In-Memory Fuzzing for Binary Code Similarity Analysis

Shuai Wang and Dinghao Wu
The Pennsylvania State University
University Park, PA 16802, USA
{szw175, dwu}@ist.psu.edu

***Abstract*—Detecting similar functions in binary executables serves as a foundation for many binary code analysis and reuse tasks. By far, recognizing similar components in binary code remains a challenge. Existing research employs either static or dynamic approaches to capture program syntax or semantics-level features for comparison. However, there exist multiple design limitations in previous work, which result in relatively high cost, low accuracy and scalability, and thus severely impede their practical use.**

In this paper, we present a novel method that leverages in-memory fuzzing for binary code similarity analysis. Our prototype tool IMF-SIM applies in-memory fuzzing to launch analysis towards *every* function and collect traces of different kinds of program behaviors. The similarity score of two behavior traces is computed according to their longest common subsequence. To compare two functions, a feature vector is generated, whose elements are the similarity scores of the behavior trace-level comparisons. We train a machine learning model through labeled feature vectors; later, for a given feature vector by comparing two functions, the trained model gives a final score, representing the similarity score of the two functions. We evaluate IMF-SIM against binaries compiled by different compilers, optimizations, and commonly-used obfuscation methods, in total over one thousand binary executables. Our evaluation shows that IMF-SIM notably outperforms existing tools with higher accuracy and broader application scopes.

***Index Terms*—in-memory fuzzing; code similarity; reverse engineering; taint analysis;**

I. Introduction

Determining the similarity between two components of binary code is critical in binary program analysis and security tasks. For example, binary code clone detection identifies potential code duplication or plagiarism by analyzing the similarities of two binary components [57], [50]. Patch-based exploitation compares the pre-patch and post-patch binaries to reveal hidden vulnerabilities fixed by the patch [11]. Malware research analyzes similarities among different malware samples to reveal malware clusters or lineage relations [8], [36].

So far, a number of binary similarity analysis tools have been developed with different techniques. The de facto industrial standard tool BinDiff identifies similar functions mostly through graph isomorphism comparison [20], [23]. This algorithm detects similar functions by comparing the control flow and call graphs. Moreover, recent research work proposes advanced techniques to identify the hidden similarities regarding program semantics [50], [21], [56], [15].

Given the critical role of similarity analysis in binary code, we observe several weaknesses in existing research. For example, dynamic analysis-based methods usually have coverage issues [35], [54], [59], [64], [14], which naturally impedes their work from testing *every* function in binary code. Typical static methods can test any program component, but they may suffer from real-world challenging settings such as compiler optimizations or program obfuscations [20], [23], [57], [19]. To overcome such limitations, we propose IMF-SIM, which leverages in-memory fuzzing to solve the coverage issue and reveal similarities between two binary code components even in front of real-world challenging settings.

Fuzz testing, or fuzzing, is a widely-used software testing technique that exercises a program by providing invalid or random data as inputs. Compared with traditional testing techniques, where a single input is used to test one execution trace, fuzzing can largely improve the code coverage and increase the chances of exposing hidden bugs. Despite its simplicity in the concept, fuzz testing is proven as robust and effective in the real-world settings and is widely used for software testing [24], [51], [32], [31], [26], [62]. While most standard fuzz testing mutates the program inputs, we have noticed a special fuzzing technique that is designed to directly fuzz the content of assembly registers or memory cells, i.e., *in-memory fuzzing* (Chapters 19 and 20 in [61]). In-memory fuzzing can start one fuzzing execution at any program point. While most testing techniques suffer from generating proper inputs to reach certain program points, in-memory fuzzing can start at the beginning of any instruction. Hence, *every* program component becomes testable.

While fuzzing technique is originally proposed for software testing, we observe that rich information regarding the program runtime behavior is indeed revealable during fuzzing without additional efforts. That means, semantics-based similarity analysis shall be boosted through well-designed fuzz testing. To this end, we propose IMF-SIM, a fuzzing based similarity analysis tool that overcomes multiple design limitations of existing research (details are discussed in §II) with higher accuracy and broader application scopes. In particular, IMF-SIM leverages in-memory fuzzing to launch dynamic testing towards every function for multiple iterations and records program runtime behaviors. We collect different kinds of program behaviors (referred as *behavior traces* in this paper) for each function, and behavior traces from two functions are compared through their *longest common subsequence*. For each comparison pair of two functions, we generate a vector including the Jaccard index rates (the Jaccard index rate of two behavior traces is derived from their longest common subsequence) of the behavior trace-level comparisons, and we then label sample vectors to train a machine learning model. Later, given a vector by comparing two functions, the trained model provides a similarity score.

The main contributions of this paper are as follows.

- We identify design limitations of existing research in similarity analysis of binary components, and propose IMF-SIM, a novel method that uses fuzz testing techniques for function-level similarity analysis in binary code.
- IMF-SIM employs the *in-memory fuzzing* technique, which is originally designed for assembly-level testing. We propose several advanced methods to overcome the unique challenges and reveal the hidden power of the fuzzing technique in our new context.
- Benefit from its runtime behavior based comparison, IMF-SIM is effectively resilient to challenges from different compilers, optimizations, and even commonly-used program obfuscations. We evaluate IMF-SIM on over one thousand widely-used binaries produced by various compilation and obfuscation settings. Our evaluation shows that IMF-SIM has promising performance regarding all the settings and outperforms the state-of-the-art tools.

II. Motivation

In this section, we summarize the limitations of previous binary code similarity analysis work and also highlight the motivation of our research. We discuss the design choices of existing work in multiple aspects here.

Dynamic Analysis. Many program runtime behaviors, such as memory accesses, function calls, and program return values, are descriptors of program semantics to some extent. Some existing work leverages dynamic analysis and software birthmarks for (function-level) similarity analysis [54], [59], [64], [38], [14], [37], [35]. However, an obvious issue for existing dynamic analysis is the potential low coverage of test targets (e.g., functions). Theoretically, generating program inputs to guarantee the reachability of every code component is an undecidable problem. That means, those existing dynamic tools are in general unable to analyze *all* the functions in binary executables, which drastically limits its application scope. Although Concolic testing techniques can improve the coverage [13], [30]; however, most of these techniques suffer from scalability issues on large programs.

Static Analysis. Static similarity analysis can start from any program point and fully cover every test target, which is, on this aspect, better than dynamic techniques [53], [50], [49], [22], [56], [15], [65], [52]. Indeed, many recent work in this field leverages symbolic execution techniques to retrieve program semantics. By leveraging constraint solvers, semantics equivalent program components are identifiable in a rigorous way. However, static analysis can have limited capabilities for real-world programs, such as complex control flows (e.g., opaque predicates [45]), libraries, or system calls. In addition, similarity analysis through constraint solving may have noticeable performance penalty [50].

Our proposed technique, IMF-SIM, is derived from typical dynamic testing techniques with concrete inputs. Thus, common challenges for static analysis should not be obstacles for our technique. Moreover, to solve the code coverage issue of previous dynamic tools, IMF-SIM leverages a unique fuzzing paradigm, in-memory fuzzing [61], to launch the execution at the beginning of *every* test target. Furthermore, fuzz testing naturally improves the code coverage within each test component, as it mutates the inputs for iterations and aims to exhaust execution paths in a best effort.

Program Syntax Information. To balance the cost and accuracy, many static analysis-based techniques capture relatively "light-weight" features, such as the number of instructions, opcode sequences, and control flow graph information [34], [22], [20], [23]. However, note that one major application of binary code similarity analysis is for malware study. Thus, to better understand the strength and weakness of similarity testing tools, we should consider commonly-used *obfuscation* techniques as well. Typical obfuscations are designed to change the program syntax and evade the similarity analysis [45]. As a result, syntax-based techniques can become inefficient in the presence of obfuscations. However, as IMF-SIM leverages dynamic analysis to collect program *runtime* behaviors, commonly-used obfuscations should not impede it. We evaluate IMF-SIM on three widely-used program obfuscations, and it shows promising results in all the settings.

III. IMF-SIM

We now outline the design of IMF-SIM. The overall workflow is shown in Fig. 1. To compare two functions in two binary executables, we first launch in-memory fuzzing to execute the functions for iterations, and record multiple kinds of behavior traces (§III-A). The central challenge at this step is the lack of data type information. For example, we have no pre-knowledge on whether a function parameter is of value or pointer type, and misuse of a non-pointer data as a pointer can lead to memory access (pointer dereference) errors. To address this issue, we propose to use *backward taint analysis* to recover the "root" of a pointer data flow whenever a dereference error occurs on this pointer. Later, we re-execute the function and update the recovered dataflow root (e.g., an input of the function) with a valid pointer value (§III-B).

After collecting behavior traces for each fuzzing iteration (third column in Fig. 1), we then concatenate behavior traces of the same kind (e.g., behavior traces representing heap memory accesses) to build the final behavior traces, each of which records one type of runtime behavior through multiple fuzzing iterations. To compare two functions, two behavior traces of the same kind are used to calculate a Jaccard index rate (§III-C); the Jaccard index rates of all the behavior traces are gathered to form one numeric vector for two functions. We label sample vectors according to the ground truth and train a machine learning model (§III-D). Later, for a given vector of two functions, the trained model can yield a similarity score. For a given function f_1 in binary bin_1, to find its matched function in bin_2, we compare f_1 against every function in bin_2, and take the sorted top one matching as the final result.

IMF-SIM consists of four modules, namely the fuzzing module, taint analysis module, similarity analysis module, and machine learning module. The in-memory fuzzing module of IMF-SIM is built on top of Pin, a widely-used dynamic binary instrumentation tool [48]. We develop a plugin of Pin (i.e., a PinTool) for the in-memory fuzzing functionality. This module consists of over 1,900 lines of C++ code. The other three modules are all written in Python, consisting of over 4,700 lines of code.

Scope and Limitations. IMF-SIM is mainly designed for similarity testing among binary components. Although in this

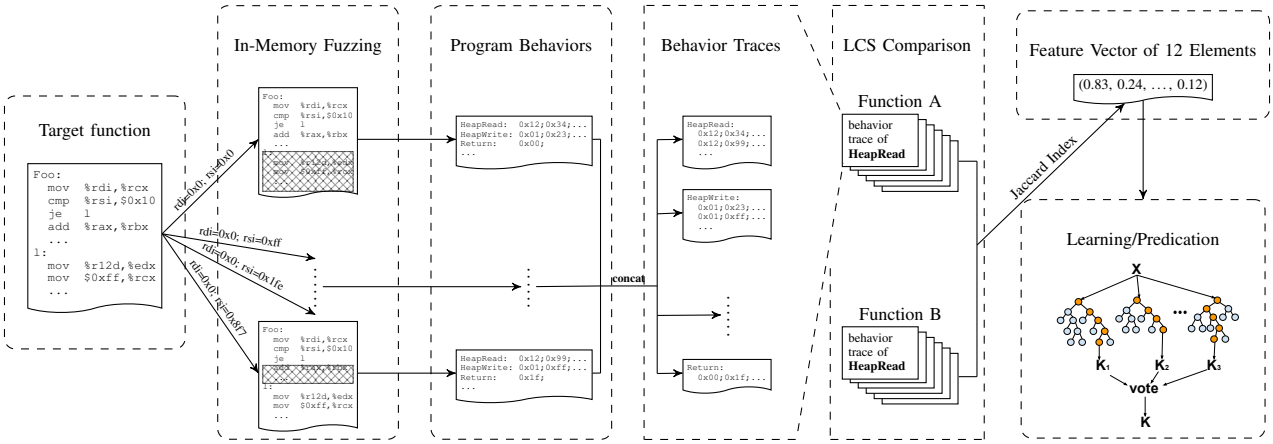

Fig. 1: The workflow of IMF-SIM. **concat** in the figure stands for "concatenation".

research we utilize IMF-SIM to analyze the function-level similarity, as in-memory fuzzing supports to execute any program component (the fuzzing inputs need to be deliberately selected regarding the context), it should be interesting to investigate whether our technique can be used to find similar code to an arbitrary trunk of code instead of a whole function.

IMF-SIM captures the program runtime behaviors, thus, stripped binaries which contain no debug or program relocation information are supported by IMF-SIM. Furthermore, our evaluation shows that syntax changes due to different compilers, optimizations or even commonly-used obfuscations can also be analyzed without additional difficulties.

The employed dynamic analysis infrastructure, Pin, is designed to instrument x86 binaries. In this work, we implement IMF-SIM to instrument binaries on 64-bit Linux platforms (i.e., binaries with the ELF format). It is not difficult to port IMF-SIM to other platforms supported by Pin (e.g., Windows or 32-bit Linux).

Naturally, users need to provide the range information (i.e., the starting and ending addresses) of the test function before processing. Although the function information is indeed absent in most stripped binaries, recent work has provided promising results to precisely recover such information [7], [63].

A. In-Memory Fuzzing

In this section, we introduce the design of the in-memory fuzzing module. Our fuzzing module is designed following the common design of a fuzzer. In general, we first setup the environment for fuzzing (§III-A1). Before each test iteration, we mutate one input and then launch the fuzzing execution (§III-A2). We collect the behavior traces during each fuzzing iteration (§III-A3). After one iteration of fuzzing, we resume from the starting point, mutate one input and launch the next iteration until we have iterated for the predefined times (§III-A4). We now elaborate on each step in details.

1) Environment Setup: To present a fair comparison regarding program behaviors, we first create an *identical* environment before the test of each function. To this end, we set all general-purpose registers to zero and maintain a memory access map. This map will be updated before every memory write operation; the key of each item is the memory address and the value is the memory cell's content before each fuzzing iteration. The memory access map is used to reset the environment to its original status at the end of each fuzzing iteration.

In the absence of program type information on assembly code, it is likely to encounter memory access errors during execution. For each invalid memory access, we use backward taint analysis to recognize the root of the memory address dataflow within the test function and update the root with a valid memory address. To this end, we allocate a chunk of (512 KB) memory (referred as v_mem_1) on the heap when setting up the environment. We also initialize every 32-bit long memory cell in v_mem_1 with a valid memory address that refers to a random position within v_mem_1; this method guarantees most memory accesses through address computation and (multi-level) pointer dereferences are still valid. Since memory write could break valid pointers in v_mem_1, we create another memory region (v_mem_2) with the same size and intercept memory write towards v_mem_1. The intercepted memory writes will be redirected to v_mem_2.

Note that we use the *same* seed to initialize the random number generator before initializing the memory chunk v_mem_1. This approach guarantees the memory chunk is identical among different testing. The pointer that refers to the *middle* of this chunk is used to fix the root of an invalid memory address data flow (§III-B).

At the program entry point, we update the value of the instruction pointer register (i.e., `rip` for the 64-bit x86 architecture); the execution flow is then redirected to the first instruction of the target function. We then launch the above procedure to initialize the context and memory.

2) Input Mutation: As we are focusing on analyzing the function-level similarity, function inputs naturally serve as the mutation targets. According to the calling convention on the 64-bit x86 architecture, caller functions use *six* predefined registers to pass the first six arguments to callees. Functions with more than six parameters use the memory stack to pass additional arguments. IMF-SIM mutates those six predefined registers to fuzz the first *six* (potential) arguments. For a function with equal or less than six parameters, all of its inputs are mutated by IMF-SIM. As for functions with more than six parameters, their first six arguments are mutated.

Typically, the fuzzing process mutates program inputs randomly or following certain predefined rules; both approaches

are widely adopted by fuzzers. Our implementation follows a common approach—step fuzzing—to mutate the inputs. Starting from zero, we increment one input with a predefined step while preserving the others each time; the step is set as `0xff` in our current prototype implementation. We mutate one register for ten times while preserving the rest, which leads to sixty execution iterations in total for one function.

3) Function Execution: During the process, we keep track of multiple kinds of program behaviors; behavior information is used for similarity comparison. In addition, to facilitate the backward taint analysis, we also record the already executed instructions; the context information (including the value of every general-purpose register) of *every* executed instruction is recorded as well. Whenever encountering a pointer dereference error, this information will be dumped out as the input of our taint analysis module (§III-B).

IMF-SIM captures various kinds of runtime behaviors. One important design concept of IMF-SIM is that it treats the target function as a *blackbox*. That means, we only capture the *interaction* between the monitored function and its runtime environment. This design choice should be more resilient towards potential changes on the target program. For example, compiler-optimized programs tend to use registers instead of the memory stack to store function local variables. Hence, monitoring of stack memory access could be impeded by this optimization. We present the captured features as follows ($feature_n$ is abbreviated as f_n):

1) Value read from or write to the program heap (f_1, f_2).
2) Memory address offsets for reading the `.plt.got` section (f_3).
3) Memory address offsets for reading the `.rodata` section (f_4) .
4) Memory address offsets for reading (f_5) or writing (f_6) the `.data` section.
5) Memory address offsets for reading (f_7) or writing (f_8) the `.bss` section.
6) System calls made during execution (f_9).
7) Memory address offsets for accessing *v_mem*$_1$ (f_{10}) or *v_mem*$_2$ (f_{11}).
8) Function return value (f_{12}).

As we intercept the memory accesses, we record the value read from or write to the heap region, respectively (f_1 and f_2). We also record memory address offsets used to access four global data sections (for each memory access, the *offset* is calculated regarding the base address of the target section). `.data` and `.rodata` sections store global data objects, `.bss` stores uninitialized objects while `.plt.got` section stores routines to invoke functions in the dynamically-linked modules. For `.plt.got` and `rodata` sections we keep track of every memory read access as these two sections are read-only (f_{3-4}). We keep both read and write for other three sections (f_{5-8}).

Considering there could exist many zero bytes in the memory, we assume that the memory address offset is a more *informative* feature than the visited content in a memory access. Hence, while the *concrete value* read from or written to the heap region is recorded as f_1 and f_2, we calculate the memory address offsets of the visited memory cells for f_{3-8}. Note that heap has multiple memory allocation strategies, which could put the same data at different places. Thus, we conservatively employ the value instead of the offsets as features for f_1 and f_2. On the other hand, as global data sections in the memory essentially preserve the relative positions in original binary executables, it is mostly safe to use the memory offsets as features.

By reading the symbol tables of the input executables (symbol tables still exist in even stripped executables to provide "exported function" information for dynamic-linkage), memory addresses for accessing the `.plt.got` section can be mapped to dynamic-linked function names. In other words, we have indeed recorded the invoked library functions. Besides, system calls are recorded as well (f_9). Recall we create two special memory regions (*v_mem*$_1$ and *v_mem*$_2$) and fix memory access errors with valid pointers towards those regions (§III-A1). We create two features to represent memory read and write towards them (f_{10} and f_{11}).

Specified by the 64-bit x86 architecture calling convention, callees pass the return value to the call site through register `rax`. To capture potential return value, we record the value of `rax` at the end of every execution (f_{12}). In total, we collect 12 features (as shown in the third column of Fig. 1, each of which is actually a sequence of numeric values) in this step.

4) Termination and Exceptions of One Fuzzing Iteration: During execution, we can encounter normal exits, execution exceptions, and even infinite loops. We rely on the following rules to terminate the current iteration of execution.

$Rule_1$ Function execution normally ends at return or inter-procedural jump instructions.
$Rule_2$ A configured timeout has expired.

The first rule identifies the normal end of a function execution. Note that compiler may optimize return instructions into jump instructions, so in addition to return instructions, we also check inter-procedural jump instructions. When the target function is inside a recursive call, we cannot immediately terminate the execution only by the first rule. Hence, we keep a counter to record the number of invoked target function; the counter will be incremented every time when the target is called and decremented when it is returned. We terminate the fuzzing iteration only when the counter is zero.

The dynamic instrumentor we employed can register several handling routines to receive exceptions and signals. Handling routines check the type of exceptions; errors due to invalid memory accesses will be fixed through backward taint analysis (§III-B). However, given the diverse semantics of test candidates, it is still possible to encounter exceptions that are hard to fix (e.g., some library functions require the parameter to be a valid file descriptor, which is hard to provide in our context). To circumvent such issue, we keep track of the most-recently executed instruction within the target function; execution will be resumed from the next associated instruction in the target function when encountering such exceptions.

5) Fuzzing Stop: Recall our current implementation executes each function for sixty iterations, and at the end of each execution, we check whether we have reached this threshold. If not, we reset the environment, mutate one input, and re-launch the execution at the beginning of the test function.

Since each function is fuzzed for sixty iterations, for each type of monitored program behavior, we indeed collect sixty

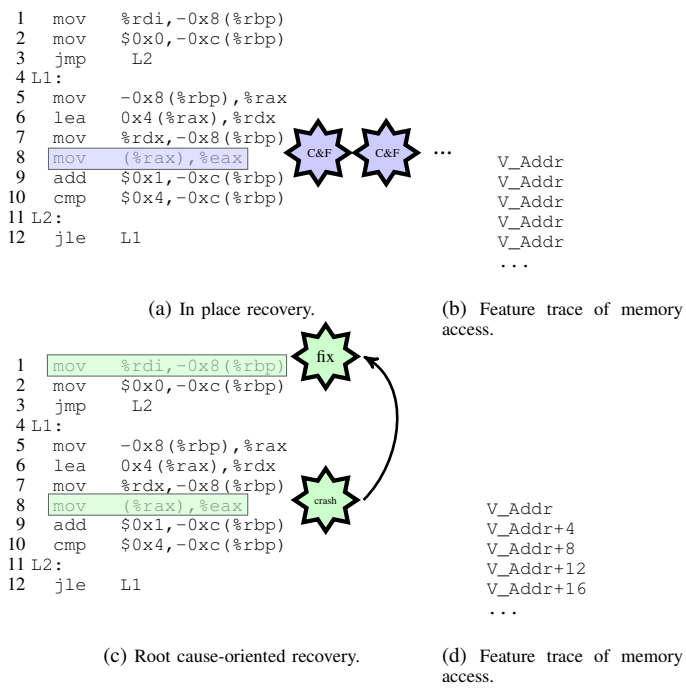

Fig. 2: Comparison between the in-place method and the root cause-oriented method. The assembly instruction is in AT&T syntax. C&F stands for "crash then fix in place". `V_Addr` is a valid memory address used to fix the memory access error.

traces. Before we launch the next step analysis, we concatenate behavior traces describing the same type of program behavior into a long trace. That means, for each function, we finally get 12 behavior traces for usage (as we monitor 12 features).

B. Backward Taint Analysis

As previously discussed, without program type information, it is likely to misuse data of value type as inputs for functions with pointer parameters. If we terminate the execution whenever encountering pointer dereference errors, only incomplete behaviors will be collected.

An "in-place" strategy to solve the issue is to update the memory access through an illegal address with a valid address; the execution can be then resumed from the current instruction. Despite the simplicity, this method does not solve the *root cause* of memory access errors, and the collected behavior traces can become uninformative.

Fig. 2a presents assembly code generated by compiling a function; this function contains a loop to access every element of an array. The base address of the array is passed in through the first argument of the function (i.e., register `rdi`) and stored onto the stack (line 1). Later, the memory address flows to `rax` (line 5). Inside the loop, the base address of the array is incremented (line 5–7) to access each memory cell (line 8). Suppose the function argument `rdi` is incorrectly assigned with data of value type. When memory access error happens (line 8), the "in-place" method discussed above would only update `rax` to facilitate the current memory access at line 8. Memory access fails again (as the pointer on the stack is not updated), and only memory cell "V_Addr" will be accessed and recorded for such case (Fig. 2b).

On the other hand, if we backtrack the pointer data flow to the *root* and update it with a valid pointer (line 1 in Fig. 2c), memory accesses originated from the root would not fail (line 8). More importantly, the typical memory access behavior of a loop—incremental memory visit with a fixed step—can be mostly revealed from the collected behavior trace (Fig. 2d).

Considering such observation, we propose to leverage backward taint analysis to identify the root cause of a memory access failure. Given a pointer dereference error, we taint the pointer and propagate the taint backwards until reaching the root (e.g., a function input parameter). We then re-execute the function from the beginning, and when encountering the root, we update it with a valid pointer. We now elaborate on each step in details.

Given a memory access failure, the first step is to identify the register that is supposed to provide a valid pointer (i.e., the *taint source*). We leverage Pin API `get_base_register` to get the *base register* of this memory access. We take the base register as the starting point of our taint analysis.

We then dump the executed instructions. As mentioned earlier, we record the context information associated with every executed instruction (§III-A3). Here the recorded contexts are also dumped out. The dumped instructions, contexts, and the taint source register are the inputs of our taint analysis module.

The next step is to leverage taint analysis to identify the taint *sink* (the "root cause"). Our taint analysis tool first parses the dumped instructions into its internal representations. Then starting from the taint source, the taint engine iterates each instruction and propagates the taint label backwards. The taint propagation will be stopped after we iterate all the instructions. We then take the *most-recently* tainted code (which can be either a register or a memory cell) as the sink point.

The taint module follows the common design of a taint analysis tool. We create a 64-bit bit vector for each 64-bit register. As registers are represented by bit vectors, tainting is indeed on the *bit-level*. This bit-level tainting guarantees operations on (small length) registers can also access and propagate the taint. To record taint propagation through memory cells, our taint engine keeps a lookup table; each element in the lookup table is a memory address, indicating the referred memory cell is tainted. Tainting operation will insert new elements into this table, while untainting will delete the corresponding entry.

As we perform static analysis on the execution traces (*straight-line code*), the taint analysis is actually quite efficient. In addition, before analyzing each instruction, our taint engine updates registers with concrete values acquired from the context information associated with the current instruction. That means, even for memory read/write operations with composite addressing, we know exactly what the memory address is. Approximations made by most static analysis are not needed.

Table I shows the propagation rules adopted in our work. Six potential backward taint flows are summarized in Table I, and our current policy propagates taint for the first one. When the left variable of an assignment instruction has taint (Policy 1 in Table I), we assume the taint is from the right variable. Thus, we propagate the taint to the right value (could be a register or memory cell) and remove the taint on the left value. Besides this case, we conservatively keep the taint without any propagation.

TABLE I: Taint flows under different conditions and their corresponding backward propagation rules.

Policy	Instruction	Is *t* Tainted?	Is *u* Tainted?	Taint Policy
1	t = u	✓	×	ι(u); ζ(t)
2	t = u	×	✓	
3	t = u	✓	✓	
4	t = t ◇ u	✓	×	
5	t = t ◇ u	×	✓	
6	t = t ◇ u	✓	✓	

Symbol *t* and *u* stand for register, memory access or constant data. ◇ denotes arithmetic or logic operators. ι(b) taints b, while ζ(b) removes the existing taint label from b.

After the taint analysis is complete, we return the discovered sink to the fuzzing module. There could exist multiple memory access errors during the execution of one function, each of which originates from a different taint sink. Thus, the fuzzing module keeps a list of all the identified taint sinks, and fixes encountered sink points by checking the list.

C. Longest Common Subsequence

As presented earlier, we acquire multiple behavior traces by fuzzing a function. To analyze the similarity of two functions, we compare each behavior trace of the same kind. In particular, we utilize the Longest Common Subsequence (LCS) algorithm to calculate the similarity among two traces. The longest common subsequence problem identifies the longest subsequence that appears in all sequences. We note that since LCS allows skipping non-matching elements, it naturally tolerates diverse or noise on the behavior traces due to *program optimizations* or even *obfuscations*.

To facilitate further "learning" step, we seek to create numeric comparison outputs. Thus, instead of directly using the common subsequence, a Jaccard index is derived from the LCS of two traces. The calculated Jaccard index is a numeric value ranging from 0 to 1. Thus, to compare two functions, we build a numeric vector including Jaccard indices of all the recorded features.

Jaccard Index. Given two behavior traces (T_1 and T_2), the Jaccard index is calculated in the following way:

$$J(T_1, T_2) = \frac{|T_1 \cap T_2|}{|T_1 \cup T_2|} = \frac{|T_1 \cap T_2|}{|T_1| + |T_2| - |T_1 \cap T_2|}$$

The $|T_1 \cap T_2|$ is the length of LCS between T_1 and T_2, while $|T_1|$, $|T_2|$ represent the length of T_1 and T_2, respectively. The $J(T_1, T_2)$ is a numeric value ranging from 0 to 1; two sequences are considered more similar when the Jaccard index is closer to 1.

As numeric vectors built in this step describe the similarity of two functions regarding different program behaviors, each element in the vector is indeed a "feature" in the comparison. Thus, we denote these vectors as "feature vectors" following the common sense.

D. Training

After collecting feature vectors, we train a machine learning model for prediction. Later, for a feature vector from the comparison of two functions, the trained model yields a probability score (range from 0 to 1) showing the possibility that two functions are matched.

In this research, we choose to train a tree-based model (i.e., *Extra-Trees* [28]) for prediction. Comparing with kernel-based models (e.g., SVM [33]) or neural network based models, tree-based models do not assume the underlying distributions (e.g., Gaussian) of the data set and are considered more understandable for human beings. Tree-based model essentially trains a set of decision trees on sub-samples and leverages a voting process on top of the decision trees for the final results [58]. While the well-known tree-based model, i.e., Random-Forest [10], uses several optimizations in constructing decision trees, Extra-Trees makes them at random. Our tentative test shows that Random-Forest model can be trapped in over-fitting (e.g., good performance on training sets while worse on test sets), and we consider the over-fitting is mostly due to its optimizations. However, without further information to solve this issue, we choose Extra-Trees in our research regarding its more generalized settings. Besides, without the optimizations, Extra-Trees usually has a shorter training time.

We leverage the Python machine learning library sklearn for the model training process [55]. Although the trained model is usually used for a binary-prediction problem, i.e., given a feature vector by comparing two functions, it yields whether they are matched (1) or not (0), we use the trained model in a slightly different way. That is, we configure the model to return a probability score (i.e., the **similarity score**) for each input vector, indicating the possibility to label this vector as one (i.e., two functions are matched). As for the parameters, we set the number of estimators (i.e., number of decision trees) as 100 which leads a time/accuracy trade-off, and use the *default* value for all the other parameters.

Before the training process, we first label the feature vectors. A feature vector is labeled as a positive sample when it is produced by comparing functions that are supposed to be matched. Such ground truth is acquired by comparing the function name; we assume two functions are similar only if they have the same name. (we introduce how we get such ground truth for our evaluation in §IV). We label the rest as negative samples. We also balance the positive and negative training samples before training. Further discussions on the applicability of our machine learning-based approach will be presented in §V.

IV. Evaluation

We now present the evaluation in this research. There are 12 features recorded along the analysis (§III-A3), and our first evaluation studies the importance of each captured feature regarding our predication task. We then launch a standard ten-fold validation towards IMF-SIM; binaries produced by nine different compilation settings are tested. The second step is cross-compiler validation; we launch experiments regarding six cross-compiler settings. In the third step evaluation, we validate IMF-SIM by comparing normal binary code and their obfuscated versions. Note that both second and third step evaluations are *cross-model* validations. That is, we train the model with data used in step one, and verify the trained model with new data produced for experiments in step two and three. This cross-model validation step is necessary as when deploying in real world scenarios, IMF-SIM is supposed to train with standard training sets (e.g. data used in step one) and do prediction towards binaries from various compilation and event obfuscation settings (typical cases are tested in step two and three). We present further discussion in §V.

Similar with previous research [21], [35], [22], we measure the performance of IMF-SIM regarding its *accuracy*, i.e., the percentage of correctly matched functions.[1] For a function f_1 in binary bin_1, we iteratively compare it with functions in binary bin_2 and take the comparison pair with the highest similarity score (i.e., rank-one) as the matched pair. In addition, we also evaluate whether a function can find its correct match when checking the top three and five function pairs (rank-three and rank-five). It is reasonable to assume experts can further identify the correct match with acceptable amount of efforts given three or five function pair candidates. Moreover, there could exist several comparisons that have equivalent similar rates, e.g., the similar rate of rank-6 and rank-5 is equal to 0.720. However, we do not consider such possibility and only take the first N (N is 3 or 5) matches for consideration. We now introduce the data set and the ground truth in the evaluation.

Data Set. We employ a widely-used program set—GNU `coreutils`—as the data set in our research. `coreutils` (version 8.21) consists of 106 binaries which provide diverse functionalities such as textual processing and system management. We rule out binaries with destructive semantics (e.g., `rm`), which leads to 94 remaining programs.

As previously mentioned, we compare normal binary code with their obfuscated versions. We employ a widely-used obfuscator (Obfuscator-LLVM [43]) in our evaluation (Obfuscator-LLVM is referred as OLLVM later). OLLVM is a set of obfuscation passes implemented inside `llvm` compiler suite [46], which provides three obfuscation methods, i.e., instruction substitution [16], bogus control flow (i.e., opaque control-flow predicate [17]), and control-flow flattening [45]. All of these methods are broadly-used in program obfuscation. We leverage all the implemented methods to produce binaries with complex structures. In summary, our data set consists of three variables, leading to 1,140 unique binaries in total:

Compiler. We use GNU `gcc` 4.7.2, Intel `icc` 14.0.1 and `llvm` 3.6 to compile programs.

Optimization Level. We test three commonly-used compiler optimization settings, i.e., -O0, -O2 and -O3.

Obfuscation Methods. we test program binaries obfuscated by three commonly-used methods, which are provided by Obfuscator-LLVM (version 3.6.1).

Comparison with Existing Work. We compare IMF-SIM with the state-of-the-art research work BinGo [15] and Blanket Execution (i.e., BLEX) [21] that deliver function-level similarity analysis. Both research work employ `coreutils` as the dataset in the evaluation. We compare IMF-SIM with them by directly using data provided in their paper (Table V).

We also compare IMF-SIM with the industrial standard binary diffing tool BINDIFF [2]. BLEX and BINGO are also compared with BINDIFF in their paper. We use BINDIFF (v4.2.0) to evaluate all the experiment settings (Table III, Table IV, Table VI). Benefit from well-designed graph isomorphism comparisons [20], [23], BINDIFF is also stated as resilient towards different compilers and optimizations [3].

Ground Truth. Although our technique itself does not rely on the debug and symbol information inside the binary executables, we compile the test programs with debug symbols to get the function information. Particularly, we disassemble the binary code with a disassembler (`objdump` [1]), and then read the function name and range information from the disassembled outputs. The ground truth in our work is acquired by comparing the function names. Two functions are considered matching only if they have the same name. Besides, the range (starting and ending memory addresses) of each function serves as the input of IMF-SIM.

[1]Same with previous research, we use accuracy instead of precision/recall rate for the evaluation. Note that "accuracy" is not equal to precision.

TABLE II: Feature importance evaluation. The definition of each feature is presented in §III-A3.

feature	importance	feature	importance	feature	importance
f_1	0.061	f_2	0.140	f_3	0.103
f_4	0.036	f_5	0.096	f_6	0.015
f_7	0.082	f_8	0.058	f_9	0.137
f_{10}	0.149	f_{11}	0.053	f_{12}	0.070

TABLE III: Ten-fold validation on the similarity score of `coreutils` binaries in terms of different compilation settings. Comparison results with three ranks are provided. We also evaluate BINDIFF with same settings.

	rank-one	rank-three	rank-five	BINDIFF
`gcc` -O0 vs. `gcc` -O3	0.704	0.813	0.846	0.192
`gcc` -O2 vs. `gcc` -O3	0.895	0.955	0.962	0.780
`icc` -O0 vs. `icc` -O3	0.702	0.763	0.796	0.216
`icc` -O2 vs. `icc` -O3	0.971	0.984	0.985	1.00
`llvm` -O0 vs. `llvm` -O3	0.775	0.842	0.864	0.285
`llvm` -O2 vs. `llvm` -O3	0.973	0.993	0.994	0.996
average	0.837	0.892	0.908	0.578

A. Feature Importance

We first evaluate the importance of each feature we captured. That is, we try to answer the question that with respect to different features, how much does each feature contribute to the overall effectiveness of IMF-SIM. The machine learning model implementation (sklearn [55]) provides standard APIs to acquire such data [5].

To this end, we fit our model with all the test samples, and present the importance of each feature. The results are shown in Table II. In general, four features have the importance of over 10%, and eleven features over 5%. The only outlier (f_6) is memory writes towards the global data section, which has an importance of 1.5%. Overall, our evaluation shows that most of the features have noticeable importances, and we interpret our feature selection step (§III-A3) as reasonable.

B. Ten-Fold Validation

In this section, we perform a standard approach, i.e., ten-fold cross validation, to test the performance of IMF-SIM. In general, this validation divides the total data set into 10 subsets and tests each subset with the model trained by the remaining 9. In this step, binary code produced by nine compilation settings are employed. Table III presents the evaluation results of three ranks, respectively. The accuracy rate represents the average of the 10 tests. Even through we train the model with data of six different comparison settings together, we separately verify the trained model regarding test data of each setting, leading to six categories.

On average, IMF-SIM has over 83% rank-one similarity score, and the accuracy rate goes to over 90% regarding the rank-five matches. The evaluation data shows that compiler optimizations indeed affect the performance. On the other

TABLE IV: Cross-compiler validation.

	rank-one	rank-three	rank-five	BINDIFF
gcc -O0 vs. icc -O3	0.634	0.721	0.772	0.187
gcc -O0 vs. llvm -O3	0.660	0.761	0.800	0.199
icc -O0 vs. gcc -O3	0.569	0.682	0.714	0.173
icc -O0 vs. llvm -O3	0.629	0.718	0.756	0.192
llvm -O0 vs. gcc -O3	0.601	0.706	0.734	0.229
llvm -O0 vs. icc -O3	0.599	0.700	0.763	0.223
average	0.615	0.715	0.757	0.201

hand, we observe the performance is notably increased regarding rank-three and rank-five. Hence, we interpret that users are very likely to identify the real matching pair by further analyzing the first three or five matches.

IMF-SIM shows consistent results comparing with BINDIFF regarding icc/llvm -O2 vs. -O3. Both IMF-SIM and BINDIFF yield very high accuracies for these two comparison settings. Our further study on the disassembled outputs shows that assembly code produced by icc/llvm -O2 is indeed similar to their corresponding -O3 level.

In addition, while the comparisons between binaries compiled by llvm (-O0 vs. -O3) show better performance, the evaluation results for gcc and icc (-O0 vs. -O3) are similar. We consider binaries compiled by gcc and icc show similar behaviors regarding features IMF-SIM captures, and our further evaluations report consistent findings. In general, while the de facto industrial tool BINDIFF suffers from heavy compiler optimizations, we interpret IMF-SIM shows promising results regarding such challenging settings.

C. Cross Compiler Validation

In this section, we launch evaluations to compare binaries produced by different compilers. To this end, we train a model leveraging *all six groups of data* used in the first evaluation (§IV-B). The trained model is then used to verify comparisons with cross compiler settings. Given the intuition that compiler optimization "obfuscates" programs (evaluation in Table III is consistent with this intuition), we compare binary code under the most challenging setting, i.e., comparing un-optimized (-O0) binary code with the heavily-optimized (-O3) code.

The evaluation results are presented in Table IV. Stating the challenge, IMF-SIM actually performs well in most of the settings. We observe the comparison between gcc/icc and llvm show similar results, especially for llvm -O0 vs. gcc/icc -O3. We interpret the results are consistent with our findings in §IV-B; binaries compiled by gcc and icc compilers show similar behaviors in front of IMF-SIM. Besides, while comparisons between llvm -O0 and gcc/icc -O3 show relatively low performance in terms of rank-one score, rank-three scores are improved to over 70%. On the other hand, BINDIFF performs poorly in terms of all the settings in this evaluation (on average 0.201 accuracy score).

D. Comparison with Existing Work

As aforementioned, we compare IMF-SIM with two state-of-the-art research work [21], [15]. Table V presents the comparison results (we also include BINDIFF's data for reference). BLEX evaluates results of three different compilation/optimization settings (last three rows in Table V). BINGO reports more settings, and we use 6 settings that overlap with our evaluation (row 2-7). In general, we interpret that all the four

TABLE V: Compare IMF-SIM with the state-of-the-art research work (BLEX [21] and BINGO [15]) and de facto industrial-standard tool (BINDIFF). × means such data is not available.

	IMF-SIM	BINDIFF	BLEX	BINGO
llvm -O0 vs. llvm -O3	0.775	0.285	×	0.305
llvm -O2 vs. llvm -O3	0.973	0.996	×	0.561
gcc -O0 vs. llvm -O3	0.660	0.199	×	0.307
llvm -O0 vs. gcc -O3	0.601	0.229	×	0.265
gcc -O0 vs. gcc -O3	0.704	0.192	0.611	0.257
gcc -O2 vs. gcc -O3	0.895	0.780	0.771	0.470
gcc -O0 vs. icc -O3	0.634	0.187	0.541	×

tools show similar findings that heavy compiler optimizations bring in additional challenges, i.e., comparison between -O2 and -O3 yields better results than -O0 vs. -O3. On the other hand, IMF-SIM can outperform all tools in all these settings.

As previously discussed, BINDIFF, which leverages program control-flow as features to analyze the similarities, shows notably low resilience towards heavy compiler optimizations (comparisons between -O0 vs. -O3 are much worse than -O2 vs. -O3 for BINDIFF). On the other hand, the other three semantics-based tools are considered more "stable" regarding optimizations. BinGo captures semantics information mostly through symbolic execution and I/O sampling, and it has relatively poor performance comparing to the other two dynamic execution-based tools. IMF-SIM and BLEX both acquire more precise information by indeed executing the test targets. However, different from BLEX which forces to execute every instruction of the test target and breaks the original execution paradigm, IMF-SIM fixes pointer dereference error on demand and reveal the original program behavior in a more faithful approach. In sum, we interpret the comparison results as promising.

E. Comparison with Obfuscation Code

In addition to the evaluations of normal binaries, we also launch comparisons between normal binaries and their corresponding obfuscated code. We follow the same approach as §IV-C to train the model. We leverage all the obfuscation methods provided by OLLVM to obfuscate binary code optimized with -O3 (OLLVM employs compiler llvm to compile the code). Given three groups of obfuscated code, we then compare each group with three sets of normal code.

The evaluation results are presented in Table VI. Instruction substitution ("sub.") replaces simple operations (e.g., addition) into semantics-equivalent but syntax-level more complex formats. Despite its efficiency of misleading syntax-based similarity analysis, such obfuscation can hardly defeat IMF-SIM which captures program semantics (runtime behaviors). Bogus control flow ("bcf.") inserts opaque predicates to protect conditional branches; such predicate is usually difficult to be reasoned until runtime [17]. We observed that analysis results become worse regarding this obfuscation technique. Considering our fuzzing strategy as relatively simple, we interpret the results as reasonable; opaque predicate complicates the monitored execution behaviors, and also potentially impedes our fuzzer from "drilling" into the guarded branch.

We also observe that fewer functions are correctly matched when applying the control-flow flattening obfuscation ("fla."). In general, control-flow flattening changes the structure of

TABLE VI: Validation towards obfuscated code compiled by OLLVM with -O3 optimization level (OLLVM employs `llvm` compiler). The name of the obfuscation methods are simplified due to the limited space.

comparison	obf.	rank-one	rank-three	rank-five	BINDIFF
`llvm` -O0	`bcf.`	0.513	0.631	0.686	0.235
`gcc` -O0	`bcf.`	0.385	0.525	0.590	0.190
`icc` -O0	`bcf.`	0.362	0.489	0.549	0.183
`llvm` -O0	`fla.`	0.649	0.753	0.797	0.269
`gcc` -O0	`fla.`	0.576	0.700	0.757	0.199
`icc` -O0	`fla.`	0.561	0.685	0.732	0.174
`llvm` -O0	`sub.`	0.779	0.845	0.870	0.285
`gcc` -O0	`sub.`	0.664	0.760	0.802	0.198
`icc` -O0	`sub.`	0.613	0.723	0.758	0.192
average		0.567	0.679	0.727	0.214

the original control flow into a "flatten" structure; the execution flow is chained by `switch` statements to iterate basic blocks [45]. Considering such design, it is not surprising that the collected behavior traces of IMF-SIM are affected due to this structure-level changes. Nevertheless, comparing with BINDIFF which relies on program control-flow information for analysis, IMF-SIM still demonstrates the obfuscation resilience and outperforms this industrial strength tool.

F. Processing Time

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

Feature Generation. In our evaluations, we test 21 groups of comparison, each of which consists of 94 program pairs. We measure the total feature generation time, from starting to execute the first test binary until finishing producing the feature vector of the last comparison. We report that IMF-SIM takes 1027.1 CPU hours to process all the comparisons (94 pairs * 21 groups). On average, it takes around 0.52 CPU hour to compare two binaries. Note that the report time indeed underestimates IMF-SIM's processing speed, as in our prototype implementation, a binary re-generates behavior traces for each comparison (a binary can participant in multiple comparisons). We also report the processing time of BINDIFF is 8830.8 CPU seconds. BINGO is reported as efficient as well [15]. In general, our study shows that (lightweight) static feature-based tools can take much less time to process (like BINDIFF and BINGO); however, they suffer from relatively worse comparison accuracies according to our evaluations (§IV). On the other hand, BLEX [21] reports to take 57 CPU days (1368 CPU hours) to process 1,140 binaries, while our work takes 1027.1 CPU hours for 1,974 comparison pairs. Although test cases used by IMF-SIM and BLEX are not exactly the same, and BLEX uses different hardware to measure the processing time, we can still see that dynamic-analysis based approaches, despite its much better comparison accuracies, would lead to relatively high performance penalties in general.

Our study reveals one task that takes a large amount of processing time: the calculation of longest common subsequence (LCS). This algorithm itself has a relatively high computation complexity. However, since each comparison task performed by IMF-SIM is independent with each other, the *real* time can be reduced by employing more CPU cores.

Model Training and Prediction. The training of the Extra-Trees model takes 609.8 CPU seconds. Given the trained model and the generated feature vectors, prediction is straightforward. We report that total prediction time is 15389.6 CPU seconds for all the experiments in §IV-B, §IV-C, and §IV-E. Recall we undertake 21 groups of comparisons, each of which consists of 94 binary pairs. Thus, on average it takes 7.8 CPU seconds to compare one pair of binaries. In general, we consider the overall model training and predication as effective. Furthermore, since we are using Extra-Trees as the learning model, it is actually suitable to speedup the *real* training time by employing multiple CPU cores.

V. DISCUSSION

Obfuscations. We have evaluated IMF-SIM on both benign and obfuscated binary code. As discussed in §II, the dynamic testing approach used in IMF-SIM is indeed quite robust even in the presence of code obfuscation techniques; program runtime behaviors are revealed and captured during the execution. On the other hand, previous static analysis based similarity testing may be impeded. According to our experiments, control-flow flattening obfuscation, which transforms the structure of the control-flow, affects the accuracy. Moreover, our evaluation shows that IMF-SIM still preserves a quite promising result with diverse settings (§IV).

Note that many binary reverse engineering and analysis tasks (e.g., indirect memory addressing) are theoretically undecidable; results from computability theory suggest that it could be very difficult to propose impeccable solutions. Hence it is challenging—if possible at all—to propose a technique that can defeat all known or to-be-invented obfuscations. Certainly IMF-SIM is not designed to defeat all of them. Nevertheless, IMF-SIM shows its practical value in the presence of commonly-used obfuscations.

Code Coverage. We have also performed some tentative tests on the code coverage of IMF-SIM before launching the experiments of function similarity. The results are promising, with an average 31.8% instruction coverage. In general, we consider the experiments and the comparisons with state-of-the-art tools have shown promising results given the current coverage. Additionally, code coverage could be further improved through advanced grey-box fuzzing technique [9]; we leave it as one future work to adopt such advanced techniques in IMF-SIM.

On the other hand, we believe pushing the code coverage to the extreme (as "blanket execution" does [21]) may **not** be the best practice in our context (i.e., similarity analysis). Improving the code coverage at a best effort while preserving the *original execution paradigm* shall be a better way to collect features that can truly reveal the program behavior. Our evaluation also reports consistent findings. We present further comparisons with BLEX in §VI.

Applicability of the Machine Learning based Approach. We do not undertake an explicit model tuning step in this work. The key motivation is that by adopting the *default* setting of the well-studied model (except setting the number of decision trees as 100), we have already observed promising results (§IV). Further tuning step can be launched by following standard procedures, which is well documented by the machine learning library we use [4].

We note that our classifier is not over-specialized regarding "obfuscation." We train IMF-SIM with only *normal* program

binaries compiled by three most-commonly used compilers on Linux platforms, and further test the learned model regarding a much broader setting (e.g., commonly used obfuscation methods). Our training set is considered *standard*, which is straightforward to produce and should be used together. This is actually how IMF-SIM is supposed to use in practice; training with a whole set of well-known binaries, and test with other (unknown) settings in the wild. For example, when performing malware similarity analysis for unknown malware samples, malware researchers can employ samples from several commonly-used malware families to train the model, and let this model to decide the similarity among different unknown pieces. In general, learning-based approach has been adopted by very related research on binary code reverse engineering [7], [60]. We believe our work can shed light on future research to employ machine learning techniques to analyze (binary) code similarities.

Future Work. Although in this paper we adopt in-memory fuzzing for function-level similarity analysis, as previously mentioned, in-memory fuzzing is indeed capable of launching test towards any program component.

Besides, we also plan to further extend our technique by studying the challenges and solutions in detecting similar binary components among large sets of program binaries. Conceptually, one potential approach is to put features of each binary component (e.g., a function or a basic block) into a "pool" and for a given binary component, querying towards the pool. To improve the efficiency, coarse-grained *syntax-based* clustering in the pool can be launched before using the *semantics-based* comparison of IMF-SIM. As for the processing time in terms of function-level comparisons, we expect the feature generation time would stay the same with our current experiments since the number of functions in this setting does not change. The model training and prediction time would increase, but since we leverage a typical tree-structure model (i.e., Extra-Trees [28]), those procedures can be boosted by multi-cores.

VI. Related Work

In this section, we review the existing work in finding similar components in programs. In particular, we focus on similarity analysis in executable files. There also exist orthogonal approaches that aim at finding code similarity at the source code level [44], [47], [39], [40], [25], [41]. We will not discuss this type of work since analysis on binary code has its own unique challenges and applications.

Naturally, syntactic features, such as opcode (e.g., RADIFF2 [6]), instructions, and control-flow structures (e.g., BINDIFF [20], [23]) and execution traces are extracted from the binary code and used for static comparison and clustering [57], [34], [12], [23], [19]. Further studies propose to leverage the semantics information to compare binary components [27], [42]. CoP combines symbolic execution with longest common subsequence (LCS) based similarity analysis for software plagiarism detection; their work is considered robust against different compilers and optimizations [50]. Ming et al. [53] detects code similarity by equivalence checking of system call sliced segments. These semantic methods are not very scalable. David et al. [18] decompose components of an executable and use statistical reasoning over small components to establish the similarity. Their method is mostly proposed as searching vulnerable procedures in a candidate binary pool, and no evaluation is reported in terms of obfuscated code. Some recent work also extends the application scope by searching for semantics-equivalent components or similar bugs in binary code cross different architectures [56], [22], [15]. BINGO [15] lifts instructions into intermediate representations and capture the semantics-level similarity through I/O samples of function model traces. However, our evaluation shows that BINGO has relatively poor performance regarding different compilation and optimization settings.

Probably the most similar work with IMF-SIM is BLEX ("blanket Execution") proposed by Egele et al. [21]. They propose a new execution model, i.e., "Blanket Execution" to execute a part of a binary in a controlled context for similarity comparison. A typical blanket execution starts from the beginning of the target function and exhaustively executes every instruction in the function. A typical blanket execution could iterate for multiple rounds. Once an error happens, such as a memory access error, it restarts from the *so-far unexecuted instruction* to make sure each instruction is executed at least once. Program behaviors are collected during execution. Independently, Godefroid [29] proposes a similar idea (referred as "Micro Execution" in his work) for testing and bug detection in binary code. The main difference between the previous work and IMF-SIM is that, while blanket execution and micro execution break the common paradigm of program execution to greatly improve the instruction coverage, IMF-SIM leverages fuzzing and backward taint analysis to exercise a program's *normal* execution with better path coverage. In other words, feature traces collected in IMF-SIM naturally represent the *true* program behaviors, which further facilitates the longest common subsequence technique to reveal the similarity among even noisy behaviors. However, as the features do not always truly reflect the normal program behavior in blanket or micro execution, features are coarsely maintained into a set for comparison [21]. In general, IMF-SIM can better reveal the hidden similarity in challenging settings, as shown in our experimental results.

VII. Conclusion

In this paper, we present IMF-SIM, a tool that identifies similar functions in binary code. IMF-SIM uses a novel method that leverages in-memory fuzzing to execute all the functions inside a binary program and collect behavior traces to train a prediction model via machine learning. Our evaluation shows that it is effective in resilience to the challenges from different compilation settings and commonly-used obfuscation methods. The experimental results also show that IMF-SIM outperforms the state-of-the-art research and industrial standard tools like BINGO, BLEX, and BINDIFF.

Acknowledgment

We thank the ASE anonymous reviewers and Tiffany Bao for their 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 N00014-16-1-2265, N00014-16-1-2912, and N00014-17-1-2894.

REFERENCES

[1] "objdump," https://sourceware.org/binutils/docs/binutils/objdump.html, 2012.

[2] "BinDiff," https://www.zynamics.com/bindiff.html, 2014.

[3] "Bindiff release note," https://blog.zynamics.com/category/bindiff/, 2016.

[4] "Comparing randomized search and grid search for hyperparameter estimation," http://scikit-learn.org/0.17/auto_examples/model_selection/randomized_search.html, 2016.

[5] "Feature importance," http://scikit-learn.org/stable/auto_examples/ensemble/plot_forest_importances.html, 2016.

[6] "Radiff2," https://github.com/radare/radare2/wiki/Radiff2, 2016.

[7] 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*. USENIX Association, 2014.

[8] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering," in *Proceedings of the 2009 Network and Distributed System Security Symposium*, ser. NDSS '09. Internet Society, 2009.

[9] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. ACM, 2016, pp. 1032–1043.

[10] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, Oct. 2001.

[11] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, ser. S&P '08, 2008, pp. 143–157.

[12] D. Bruschi, L. Martignoni, and M. Monga, "Detecting self-mutating malware using control-flow graph matching," in *Proceedings of the Third International Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, ser. DIMVA'06. Springer-Verlag, 2006, pp. 129–143.

[13] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USENIX Association, 2008, pp. 209–224.

[14] S. Cesare and X. Yang, *Software Similarity and Classification*. Springer Science, 2012.

[15] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "BinGo: Cross-architecture cross-OS binary search," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. ACM, 2016, pp. 678–689.

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

[17] C. collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '98, 1998, pp. 184–196.

[18] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16. ACM, 2016.

[19] Y. David and E. Yahav, "Tracelet-based code search in executables," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. ACM, 2014, pp. 349–360.

[20] T. Dullien and R. Rolles, "Graph-based comparison of executable objects," *SSTIC*, vol. 5, pp. 1–3, 2005.

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

[22] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovRE: Efficient cross-architecture identification of bugs in binary code," in *Proceedings of the 2016 Network and Distributed System Security Symposium*, ser. NDSS '16. Internet Society, 2016.

[23] H. Flake, "Structural comparison of executable objects," in *Proceedings of the First International Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, ser. DIMVA'04. Springer-Verlag, 2004, pp. 161–173.

[24] J. E. Forrester and B. P. Miller, "An empirical study of the robustness of windows NT applications using random testing," in *Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4*, ser. WSS'00. USENIX Association, 2000, pp. 6–6.

[25] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. ACM, 2008, pp. 321–330.

[26] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. IEEE Computer Society, 2009, pp. 474–484.

[27] D. Gao, M. K. Reiter, and D. Song, "BinHunt: Automatically finding semantic differences in binary programs," in *Proceedings of the 10th International Conference on Information and Communications Security*, ser. ICICS '08. Springer-Verlag, 2008, pp. 238–255.

[28] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine Learning*, vol. 63, no. 1, pp. 3–42, 2006.

[29] P. Godefroid, "Micro execution," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE' 2014. ACM, 2014, pp. 539–549.

[30] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. ACM, 2005, pp. 213–223.

[31] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proceedings of the 22th Network Distributed Security Symposium*, ser. NDSS '08. Internet Society, 2008.

[32] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *Proceedings of the 22nd USENIX Conference on Security*, ser. SEC'13. USENIX Association, 2013, pp. 49–64.

[33] M. A. Hearst, S. T. Dumais, E. Osman, J. Platt, and B. Schölkopf, "Support vector machines," *IEEE Intelligent Systems and their Applications*, vol. 13, no. 4, pp. 18–28, 1998.

[34] X. Hu, T.-c. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09. ACM, 2009, pp. 611–620.

[35] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Cross-architecture binary semantics understanding via similar code comparison," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, ser. SANER 2016, 2016.

[36] J. Jang, M. Woo, and D. Brumley, "Towards automatic software lineage inference," in *Proceedings of the 22nd USENIX Conference on Security*, ser. USENIX Security'13. USENIX Association, 2013, pp. 81–96.

[37] Y. C. Jhi, X. Jia, X. Wang, S. Zhu, P. Liu, and D. Wu, "Program characterization using runtime values and its application to software plagiarism detection," *IEEE Transactions on Software Engineering*, vol. 41, no. 9, pp. 925–943, Sept 2015.

[38] Y.-C. Jhi, X. Wang, X. Jia, S. Zhu, P. Liu, and D. Wu, "Value-based program characterization and its application to software plagiarism detection," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. ACM, 2011, pp. 756–765.

[39] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. IEEE Computer Society, 2007, pp. 96–105.

[40] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ser. ISSTA '09. ACM, 2009, pp. 81–92.

[41] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. ACM, 2007, pp. 55–64.

[42] W. Jin, S. Chaki, C. Cohen, A. Gurfinkel, J. Havrilla, C. Hines, and P. Narasimhan, "Binary function clustering using semantic hashes," in *Proceedings of the 11th International Conference on Machine Learning and Applications*, ser. ICMLA'12, Dec 2012, pp. 386–391.

[43] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM: Software protection for the masses," in *Proceedings of the 1st International Workshop on Software Protection*, ser. SPRO '15. IEEE Press, 2015, pp. 3–9.

[44] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, Jul. 2002.

[45] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated software diversity," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, ser. S&P '14, 2014, pp. 276–291.

[46] C. Lattner, "Macroscopic Data Structure Analysis and Optimization," Ph.D. dissertation, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, May 2005.

[47] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: A tool for finding copy-paste and related bugs in operating system code," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. USENIX Association, 2004, pp. 20–20.

[48] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. ACM, 2005, pp. 190–200.

[49] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection," *IEEE Transactions on Software Engineering*, no. 99, pp. 1–1, January 2017.

[50] ——, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. ACM, 2014, pp. 389–400.

[51] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990.

[52] J. Ming, F. Zhang, D. Wu, P. Liu, and S. Zhu, "Deviation-based obfuscation-resilient program equivalence checking with application to software plagiarism detection," *IEEE Transactions on Reliability*, vol. 65, no. 4, pp. 1647–1664, Dec 2016.

[53] J. Ming, D. Xu, Y. Jiang, and D. Wu, "BinSim: Trace-based semantic binary diffing via system call sliced segment equivalence checking," in *Proceedings of the 26th USENIX Security Symposium*. USENIX Association, 2017, pp. 253–270.

[54] G. Myles and C. Collberg, "Detecting software theft via whole program path birthmarks," in *Proceedings of the 7th International Conference on Information Security*, ser. ISC '04, 2004.

[55] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[56] 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*, ser. S&P '15. IEEE Computer Society, 2015, pp. 709–724.

[57] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, "Detecting code clones in binary executables," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ser. ISSTA '09. ACM, 2009, pp. 117–128.

[58] S. R. Safavian and D. Landgrebe, "A survey of decision tree classifier methodology," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 21, no. 3, pp. 660–674, 1991.

[59] D. Schuler, V. Dallmeier, and C. Lindig, "A dynamic birthmark for Java," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07, 2007.

[60] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC'15. USENIX Association, 2015, pp. 611–626.

[61] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[62] A. R. Van Thuan Pham, Marcel Böhme, "Model-based whitebox fuzzing for program binaries," in *Proceedings of the 31th International Conference on Automated Software Engineering*, ser. ASE '16. ACM, 2016.

[63] S. Wang, P. Wang, and D. Wu, "Semantics-aware machine learning for function recognition in binary code," in *Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME '17)*, 2017.

[64] X. Wang, Y. C. Jhi, S. Zhu, and P. Liu, "Detecting software theft via system call based birthmarks," in *Proceedings of the 2009 Annual Computer Security Applications Conference*, 2009.

[65] F. Zhang, D. Wu, P. Liu, and S. Zhu, "Program logic based software plagiarism detection," in *Proceedings of the 25th IEEE International Symposium on Software Reliability Engineering*, 2014, pp. 66–77.