Learning Approximate Execution Semantics From Traces for Binary Function Similarity

Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray

Abstract—Detecting semantically similar binary functions – a crucial capability with broad security usages including vulnerability detection, malware analysis, and forensics – requires understanding function behaviors and intentions. This task is challenging as semantically similar functions can be compiled to run on different architectures and with diverse compiler optimizations or obfuscations. Most existing approaches match functions based on syntactic features without understanding the functions' execution semantics. We present TREX, a transfer-learning-based framework, to automate learning approximate execution semantics explicitly from functions' traces collected via forced-execution (i.e., by violating the control flow semantics) and transfer the learned knowledge to match semantically similar functions. While it is known that forced-execution traces are too imprecise to be directly used to detect semantic similarity, our key insight is that these traces can instead be used to teach an ML model approximate execution semantics of diverse instructions and their compositions. We thus design a pretraining task, which trains the model to learn approximate execution semantics from the two modalities (i.e., forced-executed code and traces) of the function. We then finetune the pretrained model to match semantically similar functions. We evaluate TREX on 1,472,066 functions from 13 popular software projects, compiled to run on 4 architectures (x86, x64, ARM, and MIPS), and with 4 optimizations (O0-O3) and 5 obfuscations. TREX outperforms the state-of-the-art solutions by 7.8%, 7.2%, and 14.3% in cross-architecture, optimization, and obfuscation function matching, respectively, while running 8× faster. Ablation studies suggest that the pretraining significantly boosts the function matching performance, underscoring the importance of learning execution semantics. Our case studies demonstrate the practical use-cases of TREX – on 180 real-world firmware images, TREX uncovers 14 vulnerabilities not disclosed by previous studies. We release the code and dataset of TREX at https://github.com/CUMLSec/trex.

Index Terms—Binary analysis, large language models, software security

1 INTRODUCTION

SEMANTIC function similarity, which quantifies the behavioral similarity between two functions, is a fundamental program analysis capability with a broad spectrum of real-world security usages, such as vulnerability detection [3], [13], exploit generation [7], tracing malware lineage [10], [27], [39], [76], software patching [44], [98], and forensics [53]. OWASP lists "using components with known vulnerabilities" as one of the top-10 security risks [68] in 2020. Therefore, identifying similar vulnerable functionsin massive software projects can save significant manual effort.

When matching semantically similar functions for security-critical applications (e.g., vulnerability discovery), we often have to deal with software at *binary level*, such as commercial off-the-shelf products (i.e., firmware images) and legacy programs. However, this task is challenging, as the functions' high-level information (e.g., data structure definitions) are removed during the compilation process. Establishing semantic similarity gets even harder when the functions are compiled to run on different architectures with various compiler optimizations or obfuscated with simple transformations.

Recently, Machine Learning (ML) approaches have shown promise in tackling these challenges [23], [57], [93] by learning features that can identify similar function binaries across different architectures, compiler optimizations, or even some types of obfuscation. Specifically, ML models learn function representations (i.e., embeddings) from function binaries and use the distance between the embeddings of two functions to compute their similarity. The smaller the distance, the more similar the functions are to each other. Such approaches have achieved state-of-the-art results, outperforming traditional methods [100] using hand-crafted signatures (e.g., number of basic blocks). Such embedding distance-based strategy is particularly appealing for large-scale function matching – taking only around 0.1 seconds searching over one million functions [30].

Execution Semantics. Despite the impressive progress, it remains challenging for these approaches to match semantically similar functions with disparate syntax and structure [58].

- *Kexin Pei, Junfeng Yang, Suman Jana, and Baishakhi Ray are with the Columbia University, New York, NY 10027 USA. E-mail: {kpei, junfeng, suman, rayb}@cs.columbia.edu.*
- *Zhou Xuan is with the Purdue University, West Lafayette, IN 47907 USA. E-mail: xuan1@purdue.edu.*

Manuscript received 5 May 2022; revised 31 October 2022; accepted 11 December 2022. Date of publication 28 December 2022; date of current version 18 April 2023.
This work was supported in part by NSF under Grants CCF-18-45893, CCF-18-22965, CCF-16-19123, CNS-18-42456, CNS-18-01426, CNS-16-18771, CNS-16-17670, CNS-15-64055, and CNS-15-63843, in part by ONR under Grants N00014-17-1-2010, N00014-16-1-2263, and N00014-17-1-2788, in part by an NSF CAREER award, in part by an ARO Young Investigator (YIP) award, in part by a Google Faculty Fellowship, in part by a JP Morgan Faculty Research Award, in part by a DiDi Faculty Research Award, in part by a Google Cloud Grant, in part by a Capital One Research Grant, and in part by an Amazon Web Services Grant.
(Corresponding author: Kexin Pei.)
Recommended for acceptance by D. Lo.
This article has supplementary downloadable material available at https://doi.org/10.1109/TSE.2022.3231621, provided by the authors.
Digital Object Identifier no. 10.1109/TSE.2022.3231621

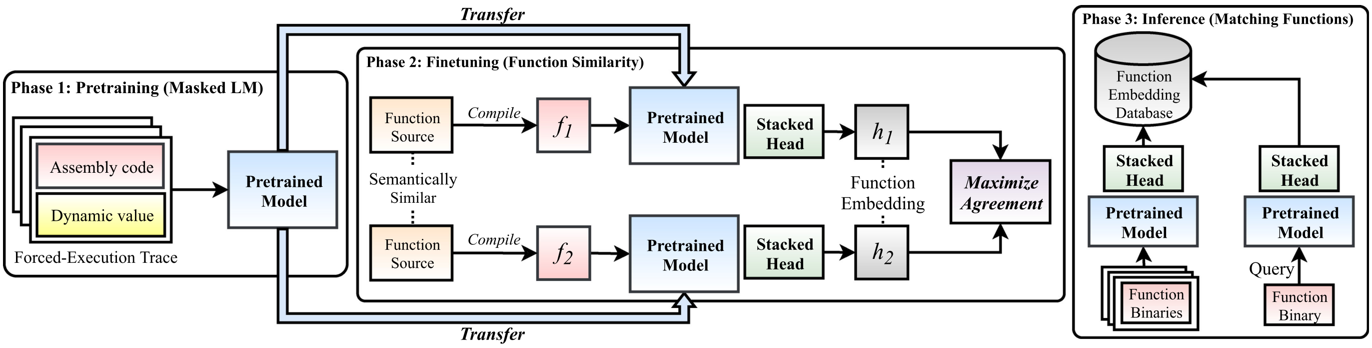


Fig. 1. TREX's workflow. We first pretrain the model on the functions' traces obtained from forced-execution, consisting of instructions and dynamic values, based on the masked LM task. We then finetune the pretrained model on the semantically similar function (only static instruction) pairs for function similarity tasks. During inference, the finetuned model computes the function embedding, whose distance encodes the function similarity.

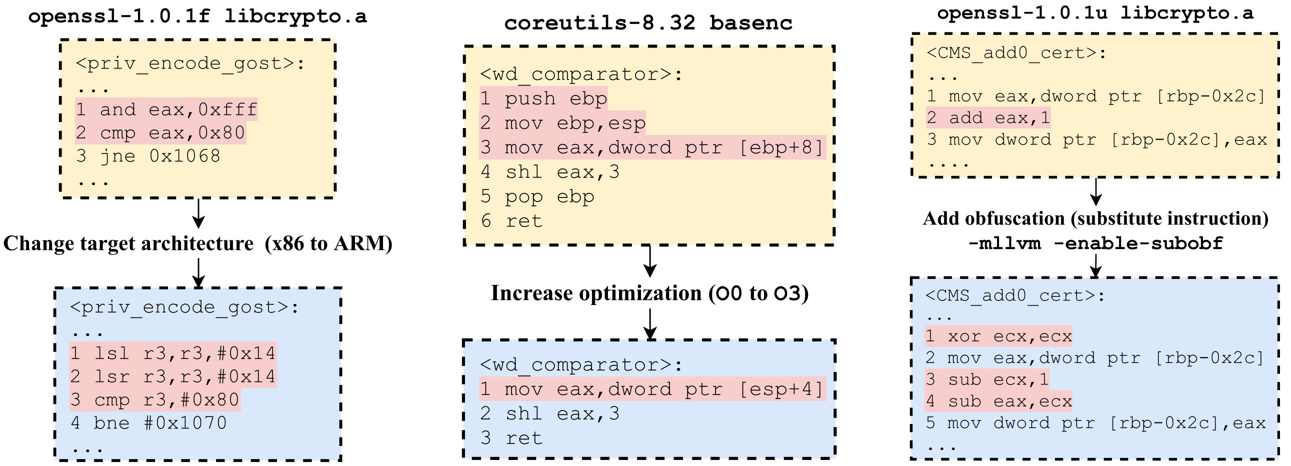
An inherent cause is that the code semantics is characterized by *its execution effects*. However, all existing learning-based approaches are *agnostic to program execution semantics*, training only on the static code. Such a setting can easily lead a model into matching simple patterns, limiting their accuracy when such spurious shortcuts are absent or changed [1], [72].

For instance, consider the following pair of x86 instructions: `mov eax,2;lea ecx,[eax+4]` are semantically equivalent to `mov eax,2;lea ecx,[eax+eax*2]`. An ML model focusing on syntactic features might pick common substrings (both sequences share the tokens `mov`, `eax`, `lea`, `ecx`) to establish their similarity, which does not encode the key reason of the semantic equivalence. Without grasping the execution semantics, an ML model can easily learn such spurious patterns without understanding the inherent cause of the equivalence: `[eax+eax*2]` computes the same exact address as `[eax+4]` when `eax` is 2.

Limitations of Existing Dynamic Approaches. Existing dynamic approaches try to avoid the above issues by directly comparing the functions' behaviors. As finding program inputs reaching the target functions is extremely challenging and time-consuming, the prior works perform *forced-execution* by initializing the function input states (e.g., registers, memory) with random values and directly executing the target functions by ignoring the control flow [25], [75]. While forced-execution improves the coverage and can thus execute all instructions within a function, the traces collected in such a way is often too noisy to be representative of function behavior, as the randomly initialized inputs might not be feasible during program execution and the control flow is violated. Therefore, when such traces are directly used to compute similarities, they lead to many false positives [23]. Worse, executing every function pair during matching is extremely hard to scale to millions of function pairs.

Limitations of ML-Based Approaches on Dynamic Traces. Recent studies have shown that incorporating traces as an additional input helps the ML model to learn a better program representation [66], [87], which improves on many downstream program analysis tasks such as type inference [73] and program repair [86]. Their key idea is that instead of executing programs, they employ ML models to learn an approximate summary of program behavior from the dynamic information and use that knowledge for the target analysis task. However, these approaches are limited when directly applied for matching functions. In order to model traces, they often resort to mimicking regular execution and therefore the modeled traces have low coverage [73]. In the context of matching functions, such partial program behaviors with limited coverage are often not representative enough to help the model learn a holistic summary of the function behavior to match their similarity. An obvious followup question is can we employ forced-execution to teach an ML model to generate high-coverage representation of binary code? Unfortunately, training on forced-execution traces is challenging. As existing ML-based approaches are often formulated in a way that takes the traces and *directly* train for the target task, the noise in the traces can significantly bias the model into learning spurious correlation between the noise and the target.

Our Approach. We present TREX, a transfer-learning-based framework, that trains ML models to learn the execution semantics from forced-execution traces. Unlike prior works, which use noisy traces to directly measure similarity or learning on regular traces with limited coverage, TREX pretrains a model on a mix of regular and forced-execution traces with a dedicated pretraining task that are less susceptible to the noise in the trace. Our key observation is that while some traces are noisy, i.e., being forced-executed and occasionally violating the control flow behavior, most parts of the traces preserve the same effects to those of regular execution within some neighboring context, e.g., straight-line code or branches where the flow of control to fall through. Therefore, we design our pretraining task to make the model to observe and learn the execution effect of individual instructions and their compositions from the local context (Section 2.2). In order to generalize to diverse traces collected from various functions, the model has to be resistant to the noise introduced from forced-execution and learn the execution semantics preserved across a mix of regular and forced-execution traces. After learning the approximate execution semantics, we finetune the pretrained model to learn to compose its learned knowledge of various instructions to match semantically similar functions (Fig. 1). As a result, during inference, we *do not* need to execute any functions on-the-fly to match them. Instead, our model only uses the function instructions, but with an *augmented* understanding of their approximate execution semantics. Importantly, such a design also saves significant runtime overhead by



(a) Cross-architecture (b) Cross-optimization (c) Cross-obfuscation

Fig. 2. Challenging cases of matching similar functions across different architectures, optimizations, and obfuscations. (**a**) Function `priv_encode_gost` is from `libcrypto.a` in `openssl-1.0.1f`. The upper function is compiled to x86 while the lower is compiled to ARM. (**b**) Function `<wd_comparator>` is from `basenc` in `coreutils-8.32`. The upper and lower function is compiled by GCC-7.5 with `-O0` and `-O3`, respectively. (**c**) Function `<CMS_add0_cert>` is from `libcrypto.a` in `openssl-1.0.1 u`. The upper function is compiled using `clang` with default options. The lower function is compiled by turning on the instruction substitution using Hikari [97], e.g., `-mllvm -enable-subobf`.

eliminating the need of performing forced execution during the matching time.

We pretrain the model on traces with a task inspired from masked language modeling (masked LM) [22]. Specifically, it randomly masks instructions and values in traces and asks the model to predict the masked parts based on those not masked in the context. Such a design forces the model to learn how individual instructions and their compositions behave in order to infer the masked parts correctly, automating learning execution semantics without manual feature engineering. To facilitate learning on traces, TREX adapts the hierarchical Transformer [73] to model long-range execution effects of instructions on trace values.

To facilitate cross-architecture function matching and learning on traces collected from different architectures, we extend the existing forced-execution algorithm [25], [75] that only works for x86 to support ARM and MIPS. As a result, we are able to train and evaluate TREX on 1,472,066 functions collected from 13 open-source software projects across 4 architectures (x86, x64, ARM, and MIPS) and compiled with 4 optimizations (O0-O3), and 5 obfuscation strategies [97]. Our experiments demonstrate that TREX outperforms the state-of-the-art systems by 7.8%, 7.2%, and 14.3% in matching functions across different architectures, optimizations, and obfuscations, respectively. Our ablation studies show that the pretraining task improves the accuracy of matching semantically similar functions by 15.7%. We also apply TREX in searching vulnerable functions in 180 real-world firmware images developed by well-known vendors and deployed in diverse embedded systems, including WLAN routers, smart cameras, and solar panels. Our case study shows that TREX helps find 14 CVEs not disclosed in previous studies.

Contributions. We make the following contributions.

- We propose a new approach to first train the model to learn program execution semantics from a mix of regular and forced-execution trace and then train the model to compose its learned knowledge to identify semantically similar functions.
- We extend forced-execution that can expose diverse function behavior to support multiple architectures for pretraining. We then develop a dedicated pretraining objective that helps the model to efficiently learn the instructions' execution semantics.
- We release our large-scale binary functions and their traces collected from a wide spectrum of open-source software projects, with diverse architectures, optimizations, and obfuscations, to foster future research in this direction.
- We demonstrate that TREX is faster and more accurate than the state-of-the-art tools in cross-architecture/optimization/obfuscation function matching, while running up to 8× faster. Moreover, TREX helps uncover new vulnerabilities in real-world firmware images not disclosed by previous studies. We open-source the code, the trained model, and the dataset of TREX at https://github.com/CUMLSec/trex.

2 OVERVIEW

We use the real-world functions as motivating examples to describe the challenges of matching semantically similar functions and how the pretraining task could address them.

2.1 Challenging Cases

We use three semantically equivalent but syntactically different real-world function pairs (Fig. 2) to illustrate the typical challenges of learning from only static code for matching similar functions.

Cross-Architecture. Consider Fig. 2a, where two functions have the same effects as they both take the lower 12-bit of a register and compare it to `0x80`. Detecting they are similarity requires understanding the execution semantics of `and` in x86 and `lsl/lsr` in ARM. It also requires understanding how the values (i.e., `0xfff` and `0x14`) in the code are manipulated. However, learning on static code without observing how each instruction behaves will fall short to teach the model how to make such an inference.

Cross-Optimization. Consider the two functions in Fig. 2b. They are semantically equivalent as `[ebp+8]` and `[esp+4]` access the same argument pushed on the stack by the caller. To detect such similarity, the model should understand `push` decreases the stack pointer `esp` by 4. The model should also notice that `mov` at line 2 assigns the decremented `esp` to `ebp` such that `ebp+8` in the upper function equals `esp+4` in the lower function. However, such information is not manifested in any static code patterns.

Cross-Obfuscation. Fig. 2c demonstrates a simple obfuscation by substituting instructions, which replaces `eax+1` with `eax-(-1)`. While both functions increment the value at stack location `[rbp-0x2c]` by 1, the upper function achieves this by loading the value to `eax`, incrementing it by 1, and writing `eax` back to stack, but the lower function takes a convoluted way by first letting `ecx` to store -1, decrementing `eax` by `ecx`, and writing `eax` to stack. Detecting the equivalence requires understanding how arithmetic operations such as `xor`, `sub`, and `add`, execute. However, static information cannot fully expose such knowledge.

2.2 Pretraining Masked LM on Traces

We describe the intuition how the pretraining task encourages the model towards learning approximate execution semantics of different instructions under different masking scenarios, and thus potentially help address the challenging cases in Fig. 2. Recall the operation of our pretraining task: given a function's trace (i.e., instructions and values), we mask some random parts and train the model to predict the masked parts using those not masked.

Masking Register. Consider masking the `eax` in line 3 in the upper function of Fig. 2c. To correctly predict its name and trace value, the model has to understand the semantics of `add` and can deduce the value of `eax` in line 3 after observing the value of `eax` in line 2 (before the addition takes the effect). Similarly, when masking the values of `ecx` in line 4 and `eax` in line 5, the model needs to learn the semantics of `xor` and `sub` to minimize the prediction losses. Such an understanding helps the model to attribute the similarity (during finetuning) based on the similar execution effects between the two functions, as opposed to their similar syntax.

Masking Opcode. Besides masking the register and its value, we allow masking the opcode of an instruction. Predicting the masked opcode requires the model to reversely infer its execution effect. Consider Fig. 2b, where we mask the `mov` in line 2 of upper function. To correctly predict the opcode, the model should learn several key aspects of the function.

First, according to its context, i.e., the value of `ebp` at line 3 and `esp` at line 2, the model needs to understand that `mov` operates as an assignment in order to predict it correctly. Other opcodes are less likely as their execution effect conflicts with the observed resulting register values, e.g., `add` will assign `ebp` with `ebp+esp`, which conflicts with the value observed at line 3. Second, the model should learn the calling conventions and basic syntax of x86 instructions, e.g., only a subset of opcodes accept the stack operands (`ebp,esp`). It can thus exclude many syntactically impossible opcodes such as `push`, `jmp`, etc. As a result, the model is able to infer `ebp` (line 3 of upper function) equals to `esp`. Assuming that the model may have also learned (from other masked samples) `push` decrements stack pointer `esp` by 4 bytes, now when such a pretrained model is finetuned to match the two functions, it is more likely to learn that the similarity is due to that `[ebp+8]` in the upper function accesses the same address pointed by `[esp+4]` in the lower function.

Function	f ::=	$[i_1, i_2, i_3, ...]$
Instruction	i ::=	r:=e | nop | call(e) | jmp(e_c, e_a) | ret | store(e_v, e_a) | r:=load(e)
Expr	e ::=	c | r | r_1 *op* r_2
Operator	*op* ::=	{+, -, *, /, >, <, ...}
Register	r ::=	{pc, sp, eax, r0, $a0, ...}
Const	c ::=	{true, false, 0x0, 0x1, ...}

Fig. 3. Low-level IR for representing assembly code. The IR abstracts away the disparate syntax across multiple architectures.

Other Masking Strategies. We are not constrained by the number or the type of tokens (e.g., operand, opcode, values, etc.) in the code and trace to mask, i.e., we can mask multiple tokens in one or more instructions and also multiple trace values. During training, the masking operation selects a random subset of code blocks and trace values at *each training iteration and training samples*. Such a random masking strategy enables the model to learn execution effect of diverse instructions and their compositions.

How Pretraining on Noisy Traces Helps Match Similarity. While the examples in Fig. 2 are straight-line code that their execution will not introduce noisy traces, they can still be forced-executed if triggering them requires violating certain control flow constraints (i.e., predicates in the branch conditions). However, even though such traces might contain infeasible values, learning from such noisy traces can still be useful. As the above examples show, predicting the masked code and trace values requires the model to make *local* inference based on its understanding of the neighboring instructions. Thus, noisy forced-execution traces can still encode meaningful local behavior that requires the model to learn their approximate execution semantics. During finetuning, the model is further trained to compose its understanding of various instructions' execution effect and expected to more likely attribute the function similarity to their similar behavior instead of their syntax.

3 Methodology

This section elaborates on TREX's design, including the forced-execution algorithms, the architecture, and the training workflow.

3.1 Forced-Execution

IR Language. We extend forced-execution [75] to handle x64, ARM, and MIPS, where the original paper only describes x86 as the use case. We introduce a low-level intermediate representation (IR) to abstract away the complexity of different architectures' syntax (Fig. 3). The IR here only serves to facilitate the discussion of the forced-execution algorithm. In our implementation, we use real assembly instructions as model's input (Section 3.2).

We denote memory reads and writes by `load(`e`)` and `store(`e_v, e_a`)` (i.e., store the value e_v to address e_a), which generalize to both the load-store (i.e., ARM, MIPS) and register-memory architecture (i.e., x86). Both operations can take as input e – an expression that can be an explicit hexadecimal number (denoting the address or a constant), a register, or a result of an operation on two registers. We use `jmp` to denote the jump instruction including both direct and indirect jump (i.e., the expression e_a can be a constant c or a register r). The first parameter in `jmp` is the conditional expression e_c and it evaluates to `true` for unconditional jump. We represent function invocations and returns by `call` and `ret`, where `call` is parameterized by an expression, which can be a constant (direct call) or a register (indirect call).

Algorithm. Algorithm 1 outlines the steps to forced-execute a function f. First, it initializes the memory and all registers except the special-purpose register, such as the stack pointer and the program counter. It then linearly executes instructions of f. We map the memory *on-demand* when the instruction attempts to access them. If the instruction reads from memory, we further initialize a random value in the mapped memory addresses. We skip call/jump instructions following the forced execution strategy [75]. Forced-execution terminates when it finishes executing all instructions, reaches `ret`, or times out. Note that for straight-line programs or when the initialized inputs happen to lead all the condition-checks to false, we obtain a regular (not forced) execution trace.

Algorithm 1. Forced-Execute a Function f

Input: Function binary f. All registers r.
Output: Forced-execution trace t.

$\mathbb{I} \leftarrow get\_instructions(f)$ ▷ put all instructions in f into a queue
$t \leftarrow empty\_vector$
`sp` $\leftarrow init\_stack\_pointer\_addr()$ ▷ stack pointer address
`pc` $\leftarrow init\_program\_counter\_addr()$ ▷ first instruction's address
`sm` $\leftarrow mem\_map($`sp`, `STACK_SIZE`$)$ ▷ initialize stack memory
`cm` $\leftarrow mem\_map($`pc`$, |\mathbb{I}|)$ ▷ initialize memory for code
for each register r_i in $r \backslash \{$`sp`,`pc`$\}$ **do**
    $r_i \leftarrow$ `random_init()` ▷ initialize register values
while $\mathbb{I} \neq \emptyset$ **do**
    $i \leftarrow dequeue(\mathbb{I})$
    if $i.type$ = `load` **or** $i.type$ = `store` **then** ▷ memory access
        $mem\_map(i.access\_addr, i.access\_size)$
        if $i.type$ = `load` **then**
            $write\_random(i.access\_addr)$
        $t \leftarrow t \cup execute(i)$
    else if $i.type$ = `jmp` **or** $i.type$ = `call` **or** $i.type$ = `nop` **then**
        continue ▷ skip control transfer
    else ▷ all other instructions
        $t \leftarrow t \cup execute(i)$

3.2 Input Representation

Given a function f and its trace t, we prepare the model input x, consisting of 5 aligned sequences with the same size n. Fig. 4 shows the example of TREX's input and output and how the input tokens are embedded using different strategies. We follow StateFormer's [73] approach for tokenizing inputs so we only briefly describe each sequence below for completeness.

Code. The first sequence x_f is the assembly code sequence: $x_f = \{mov, eax, +, \ldots\}^n$, generated by tokenizing all assembly instructions. Note that unlike StateFormer, where their code sequences come from complete static code of a function, here x_f are instructions *along one forced-executed path* in a function. We move all numeric values to the value sequence (see below) and replace them with a special token `num`. With all these preprocessing steps, the vocabulary size of x_f across all architectures is 3,300.

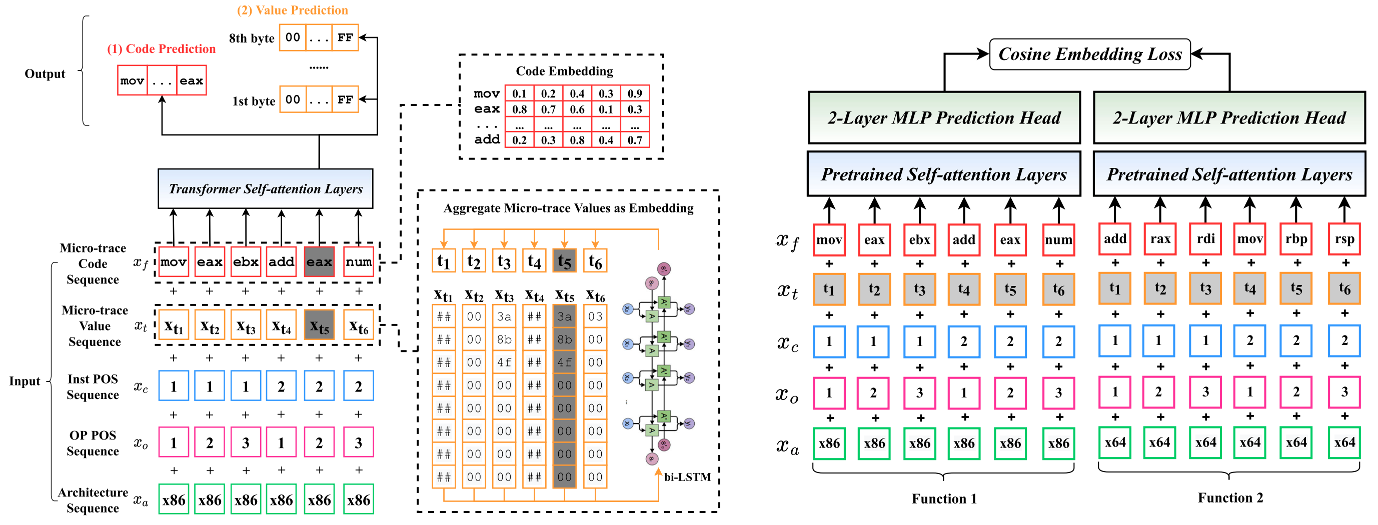
Value. The second sequence x_t is the trace value sequence. As discussed in Section 2, we keep *explicit* numerical values in x_t, which denote the value for each token (e.g., register) in an instruction before it is executed. For example, in `mov eax,0x8; mov eax,0x3`, the trace value of the second `eax` is `0x8`. For code token without dynamic value, we use dummy values (see how we encode trace values in the following).

Auxiliary Sequences. There are 3 additional sequences to encode some structural and syntactic hints: the instruction positions x_c, opcode/operand positions x_o, and the architecture sequence x_a. x_c is a sequence of integers encoding the position of each instruction. All opcodes/operands within a single instruction share the same value. x_o is a sequence of integers encoding the position of each opcode and operands within a single instruction. x_a specifies which instruction set architecture that the trace belongs to: x_a = {x86, x64, ARM, MIPS}^{n}.

Encoding Trace Values. As numerical values can lead to prohibitively large vocabulary (2^{64} possible values on a 64-bit machine), we follow StateFormer's hierarchical encoding to address this challenge. Let x_{t_i} denote the ith value in x_t, we represent x_{t_i} as an (padded) 8-byte fixed-length byte sequence x_{t_i} ={`0x00`,..., `0xff`}^{8} ordered in Big-Endian. Unlike StateFormer that uses a neural arithmetic unit (NAU) that treats each byte independently, we employ a bidirectional LSTM (bi-LSTM) that takes x_{t_i} as input and use its last hidden cell's output as the value representation $t_i = bi\text{-}LSTM(x_{t_i})$. As a recurrent network, bi-LSTM is more amenable to learn the dependencies between high and low bytes within a single value. To make the micro-trace code tokens without dynamic values (e.g., opcode) align with the byte sequence, we use a dummy sequence (`##`) with the same length. Fig. 4a shows how bi-LSTM takes the byte sequence and computes the embedding.

3.3 Pretraining With Traces

Input Embeddings. We embed each token in the 5 sequences with the same embedding dimension d_{emb}. Specifically, let $E_f(x_f)$, $E_t(x_t)$, $E_c(x_c)$, $E_o(x_o)$, $E_a(x_a)$ denote applying the embedding to the tokens in each sequence, respectively. We have the embedding of x_i: $E_i = E_f(x_{f_i}) + E_t(x_{t_i}) + E_c(x_{c_i}) + E_o(x_{o_i}) + E_a(x_{a_i})$. Here x_{f_i} denotes the ith token in x_f, where other sequences follow the similar notation. Note that $E_t(x_{t_i})$ is the output from bi-LSTM (Section 3.2) while the others are simply one-hot encoded with an embedding matrix. Fig. 4a illustrates the two embedding strategies.



(a) Pretraining for masked code and trace value prediction (b) Finetuning for binary similarity

Fig. 4. (Left) Pretraining input-output examples and model architecture. (Right) Finetuning input-output examples. We mark the value sequence x_t as grey to indicate they are dummy values (Section 3.4), i.e., we statically compare two functions' similarity. The border colors of the box are made consistent across sub-figures to indicate the same type of input sequences.

Masked LM. To pretrain the model with the masked LM objective, we mask the code token and value token in x_f and x_t, respectively, and replace them with a special token <MASK>. Let $m(E_i)$ denote the embedding of the masked x_i and $\mathbb{MP}$ a set of positions on which the masks are applied. The model g_p (to be pretrained) takes as input a sequence of embeddings with random tokens masked: $(E_1, \ldots, m(E_i), \ldots, E_n), i \in \mathbb{MP}$, and predicts the code and the values of the masked tokens: $\{\hat{x}_{f_i}, \hat{v}_i | i \in \mathbb{MP}\} = g_p(E_1, \ldots, m(E_i), \ldots, E_n)$. Let g_p be parameterized by θ, the objective of training g_p is to search for θ that minimizes the cross-entropy losses between (1) the predicted masked code and the actual code, and (2) the predicted masked values (8 bytes) and the actual values (Fig. 4a).

$$\underset{\theta}{\operatorname{argmin}} \sum_{i=1}^{|\mathbb{MP}|} (-x_{f_i} \log(\hat{x}_{f_i}) + \alpha \sum_{j=1}^{8} -x_{t_{ij}} \log(\hat{x}_{t_{ij}})), \quad (1)$$

$\hat{x}_{t_{ij}}$ denotes the predicted jth byte of x_{t_i} (the ith token in x_t). α is a hyperparameter that weighs the cross-entropy losses between predicting code tokens and predicting values.

Contextualized Embeddings. We employ the self-attention layers [84] to endow contextual information to each embedding E_i, which encodes the context-sensitive meaning of each token (e.g., `eax` in `mov eax,ebx` has different embedding with that in `jmp eax`). This is in contrast with static embeddings commonly used in the prior works [23], [24], where a code token is assigned to a fixed embedding regardless of the changed context. Given k self-attention layers, let $E_{k,i}$ denote the learned embeddings after the last layer. $E_{k,i}$ will be used to predict the masked code in pretraining and match similar functions in finetuning (see following).

3.4 Finetuning for Function Similarity

After the model is pretrained, we finetune the model to predict function similarity. Given a function pair, we feed each function's *static code* (instead of the traces that only cover one path as described in Section 3.2) to the pretrained model g_p and obtain the pair of embedding sequences produced by the last self-attention layer of g_p: $E_k^{(1)} = (E_{k,1}^{(1)}, \ldots, E_{k,n}^{(1)})$ and $E_k^{(2)} = (E_{k,1}^{(2)}, \ldots, E_{k,n}^{(2)})$ where $E_k^{(1)}$ and $E_k^{(2)}$ correspond to the first and second function, respectively. Let $y = \{-1, 1\}$ be the ground-truth indicating the similarity between two functions. We stack a 2-layer Multi-layer Perceptrons (MLP) g_t, taking as input the mean pooling of all embeddings within each function, and producing a function embedding

$$g_t(E_k) = tanh((\sum_{i=1}^{n} E_{k,i})/n) \cdot W_1) \cdot W_2.$$

Here $W_1 \in \mathbb{R}^{d_{emb} \times d_{emb}}$ and $W_2 \in \mathbb{R}^{d_{emb} \times d_{func}}$ transforms the mean-pooled E_k with d_{emb} dimensions into the function embedding with d_{func} dimensions. Let g_t be parameterized by θ, the finetuning objective minimizes the cosine embedding loss l_{ce} between the ground-truth and the cosine distance between two function embeddings (Fig. 4b): $\operatorname{argmin}_\theta\ l_{ce}(g_t(E_k^{(1)}), g_t(E_k^{(2)}), y)$, where

$$l_{ce}(x_1, x_2, y) = \begin{cases} 1 - cos(x_1, x_2) & y = 1 \\ max(0, cos(x_1, x_2) - \xi) & y = -1 \end{cases}, \quad (2)$$

ξ is the margin chosen between 0 and 0.5 [70]. As both g_p and g_t are differentiable, optimizing Equations (1) and (2) can be guided by gradient descent via backpropagation. After finetuning g_t and g_p, we compute the function embedding $f_{emb} = g_t(g_p(f))$ and the function similarity is measured by the cosine similarity between their embedding vectors: $cos(f_{emb}^{(1)}, f_{emb}^{(2)})$.

4 Implementation and Setup

We implement Trex using fairseq [67] based on PyTorch 1.6.0 with CUDA 10.2 and CUDNN 7.6.5. We run all experiments on a Linux server running Ubuntu 18.04, with an Intel Xeon 6230 at 2.10 GHz with 80 virtual cores including

TABLE 1
Number of Functions for Each Project Across 4 Architectures With 4 Optimization Levels and 5 Obfuscations

ARCH	OPT OBF	# Functions Binutils	Coreutils	Curl	Diffutils	Findutils	GMP	ImageMagick	microhttpd	TomCrypt	OpenSSL	PuTTy	SQLite	Zlib	Total
ARM	`O0`	25,492	19,992	1,067	944	1,529	766	2,938	200	779	11,918	7,087	2,283	157	75,152
	`O1`	20,043	14,918	771	694	1,128	704	2,341	176	745	10,991	5,765	1,614	143	60,033
	`O2`	19,493	14,778	765	693	1,108	701	2,358	171	745	11,001	5,756	1,473	138	59,180
	`O3`	17,814	13,931	697	627	983	680	2,294	160	726	10,633	5,458	1,278	125	55,406
	Total # Functions of ARM														**249,771**
MIPS	`O0`	28,460	18,843	1,042	906	1,463	734	2,840	200	779	11,866	7,003	2,199	153	76,488
	`O1`	22,530	13,771	746	653	1,059	670	2,243	176	745	10,940	5,685	1,530	139	60,887
	`O2`	22,004	13,647	741	653	1,039	667	2,260	171	743	10,952	5,677	1,392	135	60,081
	`O3`	20,289	12,720	673	584	917	646	2,198	161	724	10,581	5,376	1,197	121	56,187
	Total # Functions of MIPS														**253,643**
x86	`O0`	37,783	24,383	1,335	1,189	1,884	809	3,876	326	818	12,552	7,548	2,923	204	95,630
	`O1`	32,263	20,079	1,013	967	1,516	741	3,482	280	782	11,578	6,171	2,248	196	81,316
	`O2`	32,797	21,082	1,054	1,006	1,524	728	3,560	265	784	11,721	6,171	2,113	183	82,988
	`O3`	34,055	22,482	1,020	1,052	1,445	707	3,597	284	760	11,771	5,892	1,930	197	85,192
	Total # Functions of x86														**358,261**
x64	`O0`	26,757	17,238	1,034	845	1,386	751	2,970	200	782	12,047	7,061	2,190	151	73,412
	`O1`	21,447	12,532	739	600	1,000	691	2,358	176	745	11,120	5,728	1,523	137	58,796
	`O2`	20,992	12,206	734	596	976	689	2,374	171	742	11,136	5,703	1,380	132	57,831
	`O3`	19,491	11,488	662	536	857	667	2,308	160	725	10,768	5,390	1,183	119	54,354
	Total # Functions of x64														**244,393**
x64	`bcf`	27,734	17,093	998	840	1,388	746	2,833	200	782	10,768	7,069	2,183	151	72,785
	`cff`	27,734	17,093	998	840	1,388	746	2,833	200	782	10,903	7,069	2,183	151	72,920
	`ibr`	27,734	17,105	998	842	1,392	746	2,833	204	782	12,045	7,069	2,183	151	74,084
	`spl`	27,734	17,093	998	840	1,388	746	2,833	200	782	10,772	7,069	2,183	151	72,789
	`sub`	27,734	17,093	998	840	1,388	746	2,833	200	782	11,403	7,069	2,183	151	73,420
	Total # Obfuscated Functions														**365,998**
Total # Functions															**1,472,066**

The functions with the same name in different version of projects or in different projects are considered as different functions.

hyperthreading, 385 GB RAM, and 8 Nvidia RTX 2080-Ti GPUs.

Datasets. To train and evaluate TREX, we collect 13 popular open-source software projects (Table 1). We compile these projects into 4 architectures, i.e., x86, x64, ARM, and MIPS, with 4 optimization levels, i.e., `O0`-`O3`, using `GCC-7.5`. We also obfuscate all projects using 5 types of obfuscations by Hikari [97] on x64. The obfuscations include bogus control flow (`bcf`), control flow flattening (`cff`), register-based indirect branching (`ibr`), basic block splitting (`spl`), and instruction substitution (`sub`). We turn off the compiler optimization in case it optimizes away the obfuscated code. As we encounter several errors in cross-compilation using Hikari (based on Clang) [97], and the baseline system (i.e., Asm2Vec [23]) to which we compare only evaluates on x64, we restrict the obfuscated binaries for x64 only. As a result, we have 1,472,066 functions.

Forced-Execution. We implement forced-execution by Unicorn [79]. We forced-execute each function 3 times with different initialized registers and memory, generating 3 traces for each function in pretraining. We leverage multi-processing to parallelize forced-executing each function and set 30 seconds as the timeout in case any instruction gets stuck (i.e., infinite loops).

Baselines. For comparing cross-architecture performance, we consider 2 state-of-the-art baselines. The first one is SAFE [57]. As SAFE's model is publicly available, we run their trained models on our collected binaries. We also compare TREX with SAFE's reported results on their dataset, i.e., OpenSSL-1.0.1f and OpenSSL-1.0.1 u. The second baseline is Gemini [93]. As Gemini's trained model is not available, we use their reported numbers directly on their evaluated dataset, i.e., OpenSSL-1.0.1f and OpenSSL-1.0.1 u.

For cross-optimization/obfuscation comparison, we consider Asm2Vec [23] and Blex [25] as the baselines. Asm2Vec achieves the state-of-the-art cross-optimization/obfuscation results, based on learned embeddings from static assembly code. Blex, on the other hand, leverages functions' dynamic behavior to match function binaries. As we only find a third-party implementation of Asm2Vec that achieves extremely low Precision@1 (the metric used in Asm2Vec) from our testing (e.g., 0.02 versus their reported 0.814), and we have contacted the authors and do not get replies, we directly compare to their reported numbers. Blex is not publicly available either, so we also compare to their reported numbers directly.

Metrics. As the cosine similarity between two function embeddings can be an arbitrary real value between -1 and 1, we consider the receiver operating characteristic (ROC) curve, which measures the model's false positives/true positives under different thresholds. Notably, we use the area under curve (AUC) of the ROC curve to quantify the accuracy of TREX to facilitate benchmarking – the higher the AUC score, the better the model's accuracy. Certain baselines do not use AUC score to evaluate their system. For example, Asm2Vec uses Precision at Position 1 (Precision@1), and Blex uses the number of matched functions as the metric. Therefore, we also include these metrics to evaluate TREX when needed.

Training Setup. To separate the functions in pretraining, finetuning, and testing, we pretrain TREX on all functions in the dataset except the project to be finetuned and evaluated. Note that pretraining *is agnostic to any ground-truth indicating similar functions*. Therefore, we can in theory pretrain on large-scale codebases, which can include the functions for finetuning [22]. It is thus worth noting that our setup of separating functions for pretraining and finetuning makes our testing more challenging. For finetuning, we choose 50,000 random function pairs for each project and select random

80% for training, and the remaining is used as the testing set.

Hyperparameters. We pretrain and finetune the models for 10 epochs and 30 epochs, respectively. We choose $\alpha = 0.125$ in Equation (1) such that the cross-entropy loss of code prediction and value prediction have the same weight. We pick $\xi = 0.1$ in Equation (2) to make the model slightly inclined to treat functions as dissimilar because functions in practice are mostly dissimilar. We use 12 self-attention layers with each having 12 self-attention heads. We fix the largest input length to be 512 and split the functions longer than this length into subsequences for pretraining. The complete description of the hyperparameters can be found in our supplementary material, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2022.3231621.

5 Evaluation

Our evaluation aims to answer the following questions.

- RQ1: How accurate is Trex in matching functions across different architectures, optimizations, and obfuscations?
- RQ2: How does Trex compare to the state-of-the-art?
- RQ3: How fast is Trex compared to other tools?
- RQ4: How much does pretraining on forced-execution traces help improve the accuracy of matching functions?

5.1 RQ1: Accuracy

We evaluate how accurate Trex is in matching similar functions across different architectures, optimizations, and obfuscations. We prepare function pairs for each project with 5 types of partitions. (1) ARCH: the function pairs have *different architectures* but same optimizations. (2) OPT: the function pairs have *different optimizations* but same architectures. (3) OBF: the function pairs have *different obfuscations* with same architectures (x64). (4) ARCH+OPT: the function pairs have *both different architectures and optimizations*. (5) ARCH+OPT +OBF: the function pairs can have arbitrary architectures, optimizations, and obfuscations.

Table 2 reports the testing AUC scores and standard deviation with 3 runs of Trex. On average, Trex achieves > 0.958 (and up to 0.995) AUC scores, even in the most challenging setting where the functions can come from different architectures, optimizations, and obfuscations at the same time. We note that Trex performs the best on cross-optimization matching. This is intuitive as the syntax of two functions from different optimizations are not changed significantly (e.g., the name of opcode, operands remain the same). Nevertheless, we find the AUC scores for matching functions from different architectures is only 0.001 lower, which indicates the model is robust to entirely different syntax between two architectures.

TABLE 2
Trex Results (in AUC Score and Its Standard Deviation) on Function Pairs Across Architectures, Optimizations, and Obfuscations

	Cross-				
	ARCH	OPT	OBF	ARCH+ OPT	ARCH+ OPT+ OBF
Binutils	.991	.992	.988	.959	.947
Coreutils	.988	.99	.989	.955	.945
Curl	.991	.993	.99	.967	.956
Diffutils	.989	.992	.99	.973	.961
Findutils	.99	.991	.99	.966	.962
GMP	.99	.989	.989	.967	.964
ImageMagick	.992	.994	.987	.957	.95
microhttpd	.991	.994	.99	.97	.965
TomCrypt	.989	.991	.99	0.971	.97
OpenSSL	.991	.99	.989	.967	.957
PuTTy	.989	.992	.99	.965	.95
SQLite	.99	.992	.99	.967	.959
Zlib	.989	.992	.987	.968	.961
Average	.99	.992	.989	.966	.957

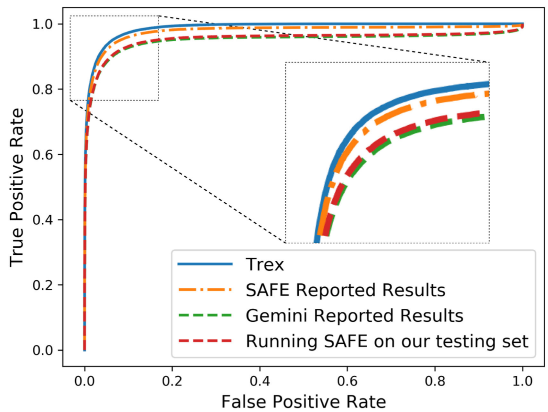


Fig. 5. ROCs of matching functions across different architectures.

5.2 RQ2: Baseline Comparison

Cross-Architecture. As described in Section 4, we first compare Trex with SAFE and Gemini on OpenSSL-1.0.1f and OpenSSL-1.0.1 u with their reported numbers (as they only evaluated on these two projects). We then run SAFE's released model on our dataset.

Fig. 5 shows that Trex's AUC score is higher than those reported in SAFE and Gemini. While SAFE's AUC score is close to Trex's, it drops to 0.976 when run our testing set – possibly because the distribution shift between different testing set [95]. For example, Fig. 6 shows that Trex consistently outperforms SAFE on our dataset, i.e., by 7.3% on average. As SAFE is only trained on OpenSSL, we also train Trex on the same dataset.

Inspired by Arp et al. [6], we study the distribution shift by measuring the KL-divergence [49] between SAFE's dataset and ours. We find the KL-divergence is 0.02, which is significant to indicate the distribution shift. Therefore, this observation demonstrates the generalizability of Trex– when pretrained to approximately learn execution semantics explicitly, it can quickly generalize to match unseen functions.

Cross-Optimization. We compare Trex with Asm2Vec and BLEX on matching functions compiled by different optimizations. As both Asm2vec and Blex run on single architecture, we restrict the comparison on x64. Besides, since Asm2Vec uses Precision@1 and Blex uses accuracy as the metric (Section 4), we compare with each tool separately using their metrics and on their evaluated dataset.

Table 3 shows Trex outperforms Asm2Vec in Precision@1 (by 7.2% on average) on functions compiled by different

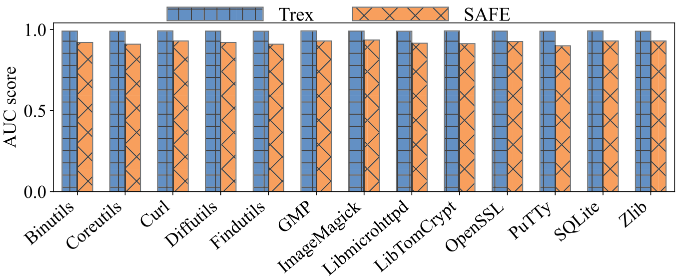


Fig. 6. Comparison between TREX and SAFE on matching functions compiled to different architectures.

TABLE 3
Comparison Between TREX and Asm2Vec (In Precision@1) on Function Pairs Across Optimizations

	Cross Compiler Optimization			
	O2 and O3		O0 and O3	
	TREX	Asm2Vec	TREX	Asm2Vec
Coreutils	**0.955**	0.929	**0.913**	0.781
Curl	**0.961**	0.951	**0.894**	0.850
GMP	**0.974**	0.973	**0.886**	0.763
ImageMagick	**0.971**	0.971	**0.891**	0.837
LibTomCrypt	**0.991**	0.991	**0.923**	0.921
OpenSSL	**0.982**	0.931	**0.914**	0.792
PuTTy	**0.956**	0.891	**0.926**	0.788
SQLite	**0.931**	0.926	**0.911**	0.776
Zlib	**0.890**	0.885	**0.902**	0.722
Average	**0.957**	0.939	**0.907**	0.803

optimizations (i.e., between O2 and O3 and between O0 and O3). As the syntactic difference between O0 and O3 is more significant than that between O2 and O3, both tools' AUC scores decrease (5% drop for TREX and 14% for Asm2Vec), but TREX's AUC score drops much less than that of Asm2Vec.

To compare to Blex, we evaluate TREX on Coreutils between optimizations O0 and O3, where they report to achieve better performance than BinDiff [100]. As Blex shows the matched functions of each program in a barchart without including the numbers of matched functions, we estimate their matched functions using their reported average percentage, i.e., 75%.

Fig. 7 shows that TREX consistently outperforms Blex in number of matched functions in all utility programs of Coreutils. Note that Blex also executes the function and uses the dynamic features to match binaries. The observation here thus implies that the learned execution semantics from TREX is more effective than the hand-coded features in Blex for matching similar binaries.

Cross-Obfuscation. We compare TREX to Asm2Vec on matching obfuscated function binaries. Asm2Vec is evaluated on obfuscations including bcf, cff, and sub – a subset of our evaluated obfuscations. As Asm2Vec only evaluates on 4 projects, i.e., GMP, ImageMagic, LibTomCrypt, and OpenSSL, we focus on the same ones, and Table 2 shows the TREX's results on other projects.

Table 4 shows TREX achieves better Precision@1 score (by 14.3% on average) throughout different obfuscations. Importantly, the last two rows show when multiple obfuscations are combined, TREX performance is not dropping as significant as Asm2Vec. It also shows TREX remains robust under varying obfuscations with different difficulties. For

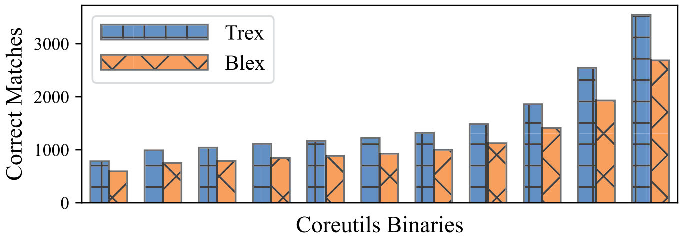


Fig. 7. Cross-optimization matching between O0 and O3 in Coreutils by TREX and Blex. We sort the 109 binaries by their function count, and aggregate the matched functions every 10 utilities.

TABLE 4
Comparison Between TREX and Asm2Vec (In Precision@1) on Function Pairs Across Different Obfuscations

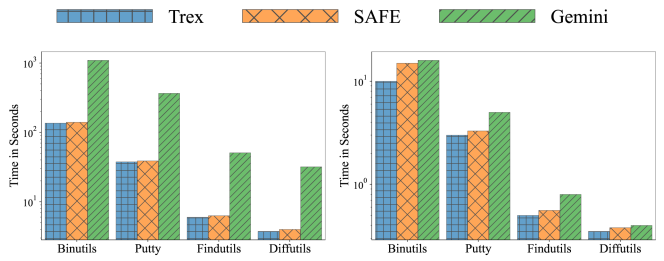
		GMP	LibTomCrypt	ImageMagic	OpenSSL	Average
bcf	TREX	**0.926**	**0.938**	**0.934**	**0.898**	**0.924**
	Asm2Vec	0.802	0.920	0.933	0.883	0.885
ccf	TREX	**0.943**	**0.931**	**0.936**	**0.940**	**0.930**
	Asm2Vec	0.772	0.920	0.890	0.795	0.844
sub	TREX	**0.949**	**0.962**	**0.981**	**0.980**	**0.968**
	Asm2Vec	0.940	0.960	0.981	0.961	0.961
All	TREX	**0.911**	**0.938**	**0.960**	**0.912**	**0.930**
	Asm2Vec	0.854	0.880	0.830	0.690	0.814

example, instruction substitution simply replaces a limited instructions (i.e., arithmetic operations) while control flow flattening dramatically changes the function code. Asm2Vec has 12.2% decrease when the obfuscation is changed from sub to ccf, while TREX only decreases by 4%.

5.3 RQ3: Execution Time

We evaluate the speed of generating function embeddings for computing similarity. We compare TREX with SAFE and Gemini on generating functions in 4 projects, i.e., Binutils, Putty, Findutils, and Diffutils, which have disparate total number of functions (see supplementary material), available online. This tests how TREX scales to different number of functions. Since the offline training (i.e., pretraining TREX) of all the learning-based tools is a one-time cost, it can be amortized in the function matching process so we do not explicitly measure the pretraining time. Moreover, the output of all tools are embedding vectors, which can be indexed and efficiently searched using locality sensitive hashing (LSH) [33]. Therefore, we do not compare the matching time of function embeddings as it simply depends on the underlying LSH implementation. Particularly, we compare the runtime of two procedures in matching functions. (1) Function parsing, which transforms the function binaries into the format that the model needs. (2) Embedding generation, which computes the embedding for the parsed function binary. We test the embedding generation using our GPU (see Section 4).

Fig. 8 shows that TREX is more efficient than the other tools in both function parsing and embedding generation for projects with different number of functions. Gemini requires manually constructing control flow graph and extracting inter-/intra-basic-block features. It thus incurs the largest overhead. For generating function embeddings, our underlying network architectures leverage the self-attention layers, which is more amenable to parallelization



(a) Function parsing (b) Embedding generation

Fig. 8. Runtime (log-scaled) performance of TREX, SAFE, and Gemini on (a) function parsing and (b) embedding generation.

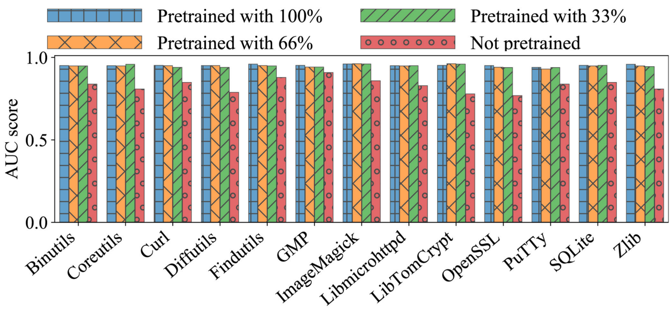


Fig. 9. Comparison of testing AUC scores between models pretrained with different fraction of the pretraining set.

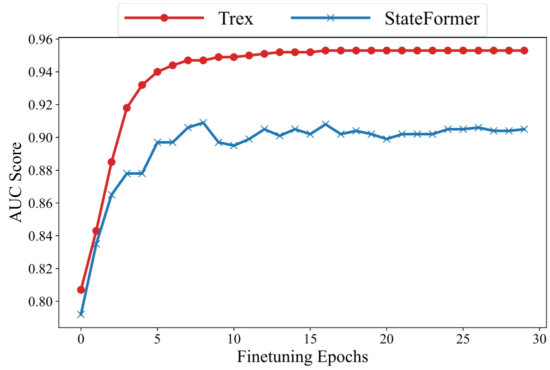


Fig. 10. Comparison between TREX and StateFormer, where StateFormer does not perform forced execution and omits predicting code tokens during pretraining.

with GPU than the recurrent counterpart (used by SAFE) and graph neural network (used by Gemini) [84]. As a result, TREX runs up to 8× faster than SAFE and Gemini.

5.4 RQ4: Ablation Study

In this section, we perform extensive ablation studies to show the effectiveness of various design in TREX. We also compare to existing baselines We first quantify how pretraining helps in matching function binaries. We then evaluate how TREX's pretraining strategy, i.e., predicting both code and trace values on the forced-executed traces, compares to the that on regular traces. We leave other ablations such as the effectiveness of including traces in pretraining and the contribution of each auxiliary field (Section 3.2) to the supplementary material, available online.

Pretraining Effectiveness. We compare the testing AUC scores achieved by TREX (1) with pretraining (except the target project that will be finetuned), (2) with 66% of pretraining functions in (1), (3) with 33% of pretraining functions in (1), and (4) without pretraining (the function embedding is computed by randomly-initialized model not pretrained). The function pairs can come from arbitrary architectures, optimizations, and obfuscations.

Fig. 9 shows that the model's AUC score drops significantly (on average 15.7%) when the model is not pretrained. Interestingly, we observe that the finetuned models achieve similar AUC scores, i.e., with only 1% decrease when pretrained with just 33% of the functions. This is likely that 33% of the pretraining set still has around 400 k functions for pretraining. Therefore, such a pretraining set can still be large enough to achieve a decent finetuning performance. To test this hypothesis, we further reduce the number of pretraining set by 10x, using 40 k samples. We find the finetuning performance drops by 11.6% on OpenSSL, getting much closer to the drop (15.7%) when the model is not pretrained. This implies pretraining on large-scale dataset is necessary to effectively boost the finetuning performance.

Comparison to StateFormer. To empirically evaluate which pretraining strategy (between TREX and StateFormer) is better suited for matching similar functions, we take the pretrained model from StateFormer, which is pretrained on regular execution traces with control/data-flow prediction as the pretraining objective, and finetune on the same set of functions. Fig. 10 shows the testing AUC scores in matching similar functions across different architectures, optimizations, and obfuscations, at each finetuning epoch. We observe that TREX outperforms StateFormer by 5.3% in AUC score and is more stable during finetuning.

Pretraining w/o Traces. The above experiment studies TREX's *finetuning* performance when excluding each of the input sequences. In this section, we also study whether including the trace values in *pretraining* can help the model to learn better execution semantics than learning from only static assembly code, which in turn results in better function matching accuracy. Specifically, we pretrain the model on the data that contains only dummy value sequence (see Section 3), and follow the same experiment setting as described above. Besides replacing the input value sequence as dummy value, we accordingly remove the prediction of dynamic values in the pretraining objective (Equation (1)). Fig. 11 shows that the AUC scores decrease by 7.2% when the model is pretrained without traces (and even 0.035 lower than that of SAFE). However, the model still performs reasonably well, achieving 0.88 AUC scores even when the functions can come from arbitrary architectures, optimizations, and obfuscations. Moreover, we observe that pretraining without traces has less performance drop than the model simply not pretrained (7.2% versus 15.7%). This demonstrates that even pretraining with only static assembly code is indeed helpful to improve matching functions. One possible interpretation is that similar functions are statically similar in syntax, while understanding their inherently similar execution semantics just further increases the similarity score.

6 CASE STUDIES ON VULNERABILITY SEARCHING

In this section, we study how TREX can help discover vulnerabilities in firmware images. Firmware images often include third-party libraries. However, these libraries are

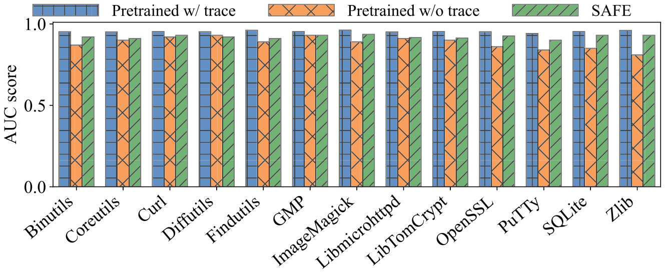
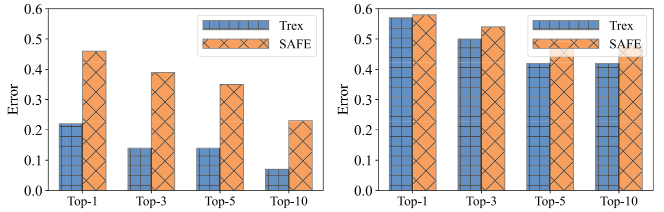


Fig. 11. Comparison of testing AUC scores between models pretrained *with* and *without* forced-execution trace.



(a) Same architecture (b) Cross architecture

Fig. 12. Top-1/3/5/10 error of Trex and SAFE in searching functions in firmware. The queries and the firmware are from (a) both ARM, (b) x64 and ARM, respectively.

TABLE 5
Vulnerabilities We Have Confirmed (✓) in Firmware Images (Latest Version) From 4 Well-Known Vendors and Products

CVE	Ubiquiti sunMax	TP-Link Deco-M4	NETGEAR R7000	Linksys RE7000
CVE-2016-6303	✓	✓	✓	✓
CVE-2016-6302	✓	✓	✓	✓
CVE-2016-2842	✓	✓	✓	✓
CVE-2016-2182	✓	✓	✓	✓
CVE-2016-2180	✓	✓	✓	✓
CVE-2016-2178	✓	✓	✓	✓
CVE-2016-2176	✓	✓	✗	✓
CVE-2016-2109	✓	✓	✗	✓
CVE-2016-2106	✓	✓	✗	✓
CVE-2016-2105	✓	✓	✗	✓
CVE-2016-0799	✓	✓	✗	✓
CVE-2016-0798	✓	✓	✗	✓
CVE-2016-0797	✓	✓	✗	✓
CVE-2016-0705	✓	✓	✗	✓

frequently patched but the manufacturers often fall behind in updating them accordingly [68]. Indeed, OWASP lists "using components with known vulnerabilities" as one of the top-10 application security risks in 2020 [68]. Therefore, we study whether our tool can uncover functions in firmware images similar to known vulnerable functions. We find existing state-of-the-art binary similarity tools all perform their case studies on the firmware images and vulnerabilities that have already been studied before. Therefore, we decide to collect our own dataset with more updated firmware images and the latest vulnerabilities, instead of reusing the existing benchmarks. This facilitates finding 1-day vulnerabilities in most recent firmware images not disclosed before.

Specifically, we crawl firmware images in 180 products from 22 vendors including WLAN routers, smart cameras, and solar panels, from well-known manufacturers' latest official releases and third-party providers such as DD-WRT [34] (see our supplementary material for firmware details), available online. For each function in the firmware images, we construct function embedding and build a database using Open Distro for Elasticsearch [5], which supports vector-based indexing with efficient search support based on NMSLIB [12].

Table 5 shows the 14 CVEs we use to search in the firmware images and we include their details in the supplementary details, available online. For each CVE, we compile the corresponding vulnerable functions in the specified library version and generate the vulnerable function embeddings via Trex. As the firmware images are stripped, we do not know with which optimizations they are compiled, we compile the vulnerable functions to both MIPS and ARM with `O3` and rely on Trex's cross-architecture/optimization matching capability to match functions potentially compiled in different architectures and with different optimizations. We then obtain the functions ranked top-10 similar to the vulnerable one and manually verify if they are vulnerable. We leverage `strings` command to identify the OpenSSL versions indicative of the corresponding vulnerabilities. Note that such information can be stripped for other libraries so it is not a reliable approach in general. We have confirmed all 14 CVEs in 4 firmware models (Table 5) developed by well-known vendors, i.e., Ubiquiti, TP-Link, NETGEAR, and Linksys. These cases demonstrate the practicality of Trex, which helps discover real-world vulnerabilities in large-scale firmware databases.

Vulnerability Search Performance. We quantify the accuracy of Trex in searching vulnerable functions in the firmware images and compare it to that of SAFE. As SAFE does not work for MIPS, we study how it performs on NETGEAR R7000 model, the only model that runs on ARM architecture from Table 5. Specifically, we compile OpenSSL to ARM and x64 with `O3`, and feed both our compiled and firmware's binaries to Trex and SAFE to compute embeddings. Based on the embeddings, we search the compiled OpenSSL functions in the NETGEAR R7000's embedded libraries, and test their top-1/3/5/10 errors. For example, the top-10 error measures when the query function does not appear in the top-10 most similar functions in the firmware. Fig. 12 shows that Trex has 5.5% and 5.6% lower error rate than SAFE, when the query functions are from the same or different architectures, respectively.

7 Threats to Validity

Learning Approximate Execution Semantics. In this paper, our pretraining task is designed to help an ML model towards reasoning how programs execute. However, it does not guarantee the trained model fully understands the execution semantics. Therefore, we can only resort to empirical studies by designing various measurements to test the trained model's understanding of execution semantics. Our evaluation shows the promise (see Appendix), available online, – Trex obtains high accuracy on predicting diverse masked code and trace values of millions of functions and generalizes to unseen functions and trace values. Our case studies (see Appendix), available online, on unseen test samples also demonstrate the model learns beyond simply memorizing patterns or taking spurious shortcuts.

While empirical evidence suggests that TREX likely learns approximate execution semantics, formally proving that an ML model has learned execution semantics precisely remains an open problem [43], [47]. So far, only a simple and restricted set of properties can be formally verified on a limited types of neural net architectures [16], [88], [92]. To the best of our knowledge, no existing works can verify that a model has learned execution semantics – an extremely complex and non-linear property of program code. Therefore, we envision an appealing future research direction in verifying that an ML model learned execution semantics correctly.

Ground Truth Bias. Following previous works [23], [57], [93], we treat functions compiled from the same source as similar, regardless of architectures, compilers, optimizations, and obfuscation transforms. However, two semantically similar functions can differ beyond architectures, compilers, etc., as long as they have the similar input-output behavior. For example, quick sort and merge sort are two equivalent implementations in terms of their input-output behavior. Therefore, TREX can suffer from potentially uncaptured false negatives, missing retrieving vulnerable functions when matching firmware images.

We note that obtaining the ground truths of arbitrary semantically similar functions is not easy. Without developers annotating those functions, it is hard to collect a large-scale training dataset on various software programs. As TREX aims to learn execution semantics without the function pair ground truth, it can potentially benefit this task as well. We leave this as the future study.

Dynamic Trace Bias. In this work, we use *concrete* dynamic traces to pretrain TREX. However, the concrete value space can be extremely large. This leads to the question how TREX pretrained on limited trace values generalizes to unseen samples. In Appendix, available online, we have included generalizability study for binary similarity. Moving forward, it can be interesting to study how pretrained TREX generalizes to unseen trace values and how to improve it. An interesting future direction is to use symbolic execution traces, a more compact form of program behavior, but the caveat is that symbolic execution is much more expensive than micro-execution, restricting its capability in obtaining large-scale training samples.

8 RELATED WORK

Traditional Approaches to Binary Similartiy. Existing static approaches extract hand-crafted features by domain experts to match similar functions. The features often encode the functions' syntactic characteristics. For example, BinDiff [100] extracts the number of basic blocks and the number of function calls to determine the similarity. Other works [18], [19], [29], [46], [47], [65] introduce more carefully-selected static features. For example, ESH [19] decomposes functions into strands of instructions based on data dependencies. They compare the function by the compose the similarity across these strands. Instead of relying on manually-defined compositions, TREX learns the semantics of compositions of instructions by predicting their execution effects (Section 2.2). Therefore, our pretraining task automates the process of encoding compositions. Our case study (see Appendix), available online, demonstrated how our model predicts the trace values by reasoning over the compositions of multiple instructions.

Another popular static approach is to compute the structural distance between functions to determine the similarity [11], [20], [21], [26], [38], [78], [100], such as the edit distance between basic block expression trees [78] or instruction sequences [21], [38]. As discussed in Section 2, these features are susceptible to obfuscations and optimizations. TREX automates learning approximate execution semantics and has been empirically shown more robust. In addition to the static approaches, dynamic approaches [25], [32], [37], [39], [45], [58], [61], [62], [77], [82] construct hand-coded dynamic features, such as values written to memory [25] or system calls [62] by executing the function to match similar functions. These approaches can detect semantically similar functions by observing their similar execution behavior. However, these approaches are expensive (because execution happens at query time) [77] and can suffer from false positives due to the noise introduced by forced-execution [23], [41]. TREX only uses the traces to learn approximate execution semantics of instructions and transfer the learned knowledge to match functions *without directly comparing their dynamic behavior*. Therefore, it is more efficient and less susceptible to the imprecision introduced by the forced-execution. Besides dynamic execution based on concrete inputs, symbolic execution has been proposed as an effective alternative to capture the comprehensive behavior of the program over all paths [15], [31], [53], [63]. However, the key limitation of symbolic execution approach is their scalability. The authors of CoP [53] have acknowledged their high computational overhead, taking an hour to complete a comparison between two reasonable-sized programs, e.g., thttpd and sthttpd. As a reference comparison, we run TREX on all the function pairs between thttpd (102 functions) and sthttpd (103 functions). TREX takes only 6.8 minutes to compare all the 10,000 pairs. Therefore, symbolic execution is much less practical in real-world use cases, e.g., matching large-scale functions, and the most recent study on binary similarity task [56] chose to discard all these approaches as they are inherently slow.

ML-Based Approaches to Binary Similarity. Most recent learning-based works [23], [24], [30], [42], [50], [57], [81], [93], [94], [96], [99] learn a function representation that is supposed to encode the function in low dimensional vectors, known as function embeddings [56]. The embeddings are constructed by taking the functions' structures (e.g., control flow graph) [24], [28], [30], [59], [93] or instruction sequences [23], [50], [57] and training a neural net to align the function embedding distances to the similarity scores. All existing approaches are based only on static code, which lacks the knowledge of function execution semantics. Moreover, the ML architectures adopted in these approaches require constructing expensive graph features (attributed CFG [30], [93]). By contrast, TREX learns approximate execution semantics from traces without extra manual feature engineering effort. Recently, Marcelli et al. [56] evaluated a fairly comprehensive set of ML based binary similarity tools, in which TREX ranks the second in terms of vulnerability searching performance. The best performed model based on graph matching neural networks [51], however, requires pairwise comparison for retrieval, i.e., it cannot extract embeddings and perform approximate nearest neighbor searching. Therefore, it suffers from poor scalability.

Learning Representations of Program Code. There has been a growing interest in learning neural program representation for code modeling tasks [4]. The learned embedding of the code encodes the program's key properties, useful for many applications such as program repair [69], [86], [89], recovering symbol names, types, memory dependencies, and other higher-level constructs [8], [9], [17], [36], [40], [50], [55], [71], [73], [74], bug detection and investigation [2], [35], [54], [60], [64], [80], [83], [85], [90], [91], and forensics [14], [48], [52]. Recent studies have shown promising results that the learned program representations can be further improved by program execution behaviors [66], [73], [86], [87]. As opposed to just incorporating traces as additional input [86], [87], TREX shows that ML models can learn approximate execution semantics from large-scale traces explicitly and still improves downstream analysis tasks, even the traces are noisy and might deviate substantially from their actual program behavior. Such a relaxation on the quality of traces can potentially benefit a broad spectrum of program analysis tasks where collecting traces is challenging (i.e., interpreting compiled languages directly).

9 CONCLUSION

We introduced TREX to match semantically similar functions based on the function execution semantics. We design a pretraining task to pretrain an ML model to learn approximate execution semantics from noisy forced-execution traces and then transfer the learned knowledge to match semantically similar functions. Our evaluation showed that pretraining on forced-execution traces drastically improves the accuracy of matching semantically similar functions – TREX excels in matching functions across different architectures, optimizations, and obfuscations. We release the code and dataset of TREX at https://github.com/CUMLSec/trex.

ACKNOWLEDGMENTS

Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, ONR, ARL, NSF, Captital One, Google, JP Morgan, DiDi, or Amazon.

REFERENCES

[1] H. Aghakhani et al., "When malware is packin' heat; limits of machine learning classifiers based on static analysis features," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020.

[2] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics*, 2021, pp. 2655–2668.

[3] M. Ahmadi, R. M. Farkhani, R. Williams, and L. Lu, "Finding bugs using your own code: Detecting functionally-similar yet inconsistent code," in *Proc. 30th USENIX Secur. Symp.*, 2021, pp. 2025–2040.

[4] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Comput. Surv.*, vol. 51, 2018, Art. no. 81.

[5] I. Amazon Web Services, "Open distro for elasticsearch," 2020. [Online]. Available: https://opendistro.github.io/for-elasticsearch/

[6] D. Arp et al., "Dos and don'ts of machine learning in computer security," 2020, *arXiv:2010.09470*.

[7] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic exploit generation," *Commun. ACM*, vol. 57, pp. 74–84, 2014.

[8] P. Banerjee, K. K. Pal, F. Wang, and C. Baral, "Variable name recovery in decompiled binary code using constrained masked language modeling," 2021, *arXiv:2103.12801*.

[9] S. Bardin, T. Benoit, and J.-Y. Marion, "Compiler and optimization level recognition using graph neural networks," in *Proc. Mach. Learn. Prog. Anal.*, 2021.

[10] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2009, pp. 8–11.

[11] M. Bourquin, A. King, and E. Robbins, "BinSlayer: Accurate comparison of binary executables," in *Proc. 2nd ACM SIGPLAN Prog. Protection Reverse Eng. Workshop*, 2013, Art. no. 4.

[12] L. Boytsov and B. Naidan, "Engineering efficient and effective non-metric space library," in *Proc. Int. Conf. Similarity Search Appl.*, 2013, pp. 280–293.

[13] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *Proc. IEEE Symp. Secur. Privacy*, 2008, pp. 143–157.

[14] L. Cai et al., "Structural temporal graph neural networks for anomaly detection in dynamic graphs," in *Proc. 30th ACM Int. Conf. Inf. Knowl. Manage.*, 2021, pp. 3747–3756.

[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 *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 678–689.

[16] Y. Chen, S. Wang, Y. Qin, X. Liao, S. Jana, and D. Wagner, "Learning security classifiers with verified global robustness properties," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2021, pp. 477–494.

[17] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 99–116.

[18] J. Crussell, C. Gibler, and H. Chen, "AnDarwin: Scalable detection of semantically similar android applications," in *Proc. Eur. Symp. Res. Comput. Secur.*, 2013, pp. 182–199.

[19] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," *ACM SIGPLAN Notices*, vol. 51, pp. 266–280, 2016.

[20] Y. David, N. Partush, and E. Yahav, "Similarity of binaries through re-optimization," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2017, pp. 79–94.

[21] Y. David and E. Yahav, "Tracelet-based code search in executables," *ACM SIGPLAN Notices*, vol. 49, pp. 349–360, 2014.

[22] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Annu. Conf. North Amer. Chapter Assoc. Comput. Linguistics: Hum. Lang. Technol.*, 2019, pp. 4171–4186.

[23] S. H. Ding, B. C. Fung, and P. Charland, "Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *Proc. IEEE Symp. Secur. Privacy*, 2019, pp. 472–489.

[24] Y. Duan, X. Li, J. Wang, and H. Yin, "DeepBinDiff: Learning program-wide code representations for binary diffing," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020.

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

[26] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovRE: Efficient cross-architecture identification of bugs in binary code," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2016, pp. 58–79.

[27] M. Fan et al., "Graph embedding based familial analysis of Android malware using unsupervised learning," in *Proc. IEEE/ACM Int. Conf. Softw. Eng.*, 2019, pp. 771–782.

[28] C. Fang, Z. Liu, Y. Shi, J. Huang, and Q. Shi, "Functional code clone detection with syntax and semantics fusion learning," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2020, pp. 516–527.

[29] M. R. Farhadi, B. C. Fung, P. Charland, and M. Debbabi, "BinClone: Detecting code clones in malware," in *Proc. Int. Conf. Softw. Secur. Rel.*, 2014, pp. 78–87.

[30] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 480–491.

[31] D. Gao, M. K. Reiter, and D. Song, "BinHunt: Automatically finding semantic differences in binary programs," in *Proc. Int. Conf. Inf. Commun. Secur.*, 2008, pp. 238–255.

[32] J. Gao, X. Yang, Y. Fu, Y. Jiang, H. Shi, and J. Sun, "VulSeeker-pro: Enhanced semantic learning based binary vulnerability seeker with emulation," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2018, pp. 803–808.

[33] A. Gionis et al., "Similarity search in high dimensions via hashing," in *Proc. 25th Int. Conf. Very Large Data Bases*, 1999, pp. 518–529.
[34] S. Gottschall, "DD-WRT," 2005. [Online]. Available: https://dd-wrt.com/
[35] W. Guo, D. Mu, X. Xing, M. Du, and D. Song, "DEEPVSA: Facilitating value-set analysis with deep learning for postmortem program analysis," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 1787–1804.
[36] K. Heo, H. Oh, and K. Yi, "Machine-learning-guided selectively unsound static analysis," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng.*, 2017, pp. 519–529.
[37] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Binary code clone detection across architectures and compiling configurations," in *Proc. IEEE/ACM Int. Conf. Prog. Comprehension*, 2017, pp. 88–98.
[38] H. Huang, A. M. Youssef, and M. Debbabi, "BinSequence: Fast, accurate and scalable binary code reuse detection," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, 2017, pp. 155–166.
[39] J. Jang, M. Woo, and D. Brumley, "Towards automatic software lineage inference," in *Proc. 22nd USENIX Secur. Symp.*, 2013, pp. 81–96.
[40] M. Jeon, S. Jeong, S. Cha, and H. Oh, "A machine-learning algorithm with disjunctive model for data-driven program analysis," *ACM Trans. Program. Lang. Syst.*, vol. 41, no. 2, pp. 1–41, 2019.
[41] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *Proc. 18th Int. Symp. Softw. Testing Anal.*, 2009, pp. 81–92.
[42] S. Jiang, C. Fu, Y. Qian, S. He, J. Lv, and L. Han, "IFAttn: Binary code similarity analysis based on interpretable features with attention," *Comput. Secur.*, vol. 120, 2022, Art. no. 102804.
[43] Z. Jiang, F. F. Xu, J. Araki, and G. Neubig, "How can we know what language models know?," *Trans. Assoc. Comput. Linguistics*, vol. 8, pp. 423–438, 2020.
[44] Z. Jiang et al., "PDiff: Semantic-based patch presence testing for downstream kernels," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2020, pp. 1149–1163.
[45] A. Kapravelos, Y. Shoshitaishvili, M. Cova, C. Kruegel, and G. Vigna, "Revolver: An automated approach to the detection of evasive web-based malware," in *Proc. 22nd USENIX Secur. Symp.*, 2013, pp. 637–652.
[46] W. M. Khoo, A. Mycroft, and R. Anderson, "Rendezvous: A search engine for binary code," in *Proc. 10th Work. Conf. Mining Softw. Repositories*, 2013, pp. 329–338.
[47] D. Kim, E. Kim, S. K. Cha, S. Son, and Y. Kim, "Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned," 2020, *arXiv:2011.10749*.
[48] H. Koo, S. Park, D. Choi, and T. Kim, "Semantic-aware binary code representation with bert," 2021, *arXiv:2106.05478*.
[49] S. Kullback and R. A. Leibler, "On information and sufficiency," *Ann. Math. Statist.*, vol. 22, no. 1, pp. 79–86, 1951.
[50] X. Li, Q. Yu, and H. Yin, "PalmTree: Learning an assembly language model for instruction embedding," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2021, pp. 3236–3251.
[51] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli, "Graph matching networks for learning the similarity of graph structured objects," in *Proc. Int. Conf. Mach. Learn.*, 2019, pp. 3835–3845.
[52] F. Liu, Y. Wen, D. Zhang, X. Jiang, X. Xing, and D. Meng, "Log2vec: A heterogeneous graph embedding based approach for detecting cyber threats within enterprise," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 1777–1794.
[53] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 389–400.
[54] O. Lutz et al., "ESCORT: Ethereum smart contracts vulnerability detection using deep neural network and transfer learning," 2021, *arXiv:2103.12607*.
[55] A. Maier, H. Gascon, C. Wressnegger, and K. Rieck, "TypeMiner: Recovering types in binary programs using machine learning," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*, 2019, pp. 288–308.
[56] A. Marcelli, M. Graziano, X. Ugarte-Pedrero, Y. Fratantonio, M. Mansouri, and D. Balzarotti, "How machine learning is solving the binary function similarity problem," in *Proc. 31st USENIX Secur. Symp.*, 2022, pp. 2099–2116.
[57] L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni, "SAFE: Self-attentive function embeddings for binary similarity," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*, 2019, pp. 309–329.
[58] D. McKee, N. Burow, and M. Payer, "Software ethology: An accurate and resilient semantic binary analysis framework," 2019, *arXiv:1906.02928*.
[59] N. Mehrotra, N. Agarwal, P. Gupta, S. Anand, D. Lo, and R. Purandare, "Modeling functional similarity in source code with graph-based Siamese networks," 2020, *arXiv:2011.11228*.
[60] W. Melicher, C. Fung, L. Bauer, and L. Jia, "Towards a lightweight, hybrid approach for detecting DOM XSS vulnerabilities with machine learning," in *Proc. Web Conf.*, 2021, pp. 2684–2695.
[61] J. Ming, M. Pan, and D. Gao, "iBinHunt: Binary hunting with inter-procedural control flow," in *Proc. Int. Conf. Inf. Secur. Cryptol.*, 2012, pp. 92–109.
[62] J. Ming, D. Xu, Y. Jiang, and D. Wu, "BinSim: Trace-based semantic binary diffing via system call sliced segment equivalence checking," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 253–270.
[63] J. Ming, D. Xu, and D. Wu, "Memoized semantics-based binary diffing with application to malware lineage inference," in *Proc. IFIP Int. Inf. Secur. Privacy Conf.*, 2015, pp. 416–430.
[64] D. Mu et al., "RENN: Efficient reverse execution with neural-network-assisted alias analysis," in *Proc. IEEE/ACM 34th Int. Conf. Autom. Softw. Eng.*, 2019, pp. 924–935.
[65] G. Myles and C. Collberg, "K-gram based software birthmarks," in *Proc. ACM Symp. Appl. Comput.*, 2005, pp. 314–318.
[66] M. Nye et al., "Show your work: Scratchpads for intermediate computation with language models," 2021, *arXiv:2112.00114*.
[67] M. Ott et al., "fairseq: A fast, extensible toolkit for sequence modeling," in *Proc. Annu. Conf. North Amer. Chapter Assoc. Comput. Linguistics: Hum. Lang. Technol.: Demonstrations*, 2019, pp. 48–53.
[68] OWASP, "Top 10 web application security risks," 2010. [Online]. Available: https://owasp.org/www-project-top-ten/
[69] S. Parihar, Z. Dadachanji, P. K. Singh, R. Das, A. Karkare, and A. Bhattacharya, "Automatic grading and feedback using program repair for introductory programming courses," in *Proc. ACM Conf. Innov. Technol. Comput. Sci. Educ.*, 2017, pp. 92–97.
[70] A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, Art. no. 721.
[71] J. Patrick-Evans, L. Cavallaro, and J. Kinder, "Probabilistic naming of functions in stripped binaries," in *Proc. Annu. Comput. Secur. Appl. Conf.*, 2020, pp. 373–385.
[72] M. Payer, S. Crane, P. Larsen, S. Brunthaler, R. Wartell, and M. Franz, "Similarity-based matching meets malware diversity," 2014, *arXiv:1409.7760*.
[73] K. Pei et al., "StateFormer: Fine-grained type recovery from binaries using generative state modeling," in *Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2021, pp. 690–702.
[74] K. Pei, J. Guan, D. Williams-King, J. Yang, and S. Jana, "XDA: Accurate, robust disassembly with transfer learning," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2021.
[75] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "X-force: Force-executing binary programs for security applications," in *Proc. 23rd USENIX Secur. Symp.*, 2014, pp. 829–844.
[76] R. Perdisci, A. Lanzi, and W. Lee, "McBoost: Boosting scalability in malware collection and analysis using statistical classification of executables," in *Proc. Annu. Comput. Secur. Appl. Conf.*, 2008, pp. 301–310.
[77] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *Proc. IEEE Symp. Secur. Privacy*, 2016, pp. 709–724.
[78] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow, "Leveraging semantic signatures for bug search in binary programs," in *Proc. 30th Annu. Comput. Secur. Appl. Conf.*, 2014, pp. 406–415.
[79] N. A. Quynh and D. H. Vu, "Unicorn: Next generation CPU emulator framework," *BlackHat USA*, 2015.
[80] S. Reddy, C. Lemieux, R. Padhye, and K. Sen, "Quickly generating diverse valid test inputs with reinforcement learning," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng.*, 2020, pp. 1410–1421.

[81] K. Redmond, L. Luo, and Q. Zeng, "A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis," 2018, *arXiv:1812.09652*.

[82] K. Rieck, P. Trinius, C. Willems, and T. Holz, "Automatic analysis of malware behavior using machine learning," *J. Comput. Secur.*, vol. 19, pp. 639–668, 2011.

[83] S. L. Shrestha and C. Csallner, "SLGPT: Using transfer learning to directly generate simulink model files and find bugs in the simulink toolchain," 2021, *arXiv:2105.07465*.

[84] A. Vaswani et al., "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 6000–6010.

[85] H. Wang et al., "Combining graph-based learning with automated data collection for code vulnerability detection," *IEEE Trans. Inf. Forensics Security*, vol. 16, no. 9151439, pp. 1943–1958, Dec. 2020.

[86] K. Wang, R. Singh, and Z. Su, "Dynamic neural program embedding for program repair," in *Proc. Int. Conf. Learn. Representations*, 2017.

[87] K. Wang and Z. Su, "Blended, precise semantic program embeddings," in *Proc. 41st ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2020, pp. 121–134.

[88] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana, "Formal security analysis of neural networks using symbolic intervals," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 1599–1614.

[89] S. Wang, P. Wang, and D. Wu, "Semantics-aware machine learning for function recognition in binary code," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2017, pp. 388–398.

[90] L. Wartschinski, Y. Noller, T. Vogel, T. Kehrer, and L. Grunske, "VUDENC: Vulnerability detection with deep learning on a natural codebase for python," *Inf. Softw. Technol.*, vol. 144, 2022, Art. no. 106809.

[91] M. Wen, R. Wu, and S.-C. Cheung, "How well do change sequences predict defects? Sequence learning from software changes," *IEEE Trans. Softw. Eng.*, vol. 46, no. 11, pp. 1155–1175, Nov. 2020.

[92] E. Wong and Z. Kolter, "Provable defenses against adversarial examples via the convex outer adversarial polytope," in *Proc. Int. Conf. Mach. Learn.*, 2018, pp. 5286–5295.

[93] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 363–376.

[94] J. Yang, C. Fu, X.-Y. Liu, H. Yin, and P. Zhou, "Codee: A tensor embedding scheme for binary code search," *IEEE Trans. Softw. Eng.*, vol. 48, no. 7, pp. 2224–2244, Jul. 2022.

[95] L. Yang et al., "CADE: Detecting and explaining concept drift samples for security applications," in *Proc. 30th USENIX Secur. Symp.*, 2021, pp. 2327–2344.

[96] Z. Yu, W. Zheng, J. Wang, Q. Tang, S. Nie, and S. Wu, "CodeCMR: Cross-modal retrieval for function-level binary source code matching," in *Proc. Adv. Neural Inf. Process. Syst.*, 2020, pp. 3872–3883.

[97] N. Zhang, "Hikari–an improvement over Obfuscator-LLVM," 2017. [Online]. Available: https://github.com/HikariObfuscator/Hikari

[98] L. Zhao, Y. Zhu, J. Ming, Y. Zhang, H. Zhang, and H. Yin, "PatchScope: Memory object centric patch diffing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2020, pp. 149–165.

[99] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," 2018, *arXiv:1808.04706*.

[100] Zynamics, "BinDiff," 2019. [Online]. Available: https://www.zynamics.com/bindiff.html

Kexin Pei is currently working toward the PhD degree with Columbia University, New York. His research interests include security, software engineering, and machine learning, specifically focusing on developing data-driven program analysis approaches to improve the reliability and security of traditional and AI-based software systems.

Zhou Xuan received the BE degree in software engineering from Xidian University, in China. He is currently working toward the PhD degree with Purdue University. His research interests include deep learning for program analysis and audit log analysis.

Junfeng Yang is professor of computer science, member of the Data Science Institute, and co-director of the Software Systems Lab, Columbia University. His research centers on building reliable, secure, and fast software systems. His research has resulted in numerous vulnerability patches to real-world systems, practical adoption with the largest technology companies, and press coverage with Scientific American, The Atlantic, The Register, *Communications of ACM*, and other news outlets. He won the Sloan Research Fellowship and the Air Force Office of Scientific Research Young Investigator Program Award, both in 2012; the National Science Foundation CAREER award, in 2011; and Best Paper Awards at the USENIX Symposium on Operating System Design and Implementation, in 2004 and 2022, the ACM Symposium on Operating Systems Principles, in 2017, and the USENIX Annual Technical Conference, in 2021.

Suman Jana is an Associate Professor with the Computer Science Department and the Data Science institute with Columbia University. His primary research interest is at the intersection of computer security and machine learning. His research has received seven best paper awards, a CACM research highlight, a Google faculty fellowship, a JPMorgan Chase Faculty Research Award, an NSF CAREER award, and an ARO young investigator award.

Baishakhi Ray is an Associate Professor with the Department of Computer Science, Columbia University, New York. Her research interest is in the intersection of software engineering and machine learning. She has received the prestigious IEEE TCSE Rising star award and NSF CAREER award. Her research has been acknowledged by Distinguished Paper awards with ASE 2022, FASE 2020, FSE 2017, MSR 2017, and IEEE Symposium on Security and Privacy (Oakland), 2014. Her research has also been published in CACM Research Highlights and has been widely covered in trade media.