) features for them. Zero indicates the absence and one is for the opposite.

Normalization: Although it may not be obvious, the *lexical* and *syntactic* feature extractions have implicitly "normalized" the assignment formulas (normalization here means generalizing a formula so that it can match formulas with similar structures). The reason is that we extract structural and tokenism information from the formulas (e.g., the total number of constant numbers), and ignore concrete values in the formula. The "normalization" can usually help us identify more functions. For example, if one function entry point basic block has an assignment formula `esp = reg8 - 4 + mem4`, then it is reasonable to assume a basic block with formula `esp = reg3 - 8 + mem5` belonging to a

TABLE III: Classifiers used by the majority voting mechanism.

Models	Settings
LinearSVC	penalty parameter C=16.0
AdaBoost	number of weak learners=100
GradientBoosting	number of boosting stage=100; learning rate=1.0; random state=0

TABLE IV: Boolean features to distinguish different compilers.

Feature	Definition
decESP	Does `esp` contain subtraction operator?
decEBP	Does `ebp` contain subtraction operator?
memREAD	Can we identify stack memory read operations on memory position `reg8` + 4*n (n can equal to 1, 2 and 3)?

function entry point as well since two formulas produce the same lexical and syntactic features.

Even though we distinguish ourselves from previous work as we extract features from the semantics while they focus on syntaxes, normalization is considered as a general optimization for both.

E. Classification

After producing all the numeric feature vectors, we then employ data mining methods to train a recognition model. Our preliminary test shows that multiple machine learning techniques have good performance. To present a well-performing and robust classifier, we decide to employ the *majority voting* mechanism on top of multiple learning algorithms. A typical majority voting classifier combines multiple learning methods and use a majority voter to predict. A majority voting approach can rule out weakness of each individual method, which should be more adaptable in our scenario.

As shown in Table III, our majority voting classifier contains three classic learning methods. Besides settings shown in Table III, we use the default value for all the other parameters of the three learning methods. We use these methods from Python machine learning library `scikit-learn`.

F. Distinguishing Different Compilers

Our preliminary test on three compilers (`gcc`, `icc` and `LLVM`) shows that while binaries compiled by `gcc` and `icc` share quite similar feature distribution, `LLVM` has a slightly different distribution (§IV-A). Given this observation, we construct a pre-classification step before predication in the real-world usage scenarios, determining whether a given program binary is compiled by `LLVM` compiler or `gcc/icc` (we put binaries compiled by these two compilers into one group as they have similar feature distributions according to our observation). After that, we recognize functions from the input with a model trained from binaries compiled only by the corresponding compiler (i.e., `LLVM` or `gcc/icc`).

The question we seek to answer in this step is comparable to a classic pattern recognition problem, i.e., given an image with thousands of pixels (in our case, it is compared to a program binary with thousands of basic blocks), which category does this image belong to. Enlightened by research work in that field, we classify program binaries according to the feature *distribution* of informative basic blocks. We first select representative basic blocks from thousands of candidates each binary contains, and then calculate the distribution of these selected blocks regarding some revealing features. We use the acquired distributions (in the format of numeric vectors) to train a classification model. After the training step, for a given binary code (i.e., the *input*), our classification model is able to answer which compiler it is compiled from (the *output*). We elaborate on our method in terms of a three-step approach as below:

The first step is to select "critical" basic blocks from all the blocks a binary contains. Naturally, basic blocks corresponding to function entry points should be considered as informative candidates in our research. However, our study has shown that without knowing which compiler it belongs to, identifying functions from obfuscated binaries through pre-learned models could become problematic (low recall rate in our case). Thus, we conservatively select functions identified by the callees of `call` instructions (as shown in Table VII, we can discover about 50% functions through this method).

The semantics of each basic block is translated into a feature vector with 19 elements (§III-D); as most of them are *continuous* variables, it is—if possible at all—quite challenging to calculate a feature distribution. Instead, as shown in Table IV, the second step constructs five boolean features (note that `memREAD` stands for three features, and `reg8` is the input variable of register `esp`) from the numeric features we already acquire (§III-D). As these features are all in boolean distribution, the overall distribution space (2^5) is small and practical. We then calculate the distribution of function starting basic blocks with respect to these 2^5 (i.e., 32) variants and train a model using a majority voting classifier with the same settings (§III-E). Evaluations in this step are detailed in §IV-B.

G. Call Instruction Collection

Our preliminary study shows that when analyzing complex binary code, it is *not always possible* to achieve low false positive and negative rate at the same time. Besides, we have observed that some of previous machine learning based tools can have relatively high recall rate with quite low precision rate (§IV-C). This is not satisfying in developing security applications (e.g., control-flow integrity [7], [6], [9]), as low precision rate indicates many instructions are incorrectly considered as function entry points, which potentially leaves more opportunities for attackers to hijack the control flow.

To present a practical tool, one of our central design choice is to preserve low false positive rate through deliberately-selected *conservative* features, and reduce the false negative rate with additional control flow analysis. To catch functions that are missed by the trained model, we extend the function entry point list through *call instruction collection*. That is, for a given call instruction, if we can identify its operand (say, the callee) in the code section, the callee is considered a new function beginning. Functions identified by this approach will be added to the final result of FID, thus reducing the false negative rate. We name this technique FID-CIC later in this paper.

H. Function Boundary Identification

Naturally, after recognizing function entry points, the next step is to recover the function boundary, i.e., identifying both the entry point and (multiple) exit points of functions. Most of

the existing work recovers the function boundary information through control flow analysis [19], [26]; starting from the identified function entry point, they traverse the intra-procedural control flow to rebuild the control flow graph, thus recovering the function boundaries. Actually given the identified function entries, as the intra-procedural CFG recovery techniques are mostly well-developed, function boundary identification is a matter of *engineering effort*. So our major effort in this paper is to present novel techniques in recovering the function entry points.

BYTEWEIGHT adopts one baseline method to split the whole code section into multiple regions according to the identified function beginnings; each region stands for one function. Indeed besides typical challenges such as overlapping functions in highly-optimized binary code, this "naive" method has been proved as quite reliable [19]. FID provides this method to recover the function boundary. Indeed, we can utilize the *value-set analysis* to recover indirect control destinations and precisely reconstruct the CFG [33]. Thus, the function boundaries can be distinguished even for overlapped functions. We leave it as one future work to extend FID with the precise recovery of function boundaries.

IV. EVALUATION

We undertake a three-step evaluation in this research. The first step evaluates FID in function entry point recognition of normal binaries; three compilers and four optimization levels are employed to generate test cases in this step. We then test FID in distinguishing which compiler an input binary is compiled from. The third evaluation is on obfuscated code. We leverage the Obfuscator-LLVM [23] (referred to as `ollvm`) to obfuscate all the test cases with three widely-used obfuscation methods and their compositions (in total 7 strategies). All the optimization levels are used to generate obfuscated code in this evaluation. We evaluate FID in terms of three standard criteria, i.e., *precision*, *recall*, and *F1 score*. In general, our experiments aim to address the following questions:

- Is FID resilient to compiler and optimization changes (§IV-A)?
- Is FID capable of answering which compiler an input binary is compiled from (§IV-B)?
- Is FID resilient to widely-used code obfuscation techniques (§IV-C)?

Before we present the evaluation of our approach, we first introduce the data set we use and how we acquire the ground truth for comparison.

Data Set. Our evaluation are designed to compare with the cutting-edge research and industrial binary analysis tools who features the function recognition functionality. We choose to employ a widely-used program set, i.e., GNU Coreutils as the test set in our research. GNU Coreutils consists of 106 binaries which provides diverse tasks on Linux operating systems such as textual processing, system management, and arithmetic calculation. We compile the test cases with three compilers (`gcc`, `icc` and `LLVM`) and four optimization levels to produce "normal" program binaries for evaluation.

To evaluate the resilience to program obfuscation, we employ `ollvm` in our experiments. `ollvm` is a set of obfuscation passes implemented inside `LLVM` compiler suite, which provides three widely-used obfuscation methods, i.e., instruction replace [34], opaque predicate insert [35], and control-flow flatten [36] to obfuscate the inputs. All of these methods are widely-used in typical program obfuscation tasks. In this paper, we leverage all the implemented obfuscation methods, together with their *compositions* (i.e., combining multiple methods to obfuscate) to produce binaries with complex structures. We present the obfuscation strategies we use in Table V. Note that each column name corresponds to the abbreviated name we use in evaluation. In summary, our data set consists of three variables:

TABLE V: Obfuscation strategies used in the evaluation.

Obfuscation Methods	ins	opq	flt	mix1	mix2	mix3	mix4
Instruction Replace	✓			✓	✓		✓
Opaque Predicate Insert		✓		✓		✓	✓
CFG Flatten			✓		✓	✓	✓

Compiler. We use GNU `gcc` 4.7.2, Intel `icc` 14.0.1 and `LLVM` 3.6 to produce test binaries.

Optimization Level. For both "normal" and obfuscated binary evaluation, we test all the optimization levels, i.e., O0, O1, O2 and O3.

Obfuscation Methods. We test program binaries obfuscated by 7 different strategies; each strategy is evaluated regarding 4 optimization levels as well.

In total, 4,240 (1,272 normal binaries and 2,968 obfuscated binaries) unique test cases are evaluated in our work.

Ground Truth and Tool Usage. All the test cases are compiled with the symbolic and debug information, and it is easy to get the ground truth (i.e., function beginning addresses) by disassembling the binary. Indeed we acquire the ground truth by disassembling the code section of each test case with GNU tool `objdump`, and extract all the functions with their starting address information by using `grep`.

The symbolic and debug information will then be removed from test binaries using GNU tool `strip` before analyzed by FID. Note that while FID and BYTEWEIGHT can directly output the identified function starting address, IDA-Pro recovers functions as its internal data structure for analysis and transformation. For IDA-Pro, we write scripts to dump out the function information. As previously mentioned, BYTEWEIGHT provides two syntax-based function recognition methods which generate machine byte or assembly instruction-based models. Both of them are evaluated in our research. we summarize the tools we evaluated below:

BW-Byte: The machine byte-level BYTEWEIGHT has been integrated into BAP [16]. We use BAP version 0.99 for evaluation (the newest version by the time of writing) [24].

BW-Instr: The instruction-level BYTEWEIGHT is provided through a virtual machine image [25]. The image is downloaded and configured to use.

IDA-Pro: We use IDA-Pro version 6.6 with all the function identification options enabled.[3]

A. Normal Code

We first evaluate FID in all the normal programs. As previously mentioned, we utilize three compilers and four

[3]Although the newest version of IDA-Pro is 6.9 by the time of writing, there is no improvement for function identification in x86 ELF binaries according to its release notes [37], [38], [39].

TABLE VI: Ten-fold validation on different compilers with different optimization levels.

Opt. Level	LLVM			gcc			icc		
	Precision	Recall	F1 Score	Precision	Recall	F1 Score	Precision	Recall	F1 Score
O0	0.828	0.980	0.898	0.961	0.978	0.969	0.961	0.979	0.970
O1	0.868	0.933	0.899	0.958	0.951	0.954	0.958	0.952	0.955
O2	0.792	0.961	0.868	0.957	0.946	0.951	0.957	0.945	0.951
O3	0.826	0.961	0.888	0.961	0.955	0.958	0.961	0.857	0.906
average	0.829	0.959	0.889	0.959	0.958	0.958	0.959	0.933	0.946

TABLE VII: Comparison with different tools (the "baseline" method only uses *call instruction collection* to recognize functions).

	Precision	Recall	F1 Score
Baseline	1.000	0.527	0.690
IDA-Pro	0.998	0.600	0.750
BW-BYTE	0.788	0.954	0.863
BW-INSTR	0.996	0.997	0.996
FID	0.916	0.959	0.930

optimizations to compile programs in GNU Coreutils, resulting into 1,272 test cases. To demonstrate the proposed semantics-based model in FID, *call instruction collection* (§III-G) is *not* used in this step.

We used ten-fold validation in this step. In general, this validation divides the total data set into ten subsets and tests each subset with the model trained by the remaining 9. Table VI presents the evaluation results against different compilers and optimizations. The precision and recall rates represent the average of the ten tests. On average, FID has 0.916 precision, 0.959 recall and 0.930 F1 score for the 1,272 test cases. While the precision rate of binaries compiled from the `LLVM` compiler is slightly lower than the other two, the overall data is convincing. Indeed, most of the evaluation criteria on `gcc` and `icc` compiled binaries are quite *stable* to around 96% (besides the recall rate of `icc` O3). We interpret this as a promising result to show the semantics-based technique implemented in FID has good performance against various compilers and optimization levels.

We compare FID with existing research and industry de facto tools, which features handwritten patterns or machine learning based function recognition. We also present the "baseline" method, i.e., only leveraging *call instruction collection* to recognize functions (§III-G). We evaluate all of them using the same test cases. Table VII presents the average performance results; FID notably outperforms the baseline method, IDA-Pro and BW-BYTE. Indeed by comparing with the baseline method, we have shown how FID can effectively improve the performance through the machine learning-based method. BW-INSTR can marginally outperform FID on normal code with no obfuscation; later we will see how the semantics-based model implemented in FID can surpass BW-INSTR on obfuscated code. Note that this evaluation *only* tests the learned model, and in practice, *call instruction collection* (§III-G) can always provide additional information to improve the performance of FID.

B. Different Compilers

In this section, we present the evaluation on distinguishing different compilers. As aforementioned (§III-F), for any given input binary, we aim to develop a classifier which can answer whether this binary is compiled by `LLVM` or `gcc/icc`.

Table VIII presents the performance of ten-fold validation against all the benign code. Most of the binaries compiled by `gcc/icc` can be correctly classified, with small errors (over 0.97 precision rate). As for the binaries compiled by `LLVM`, we report the recall rate is slightly lower than the other group. In particular, our finding shows that binaries compiled by `LLVM` and optimization O1 have similar distributions with binaries compiled by `gcc/icc` to certain degree. Nevertheless, given 1.000 precision and over 0.85 recall rate, we still interpret it as a promising result to show FID can recognize which compiler the input binary is compiled from for most of the cases.

TABLE VIII: Ten-fold validation on distinguishing different compilers.

Opt. Level	gcc/icc		
	Precision	Recall	F1 Score
O0	1.000	1.000	1.000
O1	1.000	1.000	1.000
O2	0.977	1.000	0.989
O3	0.940	1.000	0.969
Average	0.979	1.000	0.989
Opt. Level	**LLVM**		
	Precision	**Recall**	**F1 Score**
O0	1.000	1.000	1.000
O1	1.000	0.775	0.873
O2	1.000	0.84	0.913
O3	1.000	0.84	0.913
Average	1.000	0.863	0.926

TABLE IX: Evaluation on distinguishing different compilers on obfuscated binary code.

	Precision	Recall	F1 Score
O0	1.000	1.000	1.000
O1	1.000	0.391	0.488
O2	1.000	1.000	1.000
O3	1.000	1.000	1.000
Average	1.000	0.848	0.918

Our tentative evaluation shows that training using data from binaries compiled by O0 and O2 optimization level can lead to stable performance, so to evaluate the obfuscation-resilience, we train the model using data from Coreutils binaries compiled by `LLVM` and `icc` compilers with O0 and O2 optimization levels (in total 424 program binaries),[4] and test the trained model towards all the obfuscated binary code (seven obfuscation methods and four optimization levels). Table IX presents the average performance regarding different optimizations. We report besides four types of obfuscated binaries compiled by `LLVM` and O1 optimization, most of the obfuscated binaries perform flawlessly in this evaluation. This is consistent with our evaluation in Table VIII.

It is always possible to improve the recall rate when scarifying some precision. Besides, some tricks such as analyzing the exported function name and mangling schemes can also provide us with more clues. We leave it as one future work to

[4]Considering the similarity of binaries compiled by `icc` and `gcc`, we choose to only use binaries compiled by one of them. Our test reports similar test results when substituting with `gcc` compiled binaries.

TABLE X: Evaluation on obfuscated code compiled with O3 optimization level. FID-CIC (III-G) outperforms all the other tools in terms of F1 score (i.e., the harmonic mean of *precision* and *recall*), which demonstrates the resilience of our technique towards obfuscated code.

Obf.	IDA-Pro			BW-Byte			BW-Instr			FID			FID-CIC		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
ins	1.000	0.474	0.643	0.719	0.929	0.811	0.911	0.920	0.915	0.958	0.683	0.798	0.966	0.839	0.898
opq	1.000	0.551	0.710	0.540	0.937	0.685	0.811	0.859	0.834	0.959	0.655	0.778	0.969	0.867	0.915
flt	1.000	0.489	0.656	0.730	0.924	0.816	0.716	0.936	0.811	0.913	0.607	0.729	0.933	0.806	0.865
mix1	1.000	0.543	0.704	0.569	0.931	0.706	0.768	0.907	0.832	0.955	0.586	0.726	0.968	0.840	0.899
mix2	1.000	0.489	0.657	0.494	0.935	0.647	0.685	0.915	0.783	0.896	0.621	0.733	0.918	0.809	0.860
mix3	1.000	0.560	0.718	0.395	0.929	0.554	0.671	0.942	0.784	0.943	0.608	0.740	0.958	0.835	0.892
mix4	1.000	0.565	0.722	0.444	0.925	0.600	0.703	0.941	0.805	0.849	0.557	0.672	0.895	0.842	0.868
Average	1.000	0.524	0.688	0.556	0.930	0.696	0.752	0.917	0.826	0.925	0.617	0.740	0.944	0.834	0.886

TABLE XI: Average performance evaluation on obfuscated code.

Opt. Level	FID		
	Precision	Recall	F1 Score
O0	0.979	0.852	0.911
O1	0.900	0.555	0.685
O2	0.944	0.616	0.745
O3	0.925	0.617	0.740
Average	0.937	0.660	0.774
Opt. Level	**FID-CIC**		
	Precision	**Recall**	**F1 Score**
O0	0.981	0.936	0.958
O1	0.933	0.885	0.907
O2	0.957	0.833	0.891
O3	0.944	0.834	0.885
Average	0.954	0.872	0.911

improve the recall rate of binaries compiled by `LLVM` and O1 optimization. Overall, we assume FID can clearly distinguish whether an input binary is compiled by `LLVM` compilers or `gcc/icc` for most of the cases.

C. Obfuscated Code

The next step is to evaluate FID against the obfuscated binary code. In this test, we train a recognition model with normal binary code, and test the trained model with obfuscated code. Apparently, this is how FID is supposed to work in practice. Note that as obfuscated binaries are all compiled by `LLVM` compiler, as discussed in §III-F, we train FID with normal binaries compiled by `LLVM`. To be consistent with §IV-B, we train FID with normal binaries compiled by optimization level O0 and O2 (in total 212 binary code). We also train BW-Instr and BW-Byte with the same settings.

We first report detailed results regarding the most challenging setting, that is, obfuscations with optimization O3. Table X reports the performance data. Note that BW-Instr also provides the functionality to improve the learned model with *call instruction collection* (§III-G), therefore to present a fair comparison, BW-Instr is configured with this functionality (column four). We do not configure BW-Byte with *call instruction collection*, as the high recall rate of BW-Byte shows small space for improvement (as aforementioned, "recall" rate represents the percentage of real function entry points that is identified). As can be expected, FID-CIC outperforms all the other tools in terms of average F1 score, which is a strong evident to prove FID is resilient to obfuscated code.

In particular, our evaluation shows that while precision of FID is high whether we leverage *call instruction collection* or not, recall increases when it is applied (comparing column five and six in Table X). This is consistent with our assumption, i.e., we conservatively select semantics features to guarantee precise recognition (low false positive rate), and eliminate false negative (improve recall rate) through *call instruction collection*.

Although BW-Instr shows better recall compared with FID, FID outperforms BW-Instr in terms of precision and F1 score. Thus, we interpret FID can have much lower false positive in analyzing obfuscated code. Again, we emphasize the recall rate can be *improved* by analyzing indirect function calls through value-set analysis, which is left as our future work.

We also reported the average performance score against all four optimization levels. As shown in Table XI, the average precision is over 0.95, and recall is over 0.87. Note that our evaluation shows that with more complex optimization applied, the recall rate decreases. We consider the main reason is that advanced optimization techniques are likely to decrease the need to use stack for parameter passing and locate variable allocation. In general, given overall 0.91 F1 score, we interpret that FID is capable of identifying functions even in front of binary code obfuscated by widely-used techniques with various optimization levels.

D. Execution Time

Our experiments are launched on a server machine with Intel Xeon(R) E5-2690 CPU, 2.90GHz and 128GB memory. In this section we report the time consumption of FID.

Feature Generation. FID takes 123.2 CPU hours to process all the normal binary code, and 1659.9 CPU hours for all the obfuscated code; the process time of each test case is recorded from starting to disassemble until finishing producing the numeric feature vectors. Naturally, as obfuscation complicates control flow structure and generates more basic blocks, it is reasonable to take more time to process.

Our study shows that there are two tasks taking more time than others, i.e., symbolic execution (§III-B) and tree edit distance computation (§III-D). On the other hand, while it takes a relatively long *CPU* time to process, as FID boosts several tasks (e.g., symbolic execution) with multithreading facilities, the *real* execution time is indeed much shorter. We report FID takes 23.1 real hours to process normal binary code and 483.8 real hours for obfuscated binary code. On average, it takes *7.2 minutes* to analyze one binary code.

Model Training. Our ten-fold validation training takes 3.62 CPU hour (§IV-A), and the training time with only `LLVM` O0 and O2 binary code (§IV-C) takes 178.1 CPU seconds. We consider the training time is in general promising.

Predication. Given the trained model and numeric feature vectors extracted from an input binary, predication is straightforward. We report that the predication time is 679.53 CPU seconds (§IV-C). That is, on average it takes about 0.23s to recognize functions in one binary.

V. Discussion and Limitations

Binary Disassembling. Correct disassembling is a prerequisite step in our research. BinCFI [6] uses an iterative disassembling algorithm that guides the employed linear disassembler with a validator to correct potential disassembling errors. Uroboros [13] re-implements this algorithm as a prerequisite for program relocation symbol recovery. It has been evaluated that this algorithm can successfully disassemble real-world large applications and libraries with no error (e.g., FireFox 5, Wireshark v1.6.2). In general, in this research we assume the disassembling functionality is reliable even for large size real-world applications.

While most of the binary diversification and obfuscation methods are designed to impede code similarity test, Linn et al. present obfuscation techniques to impede static disassembling [40]. However, techniques have been proposed to defeat these obfuscation techniques [14]. In addition, recent research work has proposed techniques to defeat a long-living challenge in binary code analysis–self modifying code–with good results [41]. Although FID is not equipped with such advanced techniques, we consider it an engineering effort to solve most well-known challenges in disassembling obfuscated binary code.

Basic Block Identification. It is reported that most indirect addressing takes *code pointers* in code and data sections as the destinations of control transfers [13]. Uroboros identifies code pointers from random data in both code and data sections, and it separates the code into multiple basic blocks according to the collected pointers and control transfer instructions. Thus, although CFGs may not be precisely recovered due to the indirect addressing, as long as code pointers are correctly collected, the basic block recovery should be reliable.

Indirect addressing through dynamic calculation undermines the basic block recovery (as it is disclosed that Uroboros only identifies the *base address* for such addressing mode). Nonetheless, existing research work proposes straightforward solutions to analyze such addressing mode [6], and there have been available tools to provide similar functionality as well (e.g., DynInst [26]). Although as a proof-of-concept prototype, FID does not implement these techniques, it is only a matter of engineering effort to cooperate them with FID.

Obfuscation Methods. Our motivating example highlights the underlying design limitation of previous work, i.e., they only learn from syntax-level features. In this paper, we propose techniques to capture the semantics information; the evaluation in this work on thousands of program binaries demonstrates that our method preserves promising evaluation results even in front of drastically changed program syntax. Note that in this paper, we are not stating to defeat *all* sort of obfuscations, but mainly focus on outperforming existing research in terms of commonly-used obfuscation methods, while keep comparable performance regarding normal binary code. Note that many binary reverse engineering tasks (e.g., indirect memory addressing) are indeed undecidable; results from computability theory suggest that it could be very difficult to propose impeccable solutions through static analysis. Hence we consider it is challenging–if not impossible at all–to propose a technique that can become the "panacea" to defeat all obfuscation methods. Again, proposing such technique is not our main purpose here.

In addition, FID is designed as orthogonal to its underlying disassembler. Although Uroboros is tenable to implement FID as a proof-of-concept prototype, it is always possible to improve the disassembling and basic block recovery by using commercial reverse engineering tools (e.g., CodeSonar [42]).

VI. Related Work

Function identification is considered as the foundation for many binary analysis and test applications, and there has been a number of related work in this topic [14], [15], [20], [43], [44], [26]. Rosenblum et al. [20] propose to use machine learning based approach to address the function recognition problem. Bao et al. [19] detail multiple challenges in this topic, and propose a weighted prefix-tree based approach to train the recognition model. As aforementioned, their work have two implementations by learning both machine code bytes and instruction sequences, and it has been evaluated that ByteWeight can significantly outperform previous work [20] regarding recognition accuracy and processing time. As ByteWeight releases its implementation, in this work, we have mainly compared FID with it. Shin et al. [21] proposes a deep-learning based technique to recognize functions by learning from machine code bytes. It is reported that their work has better performance and less processing time than ByteWeight. We do not test their tool as it is not available. However, as discussed in §V (and also admitted in their paper), arbitrary inserted garbage code could break the trained model, while FID would not be trapped in such cases.

While some recent research work reports good performance using machine learning approaches, they essentially share a similar design choice, i.e., they mainly capture the *syntax-level* information to learn. Conceptually, while syntax can be easily extracted, the learned model suffers from syntax changes, e.g., binary *obfuscation* and *diversification*. On the other hand, in this paper we shows that by learning from deliberately-selected semantic information, obfuscation-resilience is achievable for widely-used techniques.

VII. Conclusion

In this paper, We present FID, a semantics-based function recognition tool in binary code. FID leverages symbolic execution to extract assignment formulas and memory access behaviors. The acquired information will then be used to train a function recognition model with well-performing machine learning techniques. Our evaluation shows that FID is comparable with the state-of-the-art tools on normal binaries, and outperforms them on obfuscated binary code.

Acknowledgment

We appreciate the anonymous reviewers for their valuable feedback. We also thank Tiffany Bao for helping us setup ByteWeight and providing valuable feedback. This research was supported in part by the National Science Foundation (NSF) under grant CNS-1652790, and the Office of Naval Research (ONR) under grants N00014-13-1-0175, N00014-16-1-2265, and N00014-16-1-2912.

REFERENCES

[1] E. J. Schwartz, J. Lee, M. Woo, and D. Brumley, "Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring," in *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[2] M. Van Emmerik and T. Waddington, "Using a decompiler for real-world source recovery," in *Proceedings of the 11th Working Conference on Reverse Engineering*, ser. WCRE '04, 2004, pp. 27–36.

[3] I. Guilfanov, "Decompilers and beyond," *Black Hat USA*, 2008.

[4] W. M. Khoo, A. Mycroft, and R. Anderson, "Rendezvous: A search engine for binary code," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. IEEE Press, 2013, pp. 329–338.

[5] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "XFI: Software guards for system address spaces," in *Proceedings of the 7th symposium on Operating Systems Design and Implementation*, 2006, pp. 75–88.

[6] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *Proceedings of the 2013 USENIX Security Symposium*, 2013.

[7] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and Communications Security*. ACM, 2005, pp. 340–353.

[8] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi, "MoCFI: A framework to mitigate control-flow attacks on smartphones." in *Proceedings of the 2012 Symposium on Network and Distributed System Security*, 2012.

[9] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, 2013.

[10] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket execution: Dynamic similarity testing for program binaries and components," in *23rd USENIX Security Symposium*, 2014, pp. 303–317.

[11] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP)*, 2015, pp. 709–724.

[12] "BinDiff," http://www.zynamics.com/bindiff.html.

[13] S. Wang, P. Wang, and D. Wu, "Reassembleable disassembling," in *Proceedings of the 25nd USENIX Security Symposium*, ser. USENIX Security '15, 2015.

[14] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries," in *Proceedings of the 13th Conference on USENIX Security Symposium*, ser. USENIX Security'04, 2004, pp. 18–18.

[15] H. Theiling, "Extracting safe and precise control flow from binaries," in *Proceedings of the Seventh International Conference on Real-Time Computing Systems and Applications*, 2000, pp. 23–30.

[16] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A binary analysis platform," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, 2011.

[17] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A new approach to computer security via binary analysis," in *Proceedings of the 4th International Conference on Information Systems Security*, 2008.

[18] S. Hex-Rays, "IDA Pro: a cross-platform multi-processor disassembler and debugger," 2014.

[19] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "ByteWeight: Learning to recognize functions in binary code," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, 2014.

[20] N. Rosenblum, X. Zhu, B. Miller, and K. Hunt, "Learning to analyze binary computer code," in *Proceedings of the 23rd National Conference on Artificial Intelligence*, 2008, pp. 798–804.

[21] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *24th USENIX Security Symposium*, 2015, pp. 611–626.

[22] S. Wang, P. Wang, and D. Wu, "Uroboros: Instrumenting stripped binaries with static reassembling," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, ser. SANER'16, 2016.

[23] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM – software protection for the masses," in *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection*. IEEE, 2015, pp. 3–9.

[24] "ByteWeight with Machine Byte Code Features," https://github.com/BinaryAnalysisPlatform/bap.

[25] "ByteWeight with Assembly Instruction Features," http://security.ece.cmu.edu/byteweight/.

[26] W. R. Williams, X. Meng, B. Welton, and B. P. Miller, "Dyninst and MRNet: Foundational infrastructure for parallel tools," in *Proceedings of the 9th Annual Parallel Tools Workshop*, 2015.

[27] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[28] D. Gao, M. K. Reiter, and D. Song, "BinHunt: Automatically finding semantic differences in binary programs," in *Proceedings of the 4th International Conference on Information Systems Security*, 2008.

[29] J. Ming, M. Pan, and D. Gao, "iBinHunt: Binary hunting with inter-procedural control flow," in *Proceedings of the 15th International Conference on Information Security and Cryptology*, 2013, pp. 92–109.

[30] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt, "De-anonymizing programmers via code stylometry," in *24th USENIX Security Symposium*, 2015, pp. 255–270.

[31] S. V. N. Vishwanathan and A. J. Smola, "Fast kernels for string and tree matching," in *Proceedings of the 15th International Conference on Neural Information Processing Systems*, ser. NIPS'02, 2002, pp. 585–592.

[32] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems," *SIAM J. Comput.*, vol. 18, no. 6, pp. 1245–1262, Dec. 1989.

[33] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *Compiler Construction*. Springer, 2004, pp. 5–23.

[34] F. B. Cohen, "Operating system protection through program evolution," *Comput. Secur.*, vol. 12, no. 6, pp. 565–584, Oct. 1993.

[35] C. Collberg, S. Martin, J. Myers, and J. Nagra, "Distributed application tamper detection via continuous software updates," in *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012.

[36] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated software diversity," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, 2014.

[37] "IDA-Pro 6.7," www.hex-rays.com/products/ida/6.7/index.shtml.

[38] "IDA-Pro 6.8," www.hex-rays.com/products/ida/6.8/index.shtml.

[39] "IDA-Pro 6.9," www.hex-rays.com/products/ida/6.9/index.shtml.

[40] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, ser. CCS '03, 2003, pp. 290–299.

[41] G. Bonfante, J. Fernandez, J.-Y. Marion, B. Rouxel, F. Sabatier, and A. Thierry, "Codisasm: Medium scale concatic disassembly of self-modifying binaries with overlapping instructions," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15, 2015, pp. 745–756.

[42] "CodeSonar," http://www.grammatech.com/products/binary-analysis.

[43] "IDA-FLIRT," https://www.hex-rays.com/products/ida/tech/flirt.shtml.

[44] "DynInst-unstrip," http://www.paradyn.org/html/tools/unstrip.