TREX: Learning Execution Semantics from Micro-Traces for Binary Similarity

Kexin Pei  
Columbia University  
kpei@cs.columbia.edu

Zhou Xuan  
University of California, Riverside  
xzuan004@ucr.edu

Junfeng Yang  
Columbia University  
junfeng@cs.cs.columbia.edu

Suman Jana  
Columbia University  
suman@cs.columbia.edu

Baishakhi Ray  
Columbia University  
rayb@cs.columbia.edu

Abstract—Detecting semantically similar functions – a crucial analysis capability with broad real-world security usages including vulnerability detection, malware lineage, and forensics – requires understanding function behaviors and intentions. However, this task is challenging as semantically similar functions can be implemented differently, run on different architectures, and compiled 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 execution semantics explicitly from functions’ micro-traces (a form of under-constrained dynamic traces) and transfer the learned knowledge to match semantically similar functions. While such micro-traces are known to be too imprecise to be directly used to detect semantic similarity, our key insight is that these traces can be used to teach an ML model the execution semantics of different sequences of instructions. We thus design an unsupervised pretraining task, which trains the model to learn execution semantics from the functions’ micro-traces without any manual labeling or feature engineering effort. We then develop a novel neural architecture, hierarchical Transformer, which can learn execution semantics from micro-traces during the pretraining phase. Finally, we finetune the pretrained model to match semantically similar functions.

We evaluate TREX on 1,472,066 function binaries from 13 popular software projects. These functions are from different architectures (x86, x64, ARM, and MIPS) and compiled with 4 optimizations (00-03) and 5 obfuscations. TREX outperforms the state-of-the-art systems by 7.8%, 7.2%, and 14.3% in cross-architecture, optimization, and obfuscation function matching, respectively, while running 8× faster. Our ablation studies show that the pretraining task significantly boosts the function matching performance, underscoring the importance of learning execution semantics. Moreover, our extensive case studies demonstrate the practical use-cases of TREX – on 180 real-world firmware images with their latest version, TREX uncovers 16 vulnerabilities that have not been disclosed by any previous studies. We release the code and dataset of TREX at https://github.com/CUMLSec/trex.

I. 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 [12], exploit generation [5], tracing malware lineage [7], [41], and forensics [49]. For example, OWASP lists “using components with known vulnerabilities” as one of the top-10 application security risks in 2020 [56]. Therefore, identifying similar vulnerable functions in 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 instruction set architectures with various compiler optimizations or obfuscated with simple transformations.

Recently, Machine Learning (ML) based approaches have shown promise in tackling these challenges [25], [50], [77] by learning robust 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 [25], [50], [77], outperforming the traditional signature-based methods [79] using hand-crafted features (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 [51]. 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 patterns are absent or changed [1], [61].
Phase 1: Pretraining (Masked LM)

- Assembly code
- Micro-traces

Pretrained Model

Transfer

Phase 2: Finetuning (Function Similarity)

Compile

Function Source

Semantically Similar

Pretrained Model

Stacked Head

h₁

Function Embedding

Maximize Agreement

Pretrained Model

Stacked Head

h₂

Function Binaries

Query

Phase 3: Inference (Matching Functions)

- Function Embedding Database
- Stacked Head

Function Model

Pretrained Model

Query

Function Binary

Fig. 1. The workflow of T Rex. We first pretrain the model on the functions’ micro-traces, consisting of both instructions and dynamic values, using the masked LM task. We then finetune the pretrained model on the semantically similar function (only static instruction) pairs by stacking new neural network layers for function similarity tasks. Finetuning updates both the pretrained model and the stacked layers. During inference, the finetuned model computes the function embedding, whose distance encodes the function similarity.

Our key observation is that while under-constrained dynamic execution traces tend to contain many infeasible states, they still encode precise execution effects of many individual instructions. Thus, we can train an ML model to observe and learn the effect of different instructions present across a large number of under-constrained dynamic traces collected from diverse functions. Once the model has gained an approximate understanding of execution semantics of various instructions, we can train it to match semantically similar functions by leveraging its learned knowledge. As a result, during inference, we do not need to execute any functions on-the-fly while matching them [45], which saves significant runtime overhead. Moreover, our trained model does not need the under-constrained dynamic traces to match functions, it only uses the function instructions, but they are augmented with rich knowledge of execution semantics.

In this paper, we extend micro-execution [34], a form of under-constrained dynamic execution, to generate micro-traces of a function across multiple instruction set architectures. A micro-trace consists of a sequence of aligned instructions and their corresponding program state values. We pretrain the model on a large number of micro-traces gathered from diverse functions as part of training data using the masked language modeling (masked LM) task. Notably, masked LM masks random parts in the sequence and asks the model to predict masked parts based on their context. This design forces the model to learn approximately how a function executes to correctly infer the missing values, which automates learning execution semantics without manual feature engineering. Masked LM is also fully self-supervised [22] – T Rex can thus be trained and further improved with arbitrary functions found in the wild.

To this end, we design a hierarchical Transformer [75] that supports learning approximate execution semantics. Specifically, our architecture models micro-trace values explicitly. By contrast, existing approaches often treat the numerical values as a dummy token [25], [50] to avoid prohibitively large vocabulary size, which cannot effectively learn the rich dependencies between concrete values that likely encode key function semantics. Moreover, our architecture’s self-attention
layer is designed to model long-range dependencies in a sequence [75] efficiently. Therefore, T\textsc{rex} can support roughly 170× longer sequence and runs 8× faster than existing neural architectures, essential to learning embeddings of long function execution traces.

We evaluate T\textsc{rex} on 1,472,066 functions collected from 13 popular open-source software projects across 4 architectures (x86, x64, ARM, and MIPS) and compiled with 4 optimizations (O0-O3), and 5 obfuscation strategies [78]. T\textsc{rex} 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 significantly improves the accuracy of matching semantically similar functions (by 15.7%). We also apply T\textsc{rex} 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 T\textsc{rex} helps find 16 CVEs in these firmware images, which have not been disclosed in previous studies. We make the following contributions.

- We propose a new approach to matching semantically similar functions: we first train the model to learn approximate program execution semantics from micro-traces, a form of under-constrained dynamic traces, and then transfer the learned knowledge to identify semantically similar functions.
- We extend micro-execution to support different architectures to collect micro-traces for training. We then develop a novel neural architecture – hierarchical Transformer – to learn approximate execution semantics from micro-traces.
- We implement T\textsc{rex} and evaluate it on 1,472,066 functions from 13 popular software projects and libraries. T\textsc{rex} outperforms the state-of-the-art tools by 7.8%, 7%, and 14.3%, in cross-architecture, optimization, and obfuscation function matching, respectively, while running up to 8× faster. Moreover, T\textsc{rex} helps uncover 16 vulnerabilities in 180 real-world firmware images with the latest version that are not disclosed by previous studies. We release the code and dataset of T\textsc{rex} at https://github.com/CUMLSec/trex.

II. Overview

In this section, we use the real-world functions as motivating examples to describe the challenges of matching semantically similar functions. We then overview our approach, focusing on how our pretraining task (masked LM) addresses the challenges.

A. Challenging Cases

We use three semantically equivalent but syntactically different function pairs to demonstrate some challenges of learning from only static code. Figure 2 shows the (partial) assembly code of each function.

Cross-architecture example. Consider the functions in Figure 2a. Two functions have the same execution semantics as both functions take the lower 12-bit of a register and compare it to 0x80. Detecting this similarity requires understanding the approximate execution semantics of and in x86 and lsl/lsr in ARM. Moreover, it also requires understanding how the values (i.e., 0xfff and 0x14) in the code are manipulated. However, all existing ML-based approaches [50] only learn on static code without observing each instruction’s real execution effect. Furthermore, to mitigate the potentially prohibitive vocabulary size (i.e., all possible memory addresses), existing approaches replace all register values and memory addresses with an abstract dummy symbol [26], [50]. They thus cannot access the specific byte values to determine inherent similarity.

Cross-optimization example. Now consider two functions in Figure 2b. They are semantically equivalent as [ebp+8] and [esp+4] access the same memory location, i.e., the function’s first 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 dynamic information is not reflected in the static code.

Cross-obfuscation example. Figure 2c demonstrates a simple obfuscation by instruction substitution, which essentially replaces eax+1 with eax=−(−1). Detecting the equivalence requires understanding approximately how arithmetic operations such as xor, sub, and add, executes. However, static information is not enough to expose such knowledge.

B. Pretraining Masked LM on Micro-traces

This section describes how the pretraining task, masked LM, on functions’ micro-traces encourages the model to learn execution semantics. Although it remains an open research question to explicitly prove certain knowledge is encoded by such language modeling task [70], we focus on describing the intuition behind the masked LM – why predicting masked codes and values in micro-traces can help address the challenging cases in Figure 2.

Masked LM. Recall the operation of masked LM: given a function’s micro-trace (i.e., values and instructions), we mask some random parts and train the model to predict the masked parts using those not masked.

Note that pretraining with masked LM does not need any manual labeling effort, as it only predicts the masked part in the input micro-traces without any additional labeling effort. Therefore, T\textsc{rex} can be trained and further improved with a substantial number of functions found in the wild. The benefit of this is that a certain instruction not micro-executed in one function is highly likely to appear in at least one of the other functions’ micro-traces, supporting T\textsc{rex} to approximate diverse instructions’ execution semantics.

Masking register. Consider the functions in Figure 2c, where they essentially increment the value at stack location [rbp-0x2c] by 1. The upper function directly loads the value to eax, increments by 1, and stores the value in eax back to stack. The lower function, by contrast, takes a convoluted way
by first letting `ecx` to hold the value -1, and decrements `eax` by `ecx`, and stores the value in `eax` back to stack.

We mask the `eax` at line 3 in the upper function. We find that our pretrained model can correctly predict its name and dynamic value. This implies the model understands 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). We also find the model can recover the values of masked `ecx` in line 4 and `eax` in line 5, implying the model understands the execution effect of `xor` and `sub`.

The understanding of such semantics can significantly improve the robustness in matching similar functions – when finetuned to match similar functions, the model is more likely to learn to attribute the similarity to their similar execution effects, instead of their syntactic similarity.

**Masking opcode.** Besides masking the register and its value, we can also mask the opcode of an instruction. Predicting the opcode requires the model to understand the execution effect of each opcode. Consider Figure 2b, where we mask `mov` in line 2 of upper function. We find our pretrained model predicts `mov` with the largest probability (larger than the other potential candidates such as `add`, `inc`, etc.).

To correctly predict the opcode, the model should have learned several key aspects of the function semantics. First, according to its context, i.e., the value of `ebp` at line 3 and `esp` at line 2, it learns `mov` is most probable as it assigns the value of `esp` to `ebp`. Other opcodes are less likely as their execution effect conflicts with the observed resulting register values. This also implicitly implies the model learns the approximate execution semantics of `mov`. Second, the model also learns the common calling convention and basic syntax of x86 instructions, e.g., only a subset of opcodes accept two operands (`ebp`, `esp`). It can thus exclude many syntactically impossible opcodes such as `push`, `jmp`, etc.

The model can thus infer `ebp` (line 3 of upper function) equals to `esp`. The model may have also learned push decrements stack pointer esp by 4 bytes, from other masked samples. Therefore, when the pretrained model is finetuned to match the two functions, the model is more likely to learn that the semantic equivalence is due to that `ebp+8` in the upper function and `esp+4` in the lower function refer to the same address, instead of their similar syntax.

**Other masking strategies.** Note that we are not constrained by the number or the type of items (i.e., register, opcode, etc.) in the instructions to mask, i.e., we can mask complete instructions or even a consecutive sequence of instructions, and we can mask dynamic values of random instructions’ input-output. Moreover, the masking operation dynamically selects random subsets of code blocks and program states at each training iteration and on different training samples. As a result, it enables the model to learn the diverse and composite effect of the instruction sequence, essential to detecting similarity between functions with various instructions. In this paper, we adopt a completely randomized strategy to choose what part of the micro-trace to mask with a fixed masking percentage (see Section IV-C for details). However, we envision a quite interesting future work to study a better (but still cheap) strategy to dynamically choose where and how much to mask.

### III. Threat Model

We assume no access to the debug symbols or source while comparing binaries. Indeed, there exist many approaches to reconstruct functions from stripped binaries [4], [6], [24], [62], [72]. Moreover, we assume the binary can be readily disassembled, i.e., it is not packed nor transformed by virtualization-based obfuscator [73], [74].

**Semantic similarity.** We consider two semantically similar functions as having the same input-output behavior (i.e., given the same input, two functions produce the same output). Similar to previous works [25], [50], [77], we treat functions compiled
from the same source as similar, regardless of architectures, compilers, optimizations, and obfuscation transforms.

IV. METHODOLOGY

This section describes T.REX’s design specifics, including our micro-tracing semantics, our learning architecture’s details, and pretraining and finetuning workflow.

A. Micro-tracing Semantics

We implement micro-execution by Godefroid [34] to handle x64, ARM, and MIPS, where the original paper only describes x86 as the use case. In the following, we briefly explain how we micro-execute an individual function binary, highlighting the key algorithms in handling different types of instructions.

IR Language. To abstract away the complexity of different architectures’ assembly syntax, we introduce a low-level intermediate representation (IR) that models function assembly code. We only include a subset of the language specifics to illustrate the implementation algorithm. Figure 3 shows the grammar of the IR. Note that the IR here only serves to facilitate the discussion of our micro-tracing implementation. In our implementation, we use real assembly instructions and tokenize them as model’s input (Section IV-B).

Notably, we denote memory reads and writes by load(e) and store(e0, eα) (i.e., store the value expression e0 to address expression eα), which generalize from the load-store architecture (i.e., ARM, MIPS) and register-memory architecture (i.e., x86). Both operations can take as input 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 general jump instruction, which can be both direct or indirect jump (i.e., the expression eα can be a constant c or a register r). The jump instruction can also be unconditional or conditional. Therefore, the first parameter in jmp is the conditional expression eα and unconditional jump will set eα to true. We represent function invocations and returns by call and ret, where call is parameterized by an expression, which can be an address (direct call) or a register (indirect call).

Micro-tracing algorithm. Algorithm 1 outlines the basic steps of micro-tracing a given function f. First, it initializes the memory to load the code and the corresponding stack. It then initializes all registers except the special-purpose register, such as the stack pointer or the program counter. Then it starts linearly executing instructions of f. We map the memory address on-demand if the instruction access the memory (i.e., read/write). If the instruction reads from memory, we further initialize a random value in the specific memory addresses. For call/jump instructions, we first examine the target address and skip the invalid jump/call, known as “forced execution” [63]. By skipping unreachable jumps and calls, it can keep executing the function till the end of the function and exposes more behaviors, e.g., skipping potential input check exceptions. Since the nop instructions can serve as padding between instructions within a function, we simply skip nop. We terminate the micro-tracing when it finishes executing all instructions, reaches ret, or times out. Figure 13 and 14 demonstrate sample micro-traces of real-world functions.

B. Input Representation

Formally, given a function f (i.e., assembly code) and its micro-trace t (by micro-executing f), we prepare the model input x, consisting of 5 types of token sequence with the same size n. Figure 4 shows the model input example and how they are masked and processed by the hierarchical Transformer to predict the corresponding output as a pretraining task.

Micro-trace code sequence. The first sequence x_f is the assembly code sequence: x_f = \{mov, eax, ...\}^n, generated by tokenizing the assembly instructions in the micro-trace. We treat all symbols appear in the assembly instructions as tokens. Such a tokenization aims to preserve the critical hint of the syntax and semantics of the assembly instructions. For example, we consider even punctuation to be one of the tokens, e.g., “,”, “[”, ]”, as “,” implies the token before and after it as...
destination and source of mov (in Intel syntax), respectively, and “[” and “]” denote taking the address of the operands reside in between them.

We take special treatment of numerical values appear in the assembly code. Treating numerical values as regular text tokens can incur prohibitively large vocabulary size, e.g., $2^{32}$ number of possibilities on 32-bit architectures. To avoid this problem, we move all numeric values to the micro-trace value sequence (that will be learned by an additional neural network as detailed in the following) and replace them with a special token num (e.g., last token of input in Figure 4). With all these preprocessing steps, the vocabulary size of $x_f$ across all architectures is 3,300.

**Micro-trace value sequence.** The second sequence $x_t$ is the micro-trace value sequence, where each token in $x_t$ is the dynamic value from micro-tracing the corresponding code. As discussed in Section II, we keep explicit values (instead of a dummy value used by existing approaches) in $x_t$. Notably, we use the dynamic value for each token (e.g., register) in an instruction before it is executed. For example, in mov eax, 0x8; mov eax, 0x3, the dynamic value of the second eax is 0x8, as we take the value of eax before mov eax, 0x3 is executed. For code token without dynamic value, e.g., mov, we use dummy values (see below).

**Position sequences.** The position of each code and value token is critical for inferring binary semantics. Unlike natural language, where swapping two words can roughly preserve the same semantic meaning, swapping two operands can significantly change the instructions. To encode the inductive bias of position into our model, we introduce instruction position sequence $x_p$ and opcode/operand position sequence $x_o$ to represent the relative positions between the instructions and within each instruction. As shown in Figure 4, $x_p$ 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.

**Architecture sequence.** Finally, we feed the model with an extra sequence $x_a$, describing the input binary’s instruction set architecture. The vocabulary of $x_a$ consists of 4 architectures: $x_a = \{x86, x64, ARM, MIPS\}^n$. This setting helps the model to distinguish between the syntax of different architecture.

**Encoding numeric values.** As mentioned above, treating concrete values as independent tokens can lead to prohibitively large vocabulary size. We design a hierarchical input encoding scheme to address this challenge. Specifically, let $x_{ti}$ denote the $i$-th value in $x_t$. We represent $x_{ti}$ as an (padded) 8-byte fixed-length byte sequence $x_{ti} = [0x00, ..., 0xff]^8$ ordered in Big-Endian. We then feed $x_{ti}$ to a 2-layer bidirectional LSTM (bi-LSTM) and take its last hidden cell’s embedding as the value representation $t_i = bi-LSTM(x_{ti})$. Here $t_i$ denotes the output of applying the embedding to $x_{ti}$. 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. Figure 4 (right-hand side) illustrates how bi-LSTM takes the byte sequence and computes the embedding.

Such a design significantly reduces the vocabulary size as now we have only 256 possible byte values to encode. Moreover, bi-LSTM encodes the potential dependencies between high and low bytes within a single value. This setting thus supports learning better relationships between different dynamic values (e.g., memory region and offset) as opposed to treating all values as dummy tokens [71].

### C. Pretraining with Micro-traces

This section describes the pretraining task on the function binaries and their micro-traces, focusing on how the model masks its input $x$ (5 sequences) and predicts the masked part to learn the approximate execution semantics.

**Learned embeddings.** We embed each token in the 5 sequences with the same embedding dimension $d_{emb}$ so that they can be summed as a single sequence as input to the Transformer. Specifically, let $E_f(x_f)$, $E_i(x_i)$, $E_c(x_c)$, $E_o(x_o)$, $E_a(x_a)$ denote applying the embedding to the tokens in each sequence, respectively. We have $E_i$, the embedding of $x_i$:

$$E_i = E_f(x_{fi}) + E_i(x_{ti}) + E_c(x_{ci}) + E_o(x_{oi}) + E_a(x_{ai})$$

Here $x_{fi}$ denote the $i$-th token in $x_f$, where other sequence (i.e., $x_i$, $x_c$, $x_o$, $x_a$) follow the similar denotation. Note that $E_i(x_{i})$ leverages the bi-LSTM to encode the byte sequences (see Section IV-B), while the others simply multiplies the token (i.e., one-hot encoded [36]) with an embedding matrix. Figure 4 (right-hand side) illustrates the embedding strategies.

**Masked LM.** We pretrain the model using the masked LM objective. Formally, for each sequence ($x_{i1}, x_{i2}, ..., x_{in}$) in a given training set, we randomly mask out a selected percentage of tokens in each sequence. Specifically, we mask the code token and value token in $x_f$ and $x_i$, respectively, and replace them with a special token <MASK> (marked gray in Figure 4).
As masked LM trains on micro-traces without requiring additional labels, it is fully unsupervised.

Let \( m(E_t) \) denote the embedding of the masked token \( x_t \) and \( \text{MP} \) 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: \((E_1, ..., m(E_t), ..., E_n), i \in \text{MP}\), and predicts the code and the values of the masked tokens: \( \{\hat{x}_1, \hat{v}_i | i \in \text{MP}\} = g_p(E_1, ..., m(E_t), ..., E_n) \). Let \( g_p \) be parameterized by \( \theta \), the objective of training \( g_p \) is thus to search for \( \theta \) that minimizes the cross-entropy losses between (1) the predicted masked code tokens and the actual code tokens, and (2) predicted masked values (8 bytes) and the actual values. For ease of exposition, we omit summation over all samples in the training set.

\[
\arg\min_{\theta} \sum_{i=1}^{|	ext{MP}|} (-x_{f,i} \log(\hat{x}_{f,i}) + \alpha \sum_{j=1}^{8} -x_{t_{ij}} \log(\hat{x}_{t_{ij}})) \tag{1}
\]

\( \hat{x}_{t_{ij}} \) denotes the predicted \( j \)-th byte of \( x_{t_{i}} \) (the \( i \)-th token in \( x_t \)), \( \alpha \) is a hyperparameter that weights the cross-entropy losses between predicting code tokens and predicting values.

**Masking strategy.** For each chosen token to mask, we randomly choose from the masking window size from \( \{1, 3, 5\} \), which determines how many consecutive neighboring tokens of the chosen tokens are also masked [44]. For example, if we select \( x_5 \) (the 5-th token) in Figure 4 to mask and the masking window size is 3, \( x_4 \) and \( x_6 \) will be masked too. We then adjust accordingly to ensure the overall masked tokens still account for the initially-chosen masking percentage.

**Contextualized embeddings.** We employ the self-attention layers [75] to endow contextual information to each embedding \( E_t \) of the input token. Notably, let \( E_t = (E_{t,1}, ..., E_{t,n}) \) denote the embeddings produced by the \( t \)-th self-attention layer. We denote the embeddings before the model \( E_t \) as defined in Section IV-B as \( E_{0,t} \). Each token’s embedding at each layer will attend to all other embeddings, aggregate them, and update its embedding in the next layer. The embeddings after each self-attention layer are known as contextualized embeddings, which encodes the context-sensitive meaning of each token (e.g., \texttt{eax} in \texttt{mov eax, ebx} has different embedding with that in \texttt{jmp eax}). This is in contrast with static embeddings, e.g., word2vec commonly used in previous works [25], [26], where a code token is assigned to a fixed embedding regardless of the changed context.

The learned embeddings \( E_{t,i} \) after the last self-attention layer encodes the approximate execution semantics of each instruction and the overall function. In pretraining, \( E_{t,i} \) is used to predict the masked code. While in finetuning, it will be leveraged to match similar functions (Section IV-D).

**D. Finetuning for Function Similarity**

Given a function pair, we feed each function’s static code (instead of micro-trace as discussed in Section I) 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)}, ..., E_{k,n}^{(1)}) \text{ and } E_k^{(2)} = (E_{k,1}^{(2)}, ..., E_{k,n}^{(2)}) \text{ where } E_k^{(1)} \text{ corresponds to the first function and } E_k^{(2)} \text{ corresponds to the second. Let } y = \{-1, 1\} \text{ be the ground-truth indicating the similarity (1 - similar, -1 - dissimilar) between two functions. We stack a 2-layer Multi-layer Perceptrons } g_t, \text{ taking as input the average of embeddings for 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 average of last self-attention layers embeddings \( E_k \) with dimension \( d_{emb} \) into the function embedding with the dimension \( d_{func} \). \( d_{func} \) is often chosen smaller than \( d_{emb} \) to support efficient large-scale function searching [77]. Now let \( g_t \) be parameterized by \( \theta \), the finetuning objective is to minimize the cosine embedding loss \( l_{ce} \) between the ground-truth and the cosine distance between two function embeddings:

\[
\arg\min_{\theta} l_{ce}(g_t(E_k^{(1)}), g_t(E_k^{(2)}), y)
\]

where

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

\( \xi \) is the margin usually chosen between 0 and 0.5 [59]. As both \( g_p \) and \( g_t \) are neural nets, optimizing Equation 1 and Equation 2 can be guided by gradient descent via backpropagation.

After finetuning \( g_t \), the 2-layer multilayer perceptrons, and \( g_p \), the pre-trained model, we compute the function embedding \( f_{emb} = g_t(g_p(f)) \) and the similarity between two functions is measured by the cosine similarity between two function embedding vectors: \( \cos(f_{emb}^{(1)}, f_{emb}^{(2)}) \).

**V. IMPLEMENTATION AND EXPERIMENTAL SETUP**

We implement TREX using fairseq, a sequence modeling toolkit [55], 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 2.10GHz with 80 virtual cores including hyperthreading, 385GB RAM, and 8 Nvidia RTX 2080-Ti GPUs.

**Datasets.** To train and evaluate TREX, we collect 13 popular open-source software projects. These projects include Binutils-2.34, Coreutils-8.32, Curl-7.71.1, Diffutils-3.7, Findutils-4.7.0, GMP-6.2.0, ImageMagick-7.0.10, Libmicrohttpd-0.9.71, LibTomCrypt-1.18.2, OpenSSL-1.0.1f and OpenSSL-1.0.1u, PuTTY-0.74, SQLite-3.34.0, and Zlib-1.2.11. We compile these projects into 4 architectures i.e., x86, x64, ARM (32-bit), and MIPS (32-bit), with 4 optimization levels (OPT), i.e., O0, O1, O2, and O3, using GCC-7.5. Specifically, we compile the software projects based on its makefile, by specifying \texttt{CFLAGS} (to set optimization flag), \texttt{CC} (to set cross-compiler), and --host (to set the cross-compilation target architecture). We always compile to dynamic shared objects, but resort to static linking when we encounter build errors. We are able to compile all projects with these treatments.
We also obfuscate all projects using 5 types of obfuscations (OBF) by Hikari [78] on x64 – an obfuscator based on clang-8. The obfuscations include bogus control flow (bcf), control flow flattening (cff), register-based indirect branching (ibr), basic block splitting (spl), and instruction substitution (sub). As we encounter several errors in cross-compilation using Hikari (based on Clang) [78], and the baseline system (i.e., Asm2Vec [25]) 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, as shown in Table I.

### Micro-tracing
We implement micro-tracing by Unicorn [66], a cross-platform CPU emulator based on QEMU [8]. We micro-execute each function 3 times with different initialized registers and memory, generating 3 micro-traces (including both static code and dynamic values) for pretraining (masked LM). We leverage multi-processing to parallelize micro-executing each function and set 30 seconds as the timeout for each run in case any instruction gets stuck (i.e., infinite loops). For each function (with 3 micro-traces), we append one additional dummy trace, which consists of only dummy values (###). This setting encourages the model to leverage its learned execution semantics (from other traces with concrete dynamic values) to predict the masked code with only code context, which helps the finetuning task as we only feed the functions’ static code as input (discussed in Section I).

### Baselines
For comparing cross-architecture performance, we consider 2 state-of-the-art systems. The first one is SAFE [50] that achieves state-of-the-art function matching accuracy. As SAFE’s trained model is publicly available, we compare TREX with both SAFE’s reported results on their dataset, i.e., OpenSSL-1.0.1f and OpenSSL-1.0.1u, and running their trained models on all of our collected binaries. The second baseline is Gemini [77], which is shown outperformed by SAFE. As Gemini does not release their trained models, we compare our results to their reported numbers directly on their evaluated dataset, i.e., OpenSSL-1.0.1f and OpenSSL-1.0.1u.

For cross-optimization/obfuscation comparison, we consider Asm2Vec [25] and Blex [27] 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 vs. 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, there has to be a mechanism to threshold the similarity score to determine whether the function pairs are similar or dissimilar. The chosen thresholds largely determine the model’s predictive performance [57]. To avoid the potential bias introduced by a specific threshold value, 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. Specifically, given a sequence of query functions and the target functions to be matched, Precision@1 measures the percentage of matched query functions in the target functions. Here “match” means the query function should have the top-1 highest similarity score with the ground truth function among the target functions.

To evaluate pretraining performance, we use the standard metric for evaluating the language model – perplexity (PPL). The lower the PPL the better the pretrained model in predicting masked code. The lowest PPL is 1.

**Pretraining setup.** To strictly separate the functions in pretraining, finetuning, and testing, we pretrain TREX on all functions in the dataset, except the functions in the project going to be finetuned and evaluated for similarity. For example, the function matching results of Binutils (Table II) are finetuned on the model pretrained on all other projects without Binutils. Note that pretraining is agnostic to any labels (e.g., ground-truth indicating similar functions). Therefore, we can always pretrain on large-scale codebases, which can potentially include the functions for finetuning (this is the common practice in transfer learning [22]). It is thus worth noting that our setup of separating functions for pretraining and finetuning makes our testing significantly more challenging.

We keep the pretrained model weights that achieve the best validation PPL for finetuning. The validation set for pretraining consists of 10,000 random functions selected from Table I.

**Finetuning setup.** We choose 50,000 random function pairs for each project and randomly select only 10% for training, and the remaining is used as the testing set. We keep the training and testing functions strictly non-duplicated by ensuring the functions that appear in training function pairs not appear in the testing. As opposed to the typical train-test split (e.g., 80% training and 20% testing [50]), our setting requires the model to generalize from few training samples to a large number of unseen testing samples, which alleviates the possibility of overfitting. Moreover, we keep the ratio between similar and dissimilar function pairs in the finetuning set as roughly 1:5. This setting follows the practice of contrastive learning [15], [70], respecting the actual distribution of similar/dissimilar functions as the number of dissimilar functions is often larger than that of similar functions in practice.

**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 fix the largest input length to be 512 and split the functions longer than this length into subsequences for pretraining. We average the subsequences’ embeddings during finetuning if the function is split to more than one subsequences. In this paper, we keep most of the hyperparameters fixed throughout the evaluation if not mentioned explicitly (complete hyperparameters are defined in Appendix B). While we can always search for better hyperparameters, there is no principled method to date [9]. We thus leave as future work a more thorough study of TREX’s hyperparameters.

**VI. Evaluation**

Our evaluation aims to answer the following questions.

- **RQ1**: How accurate is TREX in matching semantically similar function binaries 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 micro-traces help improve the accuracy of matching functions?

### A. RQ1: Accuracy

We evaluate how accurate TREX is in matching similar functions across different architectures, optimizations, and obfuscations. As shown in Table II, we prepare function pairs for each project (first column) with 5 types of partitions. (1) ARCH: the function pairs have different architectures but same optimizations without obfuscations (2nd column). (2) OPT: the function pairs have different optimizations but same architectures without obfuscations (3rd column). (3) OBF: the function pairs have different obfuscations with same architectures (x64) and no optimization (4th column). (4) ARCH+OPT: the function pairs have both different architectures and optimizations without obfuscations (5th column). (5) ARCH+OPT+OBF: the function pairs can come from arbitrary architectures, optimizations, and obfuscations (6th column).

Table II reports the mean testing AUC scores of TREX on each project with 3 runs. 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.

On matching functions with different obfuscations, TREX’s results are 0.026, on average, lower than that of cross-optimizations, which indicates the obfuscation changes the code more drastically. Section VI-B will show the specific results of TREX on each obfuscations.

B. RQ2: Baseline Comparison

Cross-architecture search. As described in Section V, we first compare TREX with SAFE and Gemini on OpenSSL-1.0.1f and OpenSSL-1.0.1u with their reported numbers (as they only evaluate on these two projects). We then run SAFE’s released model on our dataset and compare to TREX.

We use our testing setup (see Section V) to evaluate SAFE’s trained model, where 90% of the total functions of those in Table I are used to construct the testing set. These testing sets are much larger than that in SAFE, where they only evaluate on 20% of the OpenSSL functions. Note that the dataset used in SAFE are all compiled by GCC-5.4 at the time when it is publicized (November 2018), while ours are compiled by GCC-7.5 (April 2020). We find these factors (the more diverse dataset and different compilers) can all lead to the possible dataset distribution shift, which often results in the decaying performance of ML models when applied in the security applications [43].

To study the distribution shift, we measure the KL-divergence [48] between SAFE’s dataset (OpenSSL compiled by GCC-5.4) and our dataset. Specifically, we use the probability distribution of the raw bytes of the compiled projects’ binaries, and compute their KL-divergence between SAFE and our. As OpenSSL is also a subset of our complete dataset, we compute the KL-divergence between our compiled OpenSSL and that of SAFE as the baseline.

We find the KL-divergence is 0.02 between SAFE’s dataset and ours, while it decreases to 0.0066 between our compiled OpenSSL and that of SAFE. This indicates that the underlying distribution of our test set shifts from that of SAFE’s. Moreover, the KL-divergence of 0.0066 between the same dataset (OpenSSL) but only compiled by different GCC versions implies that the compiler version has much smaller effect to the distribution shift than the different software projects.

As shown in Figure 5, our ROC curve is higher than those reported in SAFE and Gemini. While SAFE’s reported AUC score, 0.992, is close to ours, when we run their trained model on our testing set, its AUC score drops to 0.976 – possibly because our testing set is much larger than theirs. This observation demonstrates the generalizability of TREX– when pretrained to approximately learn execution semantics explicitly, it can quickly generalize to match unseen (semantically similar) functions with only a minimal training set.

Figure 6 shows that TREX consistently outperforms SAFE on all projects, i.e., by 7.3% on average. As the SAFE’s model is only trained on OpenSSL, we also follow the same setting by training TREX on only OpenSSL, similar to the cross-project setting described in Section VI-A.

Cross-optimization search. 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 (discussed in Section V), we compare with each tool separately using their metrics and on their evaluated dataset.

Table III shows TREX outperforms Asm2Vec in Precision@1 (by 7.2% on average) on functions compiled by different optimizations (i.e., between O2 and O3 and between O0 and O3). As the syntactic difference introduced by optimizations between O0 and O3 is more significant than that between O0 and O3, both tools have certain level of decrease in AUC scores (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 [79]. As Blex show the matched functions of each individual utility in Coreutils in a bar chart without including the concrete numbers of matched functions, we estimate their matched functions using their reported average percentage (75%) on all utilities.

Figure 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 search. We compare TreX to Asm2Vec on matching obfuscated function binaries with different obfuscation methods. Notably, Asm2Vec is evaluated on obfuscations including bogus control flow (ccf), control flow flattening (ccf), and instruction substitution (sub), which are subset of our evaluated obfuscations (Table I). As Asm2Vec only evaluates on 4 projects, i.e., GMP, ImageMagic, LibTomCrypt, and OpenSSL, we focus on these 4 projects and TreX’s results for other projects are included in Table II.

Table IV shows TreX achieves better Precision@1 score (by 14.3% on average) throughout all 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 example, instruction substitution simply replaces very limited instructions (i.e., arithmetic operations as shown in Section II) while control flow flattening dramatically changes the function code. Asm2Vec has 12.2% decreased score when the obfuscation is changed from sub to ccf, while TreX only decreases by 4%.

C. RQ3: Execution Time

We evaluate the runtime performance of generating function embeddings for computing similarity. We compare TreX with SAFE and Gemini on generating functions in 4 projects on x64 compiled by O3, i.e., Binutils, Putty, Findutils, and Diffutils, which have disparate total number of functions (see Table I). This tests how TreX scales to different number of functions compared to other baselines. 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 training time. Moreover, the output of all tools are function embeddings, which can be indexed and efficiently searched using locality sensitive hashing (LSH) [33], [67]. Therefore, we do not compare the matching time of function embeddings as it simply depends on the performance of 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 takes the parsed function binary as input and computes function embedding. We test the embedding generation using our GPU (see Section V).

Figure 8 shows that TreX is more efficient than the other tools in both function parsing and embedding generation for functions from 4 different projects with different number of functions (Table I). Gemini requires manually constructing control flow graph and extracting inter-/intra-basic-block feature.
engineering. It thus incurs the largest overhead. For generating function embeddings, our underlying network architectures leverage Transformer self-attention layers, which is more amenable to parallelization with GPU than the recurrent (used by SAFE) and graph neural network (used by Gemini) [75]. As a result, REX runs up to 8× faster than SAFE and Gemini.

D. RQ4: Ablation Study

In this section, we aim to quantify how much each key component in REX’s design helps the end results. We first study how much does pretraining, argued to assist learning approximate execution semantics, help match function binaries. We then study how does pretraining without micro-traces affect the end results. We also test how much does incorporating the micro-traces in the pretraining tasks improve the accuracy.

Pretraining effectiveness. We compare the testing between AUC scores achieved by REX (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 weights that are not pretrained). The function pairs can come from arbitrary architectures, optimizations, and obfuscations.

Figure 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 33% of the functions compared to pretraining with all functions. This indicates that we can potentially decrease our pretraining data size to achieve roughly the same performance. However, since our pretraining task does not require collect any label, we can still collect unlimited binary code found in the wild to enrich the semantics that can be observed.

Pretraining w/o micro-traces. We try to understand whether including the micro-traces in pretraining can really 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 IV), 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). We also compare to SAFE in this case.

Figure 10 shows that the AUC scores decrease by 7.2% when the model is pretrained without micro-trace. The AUC score of pretrained REX without micro-traces is 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 micro-traces has less performance drop than the model simply not pretrained (7.2% vs. 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.

Pretraining accuracy. We study the testing perplexity (PPL defined in Section V) of pretraining REX to directly validate whether it indeed helps REX to learn the approximate execution semantics. The rationale is the good performance, i.e., the low PPL, on unseen testing function binaries indicates that REX highly likely learns to generalize based on its learned approximate execution semantics.

The green line in Figure 11 shows the validation PPL in 10 epochs of pretraining REX. The testing set is constructed by sampling 10,000 random functions from the projects used in pretraining (as described in Section V). We observe that the PPL of REX drops to close to 2.1, which is far below that of random guessing (e.g., random guessing PPL is $2^{-\log(1/256)} = 256$).
indicating that TReX has around 0.48 confidence on average on the masked tokens being the correct value. Note that a random guessing has only 1/256=0.004 confidence on the masked tokens being the correct value.

VII. CASE STUDIES

In this section, we study how TReX can help discover new vulnerabilities in a large volume of latest firmware images.

Firmware images often include third-party libraries (i.e., BusyBox, OpenSSL). However, these libraries are frequently patched, but the manufacturers often fall behind to update them accordingly [56]. Therefore, we study whether our tool can uncover function binaries in firmware images similar to known vulnerable functions. We find existing state-of-the-art binary similarity tools (i.e., SAFE [50], Asm2Vec [25], Gemini [77]) all perform their case studies on the firmware images and vulnerabilities that have already been studied before them [14], [17], [19], [30]. 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 new (1-day) vulnerabilities in most recent firmware images that are not disclosed before.

We crawl firmware images (in their latest version) in 180 products including WLAN routers, smart cameras, and solar panels, from well-known manufacturers’ official releases and third-party providers such as DD-WRT [35], as shown in Table VIII. We collect firmware images with only the latest version, which are much more updated than those studied in the state-of-the-art (dated before 2015, which are all patched before their studies). For each function in the firmware images, we construct function embedding and build a firmware image database using Open Distro for Elasticsearch [3], which supports vector-based indexing with efficient search support based on NMSLIB [11].

We extract the firmware images using binwalk [23]. In total, we collect 180 number of firmware images from 22 vendors. These firmware images are developed by the original vendors, or from third-party providers such as DD-WRT [35]. Most of them are for WLAN routers, while some are deployed in other embedded systems, such as solar panels and smart cameras. Among the firmware images, 127 of them are compiled for MIPS 32-bit and 53 are compiled for ARM 32-bit. 169 of them are 2020 models, and the rest are 2019 and 2018 models.

Table V shows 16 vulnerabilities (CVEs) we use to search in the firmware images. We focus on the CVEs of OpenSSL and BusyBox, as they are widely included in the firmware. For each CVE, we compile the corresponding vulnerable functions in the specified library version and computes the vulnerability function embeddings via TReX. As the firmware images are stripped so that 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 capability in cross-architecture and optimization function matching to match functions that are potentially compiled in different architectures and with different optimizations. We then obtain the firmware functions that rank top-10 similar to the vulnerable function and manually verify if they are vulnerable. We leverage strings command to identify the OpenSSL and BusyBox 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. As shown in Table V, we have confirmed all 16 CVEs in 4 firmware models 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. Table VI shows the details of the 16 vulnerabilities that TReX uncover in the 4 firmware images shown in Table V. The description of “allow denial-of-service” usually refers to the segmentation fault that crashes the program. The cause of such error can be diverse, which is not due to the other typical causes shown in the table (e.g., integer overflow, buffer over-read, etc.).
VIII. RELATED WORK

A. Binary Similarity

Traditional approaches. Existing static approaches often extract hand-crafted features by domain experts to match similar functions. The features often encode the functions’ syntactic characteristics. For example, BinDiff [79] extracts the number of basic blocks and the number of function calls to determine the similarity. Other works [18], [29], [47], [54] introduce more carefully-selected static features such as n-gram of instruction sequences. Another popular approach is to compute the structural distance between functions to determine the similarity [10], [20], [21], [28], [40], [65], [79]. For example, BinDiff [79] performs graph matching between functions’ call graphs, TEDEM [65] matches the basic block expression trees, BinSequence [40] and Tracelet [21] uses the edit distance between functions’ instruction sequence. As discussed in Section I, both static features and structures are susceptible to obfuscations and optimizations and incur high overhead. TREX automates learning approximate execution semantics without any manual effort, and the execution semantics is more robust to match semantically similar functions.

In addition to the static approaches, dynamic approaches such as iLine [41], BinHunt [52], iBinHunt [52], Revolver [45], Blex [27], Rieck et al. [69], Multi-MH [64], BinGo [13], ESH [19], BinSim [53], CACompare [39], Vulseeker-pro [32], and Tinbergen [51] construct hand-coded dynamic features, such as values written to stack/heap [27] or system calls [53] by executing the function to match similar functions. These approaches can detect semantically similar (but syntactically different) functions by observing their similar execution behavior. However, as discussed in Section I, these approaches can be expensive and can suffer from false positives due to the under-constrained dynamic execution traces [25], [42]. By contrast, we only use these traces to learn approximate execution semantics of individual instructions and transfer the learned knowledge to match similar functions without directly comparing their dynamic traces. Therefore, we are much more efficient and less susceptible to the imprecision introduced by these under-constrained dynamic traces.

Learning-based approaches. Most recent learning-based works such as Genius [30], Gemini [77], Asm2Vec [25], SAFE [50], DeepBinDiff [26] learn a function representation that is supposed to encode the function syntax and semantics in low dimensional vectors, known as function embeddings. The embeddings are constructed by learning a neural network that takes the functions’ structures (control flow graph) [26], [30], [77] or instruction sequences [25], [50] as input and train the model 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 learning architectures adopted in these approaches require constructing expensive graph features (attributed CFG [30], [77]) or limited in modeling long-range dependencies (based on Word2Vec [25], [26]). By contrast, TREX learns approximate execution semantics to match functions. Its underlying architecture is amenable to learning long-range dependencies in sequences without heavyweight feature engineering or graph construction.

B. Learning Program Representations

There has been a growing interest in learning neural program representation as embeddings from “Big Code” [2]. The learned embedding of the code encodes the program’s key properties (i.e., semantics, syntax), which can be leveraged to perform many applications beyond function similarity, such as program repair [58], [76], recovering symbol names and types [16], [60], code completion [68], decompilation [31], [46], prefetching [71], and many others that we refer to Allamanis et al. [2] for a more thorough list. Among these works, the most closest work to us is XDA [62], which also leverages the transfer learning to learn general program representations for recovering function and instruction boundaries. However, XDA only learns from static code at the raw byte level, which lacks the understanding of execution semantics.

The core technique proposed in this paper – learning approximate execution semantics from micro-traces – is by no means limited to only function similarity task but can be applied to any of the above tasks. Indeed, we plan to explore how the learned semantics in our model can transfer to other (binary) program analysis tasks in our future work.

IX. CONCLUSION

We introduced TREX to match semantically similar functions based on the function execution semantics. Our key insight is to first pretrain the ML model to explicitly learn approximate execution semantics based on the functions’ micro-traces and then transfer the learned knowledge to match semantically similar functions. Our evaluation showed that the learned approximate execution semantics drastically improves the accuracy of matching semantically similar functions – TREX excels in matching functions across different architectures, optimizations, and obfuscations. We plan to explore in our future work how the learned execution semantics of the code can further boost the performance of broader (binary) program analysis tasks such as decompilation.

ACKNOWLEDGMENT

We thank our shepherd Lorenzo Cavallaro and the anonymous reviewers for their constructive and valuable feedback. This work is sponsored in part by NSF 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; ONR grants N00014-17-1-2010, N00014-16-1-2263, and N00014-17-1-2788; an NSF CAREER award; an ARL Young Investigator (YIP) award; a Google Faculty Fellowship; a JP Morgan Faculty Research Award; a DiDi Faculty Research Award; a Google Cloud grant; a Capital One Research Grant; and an Amazon Web Services grant. 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, Capital One, Google, JP Morgan, DiDi, or Amazon.
A. Detailed Model Architecture

Multi-head self-attention. Given the embeddings of all input token (see Section IV-C) in $i$-th layer, $(E_{t, 1}, \ldots, E_{t,n})$, the self-attention layer updates each embedding with the following steps. It first maps $E_{t,i}$ to three embeddings, i.e., query embedding $q_i$, key embedding $k_i$, and value embedding $v_i$:  

$$q_i = f_q(W_q; E_{t,i}); \quad k_i = f_k(W_k; E_{t,i}); \quad v_i = f_v(W_v; E_{t,i})$$

Here $f_q$, $f_k$, and $f_v$ are affine transformation functions (i.e., a fully-connected layer) parameterized by $W_q$, $W_k$, and $W_v$, respectively. It then computes attention $a_{ij}$ between $E_{t,i}$ and all other embeddings $E_{t,j} (j \neq i)$ by taking the dot product between the $E_{t,i}$’s query embedding $q_i$ and $E_{t,j}$’s key embedding $k_j$:  

$$a_{ij} = q_i \cdot k_j.$$  

Intuitively, attention $a$ is a square matrix, where each cell $a_{ij}$ indicates how much attention $E_{t,i}$ should pay to $E_{t,j}$ when updating itself. It then divides every row of $a$ by $\sqrt{d_{emb}}$ (the dimension of the embedding vectors) and scale it by softmax to ensure them sum up to 1:  

$$a'_{ij} = \frac{\exp(a_{ij})}{\sum_{j=1}^{n} \exp(a_{ij})}$$

The scaled attention $a'_{ij}$ will be multiplied with value embedding $v_j$ and summed up:  

$$E_{t,i}^{h} = \sum_{j=1}^{n} a'_{ij}v_j$$

Here $h$ in $E_{t,i}^{h}$ denote the updated embedding belong to attention head $h$. Assume we have total $H$ attention heads, the updated embeddings will finally go through an 2-layer feedforward network $f_{out}$ parameterized by $W_{out}$ with skip connections [37] to update embeddings from all heads:  

$$E_{t,i}^{h'} = f_{out}(E_{t,i}^{h}) + E_{t,i}^{h}.$$

APPENDIX
How to use the embedding. The learned embeddings $E_{l,i}$ after the last self-attention layer encodes the execution semantics of each instruction and the overall function. Consider predicting masked input in pretraining. Let layer $l$ be the $g_p$'s last self-attention layer. The model will stack a 2-layer multi-layer perceptron (MLP) for predicting the masked codes and values:

$$
MLP(E_{l,i}) = \text{softmax}(\tanh(E_{l,i} \cdot W_1) \cdot W_2), i \in \text{MIP}
$$

$W_1 \in \mathbb{R}^{d_{emb} \times d_{emb}}$ and $W_2 \in \mathbb{R}^{d_{emb} \times |V|}$ where $|V|$ is the vocabulary size of the code token or bytes. As shown in Equation 1, each embedding will have 9 stacked MLPs (i.e., one for predicting code and the rest for predicting bytes).

B. T.REX Hyperparameters

Network architecture. We use 12 self-attention layers with each having 8 self-attention heads. The embedding dimension $d_{emb} = d_{func} = 768$, which is also the embedding dimension used in bi-LSTM. We set 3072 as the hidden layer size of MLP in the self-attention layer. We adopt GeLU [38], known for addressing the problem of vanishing gradient, as the activation function for T.REX's self-attention module. We use the hyperbolic tangent (tanh) as the activation function in finetuning MLP. We set the dropout rate 0.1 for pretraining do not use dropout in finetuning.

Pretraining. We fix the largest input length to 512 and choose the effective batch size for both pretraining and finetuning as 512. As 512 batch size with each sample of length 512 is too large to fit in our GPU memory (11 GB), we aggregate the gradient every 64 batches and then updates the weight parameter. This setting results in the actual batch size as 8 ($64 \times 8 = 512$). We pick learning rate $5 \times 10^{-4}$ for pretraining. Instead of starting with the chosen learning rate at first epoch, we follow the common practice of using small warmup learning rate at first epoch. We use $10^{-7}$ as the initial warmup learning rate and gradually increases it until reaching the actual learning rate ($5 \times 10^{-4}$) after first epoch. We use use Adam optimizer, with $\beta_1 = 0.9$, $\beta_2 = 0.98$, $\epsilon = 10^{-6}$, and weight decay $10^{-2}$.

C. More Experiments

Cross-project generalizability. While we strictly separate the functions for pretraining, finetuning, and testing, the functions of finetuning and testing can come from the same project (but strictly different functions). Therefore, in this section, we further separate the functions in finetuning and testing by extracting them from different projects. For example, we can finetune the model on Coreutils while testing on OpenSSL. Specifically, we select 3 projects, i.e., Binutils, Coreutils, and OpenSSL, which have the largest number of functions, and evaluate how T.REX performs. We allow the functions to come from different architectures, optimizations, and obfuscations, and follow the same setup as described in Section V.

Vulnerability search performance. We quantify the accuracy of T.REX 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 V. Specifically, we compile OpenSSL to ARM and x64 with O3, and feed both our compiled and firmware’s function binaries to T.REX and SAFE to compute embeddings. Based on the function embeddings, we search the compiled OpenSSL functions in the NETGEAR R7000’s embedded OpenSSL 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.

Figure 12 shows that T.REX consistently outperforms SAFE,
achieving 24.3% lower error rate on average. Moreover, as shown in Figure 12b, when the query functions are from x64 (the firmware binaries are from ARM), T Rex outperforms SAFE by a greater margin, achieving 25.3% lower error rate in cross-architecture search.

D. Probing Learned Execution Semantics

As discussed in Section II, training the model to predict masked micro-trace codes and values compels the model to learn execution semantics by concrete examples. It thus automates the extraction of the code’s dynamic features without manual effort. In this section, we study concrete code examples showing the potential hint in the code that the model likely leverages to predict the masked part.

Predicting arithmetic instructions. Consider the example in Figure 13, which shows the function’s micro-trace (i.e., the dynamic value and assembly code). For ease of exposition, the format of micro-trace values (e.g., the value of opcode and constants) are not exactly the same as the actual format we handle, as described in Section IV-B. We mask the register rax and opcode add and their corresponding values in a sequence of arithmetic instructions.

To correctly predict the masked value 0x4c0469a of rax, the model should understand the approximate execution semantics of sub in line 2, which subtracts rax with rcx. It may also observe the value of rax at line 4, which exactly equals the result of adding rax with 7 at line 3. Therefore, we see the model predicts rax with 96% confidence and rcx with only 3%. It also predicts the value of rax correctly (right table). To correctly predict add at line 4, the model should observe that the rax at line 5 has the same exact value with the result of rax+rcx at line 4, so it predicts the opcode at line 4 to be add. As shown in Figure 13, our pretrained model predicts add with 98% confidence and mov with only 1%, which implies that the model approximately understands the execution semantics of add and mov so it predicts that add is much more likely.

Predicting stack operations. Consider the example in Figure 14. We mask the register and constant of the instruction in function epilogue – it increments the stack pointer rsp by 0x20 to deallocate the local variable stored on stack. To correctly predict the masked rsp and its value 0x8b4a3f, the model should observe the rsp is decremented (due to opcode sub) by 0x20 from 0x8b4a3f at line 3, which is part of the function prologue. Therefore, the model should understand the execution semantics of sub and basic syntax of function prologue and epilogue.

To predict the masked constant 0x20, the model should notice that line 3 decrements the stack pointer by 0x20, which is the size of local variables. As the model predicts 0x20 with 99% confidence, this implies the model likely learns patterns of function prologue and epilogue, including the context such as push ebp, pop ebp, etc. This observation indicates that our pretraining task assists in learning common function idiom (function calling convention) beyond execution semantics of individual instructions. Learning such knowledge can potentially help other tasks beyond function similarity, such as identifying function boundaries [62].
<table>
<thead>
<tr>
<th>Vendor</th>
<th>Model</th>
<th>ISA</th>
<th>Year</th>
<th>Vendor</th>
<th>Model</th>
<th>ISA</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>8devices</td>
<td>Carambola 2</td>
<td>MIPS</td>
<td>2020</td>
<td>8devices</td>
<td>Carambola 2 8MB</td>
<td>MIPS</td>
<td>2018</td>
</tr>
<tr>
<td>Actiontec</td>
<td>MI424WR</td>
<td>ARM</td>
<td>2019</td>
<td>Alfa</td>
<td>AIP-W502U</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>Belkin</td>
<td>F5D8235-4</td>
<td>MIPS</td>
<td>2020</td>
<td>Buffalo</td>
<td>BHR-4GRV</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>Buffalo</td>
<td>WHR-1166D</td>
<td>ARM</td>
<td>2020</td>
<td>Buffalo</td>
<td>WHR-300HP2</td>
<td>MIPS</td>
<td>2018</td>
</tr>
<tr>
<td>Buffalo</td>
<td>WXR-1900DHP</td>
<td>ARM</td>
<td>2020</td>
<td>Buffalo</td>
<td>WZR-1166DHP</td>
<td>ARM</td>
<td>2020</td>
</tr>
<tr>
<td>Buffalo</td>
<td>WZR-9000HP</td>
<td>ARM</td>
<td>2020</td>
<td>Buffalo</td>
<td>WZR-HP-G450H</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>Comfast</td>
<td>CF-E355AC</td>
<td>MIPS</td>
<td>2020</td>
<td>Comfast</td>
<td>CF-E380AC</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>Compart</td>
<td>WPS-46</td>
<td>MIPS</td>
<td>2020</td>
<td>D-Link</td>
<td>DAP-2330</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>D-Link</td>
<td>DAP-3662</td>
<td>MIPS</td>
<td>2020</td>
<td>D-Link</td>
<td>DAP-1565 A1</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>D-Link</td>
<td>DIR-868i Rev.C</td>
<td>ARM</td>
<td>2020</td>
<td>D-Link</td>
<td>DIR-869</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>D-Link</td>
<td>DIR-880i</td>
<td>ARM</td>
<td>2020</td>
<td>D-Link</td>
<td>DIR-882 A1</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>D-Link</td>
<td>DIR-890i</td>
<td>ARM</td>
<td>2018</td>
<td>Gateworks</td>
<td>GW2382</td>
<td>ARM</td>
<td>2018</td>
</tr>
<tr>
<td>Gateworks</td>
<td>LG400</td>
<td>MIPS</td>
<td>2020</td>
<td>Gateworks</td>
<td>LG400</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>Gigaset</td>
<td>SX763</td>
<td>MIPS</td>
<td>2020</td>
<td>JSP</td>
<td>JA76PF</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>Linksys</td>
<td>E2100L</td>
<td>MIPS</td>
<td>2020</td>
<td>Linksys</td>
<td>EA6350</td>
<td>ARM</td>
<td>2020</td>
</tr>
<tr>
<td>Linksys</td>
<td>EA6500 v2</td>
<td>MIPS</td>
<td>2020</td>
<td>Linksys</td>
<td>EA6700</td>
<td>ARM</td>
<td>2020</td>
</tr>
<tr>
<td>Linksys</td>
<td>EA8500</td>
<td>MIPS</td>
<td>2020</td>
<td>Linksys</td>
<td>RE7000</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>NETGEAR</td>
<td>AC1450</td>
<td>ARM</td>
<td>2018</td>
<td>Linksys</td>
<td>EX6200</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>NETGEAR</td>
<td>R6300v2</td>
<td>ARM</td>
<td>2020</td>
<td>Linksys</td>
<td>R6400</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>NETGEAR</td>
<td>R6700v3</td>
<td>ARM</td>
<td>2020</td>
<td>Linksys</td>
<td>R7000</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>NETGEAR</td>
<td>R7500v1</td>
<td>ARM</td>
<td>2020</td>
<td>Linksys</td>
<td>R7500v2</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>NETGEAR</td>
<td>R8500</td>
<td>ARM</td>
<td>2020</td>
<td>Linksys</td>
<td>R8900</td>
<td>ARM</td>
<td>2020</td>
</tr>
<tr>
<td>NETGEAR</td>
<td>WG302v2</td>
<td>ARM</td>
<td>2020</td>
<td>Linksys</td>
<td>WNDR3700v4</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>NETGEAR</td>
<td>WNDR4300</td>
<td>MIPS</td>
<td>2020</td>
<td>Linksys</td>
<td>WNDR4500v2</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>NETGEAR</td>
<td>XR450</td>
<td>ARM</td>
<td>2020</td>
<td>Linksys</td>
<td>XR700</td>
<td>ARM</td>
<td>2020</td>
</tr>
<tr>
<td>Senao</td>
<td>ECB3500</td>
<td>MIPS</td>
<td>2020</td>
<td>Senao</td>
<td>ECB9750</td>
<td>MIPS</td>
<td>2019</td>
</tr>
<tr>
<td>Senao</td>
<td>EXC510</td>
<td>MIPS</td>
<td>2020</td>
<td>Senao</td>
<td>EXC510</td>
<td>MIPS</td>
<td>2019</td>
</tr>
<tr>
<td>Senao</td>
<td>NOP8670</td>
<td>ARM</td>
<td>2020</td>
<td>Archer</td>
<td>Archer A7 V5</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>TP-Link</td>
<td>Archer C25 V1</td>
<td>MIPS</td>
<td>2020</td>
<td>TP-Link</td>
<td>Archer C5 V1</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>TP-Link</td>
<td>Archer C7 V2</td>
<td>MIPS</td>
<td>2020</td>
<td>TP-Link</td>
<td>Archer C7 V3</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>TP-Link</td>
<td>Archer C8 V1</td>
<td>ARM</td>
<td>2020</td>
<td>TP-Link</td>
<td>Archer C9 V1</td>
<td>ARM</td>
<td>2020</td>
</tr>
<tr>
<td>TP-Link</td>
<td>Archer C9 V3</td>
<td>ARM</td>
<td>2020</td>
<td>TP-Link</td>
<td>Deco-M4</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>TP-Link</td>
<td>TL-WDR3300 V1</td>
<td>MIPS</td>
<td>2020</td>
<td>TP-Link</td>
<td>TL-WDR4310 V1</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>TP-Link</td>
<td>TL-WR1043N V5</td>
<td>MIPS</td>
<td>2020</td>
<td>TP-Link</td>
<td>TL-WR1043ND</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>TP-Link</td>
<td>TL-WR1043N D4</td>
<td>MIPS</td>
<td>2020</td>
<td>TP-Link</td>
<td>TL-WR1043ND</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>TP-Link</td>
<td>TL-WR10701N V2</td>
<td>MIPS</td>
<td>2020</td>
<td>TP-Link</td>
<td>TL-WR10701N V2</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>TP-Link</td>
<td>TL-WR842N V1</td>
<td>MIPS</td>
<td>2018</td>
<td>TP-Link</td>
<td>TL-WR842N V2</td>
<td>MIPS</td>
<td>2018</td>
</tr>
<tr>
<td>TRENDSnet</td>
<td>TEW-812DRU V2</td>
<td>ARM</td>
<td>2018</td>
<td>TRENDSnet</td>
<td>TEW-812DRU</td>
<td>ARM</td>
<td>2018</td>
</tr>
<tr>
<td>Ubiquiti</td>
<td>AirGrid M2</td>
<td>MIPS</td>
<td>2020</td>
<td>Ubiquiti</td>
<td>AirGrid M5</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>Ubiquiti</td>
<td>AirRouter</td>
<td>MIPS</td>
<td>2020</td>
<td>Ubiquiti</td>
<td>AirRouter-HP</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>Ubiquiti</td>
<td>BulletM2 HP</td>
<td>MIPS</td>
<td>2020</td>
<td>Ubiquiti</td>
<td>BulletM5 HP</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>Ubiquiti</td>
<td>NanoBeam AC</td>
<td>MIPS</td>
<td>2020</td>
<td>Ubiquiti</td>
<td>NanoBeam M2 XW</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>Ubiquiti</td>
<td>NanoBridge M2</td>
<td>MIPS</td>
<td>2020</td>
<td>Ubiquiti</td>
<td>NanoBridge M2 XW</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>Ubiquiti</td>
<td>NanoBridge M35</td>
<td>MIPS</td>
<td>2020</td>
<td>Ubiquiti</td>
<td>NanoBridge M5 XW</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>Ubiquiti</td>
<td>NanoBridge M36</td>
<td>MIPS</td>
<td>2020</td>
<td>Ubiquiti</td>
<td>NanoBridge M5 XW</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>Ubiquiti</td>
<td>NanoStation M2</td>
<td>MIPS</td>
<td>2020</td>
<td>Ubiquiti</td>
<td>Power AP N</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>Ubiquiti</td>
<td>Pico M5</td>
<td>MIPS</td>
<td>2020</td>
<td>Ubiquiti</td>
<td>PowerBeam M5 M400 XW</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>Ubiquiti</td>
<td>PowerBridge M10</td>
<td>MIPS</td>
<td>2020</td>
<td>Ubiquiti</td>
<td>PowerBridge M5</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>Ubiquiti</td>
<td>Rocket M2 XW</td>
<td>MIPS</td>
<td>2020</td>
<td>Ubiquiti</td>
<td>Rocket M5 X3 XW</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>Ubiquiti</td>
<td>Rocket M3 XW</td>
<td>MIPS</td>
<td>2020</td>
<td>Ubiquiti</td>
<td>Rocket M3</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>Ubiquiti</td>
<td>RocketM65</td>
<td>MIPS</td>
<td>2020</td>
<td>Ubiquiti</td>
<td>RocketM5</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>Ubiquiti</td>
<td>RouterStation</td>
<td>MIPS</td>
<td>2018</td>
<td>Ubiquiti</td>
<td>RouterStation Pro</td>
<td>MIPS</td>
<td>2018</td>
</tr>
<tr>
<td>Ubiquiti</td>
<td>UAP-AC-MESH</td>
<td>MIPS</td>
<td>2020</td>
<td>Ubiquiti</td>
<td>UAP-AC-PRO</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>Ubiquiti</td>
<td>UAP-LR-v2</td>
<td>MIPS</td>
<td>2020</td>
<td>Ubiquiti</td>
<td>UAP-v2</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>Ubiquiti</td>
<td>locoM2 XW</td>
<td>MIPS</td>
<td>2020</td>
<td>Ubiquiti</td>
<td>locoM5</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>Ubiquiti</td>
<td>locoM900</td>
<td>MIPS</td>
<td>2020</td>
<td>Ubiquiti</td>
<td>locoM5 XW</td>
<td>MIPS</td>
<td>2020</td>
</tr>
<tr>
<td>Ubiquiti</td>
<td>WBD-500</td>
<td>MIPS</td>
<td>2020</td>
<td>WiliGear</td>
<td>WBD-500</td>
<td>MIPS</td>
<td>2020</td>
</tr>
</tbody>
</table>

**TABLE VIII**

Details of the real-world 180 firmware images we collected from DD-WRT.