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 train 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 exploitation [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 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) is 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].
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: \texttt{mov eax,2;lea ecx,[eax+4]} and \texttt{mov eax,2;lea ecx,[eax+eax\times2]}. An ML model focusing on syntactic features might pick common substrings (both sequences share the tokens \texttt{mov}, \texttt{eax}, \texttt{lea}, \texttt{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: \texttt{[eax+eax\times2]} computes the same exact address as \texttt{[eax+4]} when \texttt{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 T\textsc{rex}, 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, T\textsc{rex} pre-trains 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\textit{not} need to execute any functions on-the-fly to match them. Instead, our model only uses the function instructions, but with an \textit{augmented} understanding of their approximate execution semantics. Importantly, such a design also saves significant runtime overhead by...
We propose a new approach to first train the model using a masked language model to learn the instructions’ execution semantics. To facilitate 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 T\textsc{rex} 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 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 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 14 CVEs not disclosed in previous studies.

Contributions. We make the following contributions.

- 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 T\textsc{rex} is faster and more accurate than the state-of-the-art tools in cross-architecture/optimization/obfuscation function matching, while running up to 8x faster. Moreover, T\textsc{rex} uncovers new vulnerabilities in real-world firmware images not disclosed by previous studies. We open-source the code, the trained model, and the dataset of T\textsc{rex} 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 semantically similar requires understanding the execution semantics of and in x86 and lsl/lsr in ARM. It also requires understanding how the values (i.e., $0xff$ 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 Fig. 2b, where two functions are compiled to ARM. The upper function is compiled with GCC-7.5 with -O0 and -O3, respectively. 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.

Cross-Obfuscation. Consider Fig. 2c, where two functions are obfuscated using clang with default options. 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.
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 [ebp-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 fine-tuning) 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 reverse its effect and predict the trace value. 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 esp 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.

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 fine-tuning, 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, e_a) (i.e., store the value e 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.

1. \( \text{t} \leftarrow \text{get_instructions}(f) \) \( \triangleright \) put all instructions in f into a queue
2. \( t \leftarrow \text{empty_vector} \)
3. \( \text{sp} \leftarrow \text{init_stack_pointer_addr()} \) \( \triangleright \) stack pointer address
4. \( \text{pc} \leftarrow \text{init_program_counter_addr()} \) \( \triangleright \) first instruction’s address
5. \( \text{sm} \leftarrow \text{mem_map}(\text{sp}, \text{STACK_SIZE}) \) \( \triangleright \) initialize stack memory
6. \( \text{cm} \leftarrow \text{mem_map}([\text{pc}]) \) \( \triangleright \) initialize memory for code
7. for each register \( r_i \) in \( \text{sp, pc} \) do
   8. \( r_i \leftarrow \text{random_init()} \) \( \triangleright \) initialize register values
9. while \( l \neq 0 \) do
10. \( i \leftarrow \text{dequeue()} \)
11. if \( i.ty \text{pe}=\text{load or } i.ty \text{pe}=\text{store} \) then \( \triangleright \) memory access
12. \( \text{mem_map}(i.\text{access_addr}, i.\text{access_size}) \)
13. if \( i.ty \text{pe}=\text{load} \) then
14. \( \text{write_random}(i.\text{access_addr}) \)
15. \( t \leftarrow t \cup \text{execute}(i) \)
16. else if \( i.ty \text{pe}=\text{jmp or } i.ty \text{pe}=\text{call or } i.ty \text{pe}=\text{nop} \) then
17. \( \text{continue} \) \( \triangleright \) skip control transfer
18. else
19. \( t \leftarrow t \cup \text{execute}(i) \)

**3.2 Input Representation**

Given a function \( f \) and its trace \( t \), we prepare the model input \( x_t \), consisting of 5 aligned sequences with the same size \( n \). Fig. 4 shows the example of Trace’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: \( x_f = \{\text{mov eax, }+...\}^g \), 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 \( x_v \) is the trace value sequence. As discussed in Section 2, we keep explicit numerical values in \( x_v \), which denote the value for each token (e.g., register) in an instruction before it is executed. For example, in mov eax, 0xb; 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 \( x_o \), opcode/operand positions \( x_n \), and the architecture sequence \( x_a \). \( x_a \) 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: \( x_a = \{x86, x64, ARM, MIPS\}^a \).

**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 \( x_t \) denote the \( i \)th value in \( x_t \), we represent \( x_t \) as an (padded) 8-byte fixed-length byte sequence \( x_t = [0x00...0x0f]^8 \) 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 \( x_t \) as input and use its last hidden cell’s output as the value representation \( t_i = \text{bi-LSTM}(x_t) \). 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 \( E_i(x_t) \), \( E_i(x_{n_i}) \), \( E_i(x_{o_i}) \), \( E_a(x_a) \) denote applying the embedding to the tokens in each sequence, respectively. We have the embedding of \( x_t \): \( E_i = E_i(x_t) + E_i(x_{n_i}) + E_i(x_{o_i}) + E_a(x_a) \). Here \( x_j \) denotes the \( i \)th token in \( x_j \), where other sequences follow the similar notation. Note that \( E_i(x_{n_i}) \) 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.
To pretrain the model with the masked LM and $W$ where $W \triangleright$ correspond to the first and $n$ dimensions into the function is a hyperparameter that weighs the cross-entropy $-\log \frac{1}{n}$.

Log transforms the with $E$ that minimizes the $g(E) = \argmin E \in \mathbb{R}^d \exists \max_{i,j} d_{i,j}$ and $p_i$ 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).

$$\text{Contextualized Embeddings.}$$ We employ the self-attention layers [84] to endow contextual information to each embedding $E_{k,i}$ which encodes the context-sensitive meaning of each token (e.g., eax in mov eax, ebx has different meaning 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) $g$, taking as input the mean pooling of all embeddings within each function, and producing a function embedding

$$g(E_{k}) = tanh\left(\frac{1}{n} \sum_{i=1}^{n} E_{k,i}\right) \cdot W_{1} \cdot W_{2}.$$}

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 $g_{f}$ be parameterized by $\theta^f$, the finetuning objective minimizes the cosine embedding loss $l_{ce}$ between the ground-truth and the cosine distance between two function embeddings (Fig. 4b):

$$l_{ce}(g_{f}(E_{k}^{(1)}), g_{f}(E_{k}^{(2)}), y),$$

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 chosen between 0 and 0.5 [70]. As both $g_{p}$ and $g_{f}$ are differentiable, optimizing Equations (1) and (2) can be guided by gradient descent via backpropagation.

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

Datasets. To train and evaluate T\textsc{rex}, 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., \textit{O0} - \textit{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 (\textit{bcf}), control flow flattening (\textit{cff}), register-based indirect branching (\textit{ibr}), basic block splitting (\textit{spl}), and instruction substitution (\textit{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., \textsc{Asm2Vec}) 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 \textsc{SAFE} [57], as \textsc{SAFE}'s model is publicly available, we run their trained models on our collected binaries. We also compare \textsc{SAFE}'s reported results on their dataset, i.e., OpenSSL-1.0.1f and OpenSSL-1.0.1 u. The second baseline is \textsc{Gemini} [93]. As \textsc{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 \textsc{Asm2Vec} [23] and \textsc{Blex} [25] as the baselines. \textsc{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 \textsc{Asm2Vec} that achieves extremely low Precision@1 (the metric used in \textsc{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 \textsc{T\textsc{rex}} 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, \textsc{Asm2Vec} uses Precision at Position 1 (Precision@1), and \textsc{Blex} uses the number of matched functions as the metric. Therefore, we also include these metrics to evaluate \textsc{T\textsc{rex}} when needed.

Training Setup. To separate the functions in pretraining, finetuning, and testing, we pretrain \textsc{T\textsc{rex}} 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

<table>
<thead>
<tr>
<th>ARCH</th>
<th>OPT</th>
<th>OBFP</th>
<th>Binutils</th>
<th>Coreutils</th>
<th>Curl</th>
<th>Diffutils</th>
<th>Findutils</th>
<th>GMP</th>
<th>ImageMagick</th>
<th>microtikp</th>
<th>TomCrypt</th>
<th>OpenSSL</th>
<th>PaTty</th>
<th>SQLite</th>
<th>Zlib</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>O0</td>
<td>25,492</td>
<td>19,992</td>
<td>1,067</td>
<td>944</td>
<td>1,529</td>
<td>766</td>
<td>2,938</td>
<td>200</td>
<td>779</td>
<td>11,918</td>
<td>7,087</td>
<td>2,283</td>
<td>151</td>
<td>75,152</td>
</tr>
<tr>
<td></td>
<td>O1</td>
<td>20,043</td>
<td>14,918</td>
<td>771</td>
<td>694</td>
<td>1,128</td>
<td>704</td>
<td>2,341</td>
<td>176</td>
<td>745</td>
<td>10,991</td>
<td>5,765</td>
<td>1,614</td>
<td>143</td>
<td>60,033</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>19,493</td>
<td>14,778</td>
<td>765</td>
<td>693</td>
<td>1,108</td>
<td>701</td>
<td>2,358</td>
<td>171</td>
<td>745</td>
<td>11,001</td>
<td>5,756</td>
<td>1,473</td>
<td>138</td>
<td>59,180</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>17,814</td>
<td>13,931</td>
<td>697</td>
<td>627</td>
<td>983</td>
<td>680</td>
<td>2,294</td>
<td>160</td>
<td>726</td>
<td>10,633</td>
<td>5,458</td>
<td>1,278</td>
<td>125</td>
<td>55,406</td>
</tr>
<tr>
<td>MIPS</td>
<td>O0</td>
<td>24,640</td>
<td>18,843</td>
<td>1,042</td>
<td>906</td>
<td>1,463</td>
<td>734</td>
<td>2,840</td>
<td>200</td>
<td>779</td>
<td>11,866</td>
<td>7,003</td>
<td>2,199</td>
<td>153</td>
<td>75,488</td>
</tr>
<tr>
<td></td>
<td>O1</td>
<td>22,530</td>
<td>17,746</td>
<td>653</td>
<td>1,099</td>
<td>670</td>
<td>2,243</td>
<td>176</td>
<td>745</td>
<td>10,940</td>
<td>5,685</td>
<td>1,330</td>
<td>139</td>
<td>60,887</td>
<td></td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>22,004</td>
<td>13,647</td>
<td>741</td>
<td>653</td>
<td>1,039</td>
<td>667</td>
<td>2,266</td>
<td>171</td>
<td>743</td>
<td>10,952</td>
<td>5,677</td>
<td>1,395</td>
<td>135</td>
<td>60,081</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>20,289</td>
<td>12,720</td>
<td>673</td>
<td>584</td>
<td>917</td>
<td>646</td>
<td>2,198</td>
<td>161</td>
<td>724</td>
<td>10,581</td>
<td>5,376</td>
<td>1,197</td>
<td>121</td>
<td>56,187</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Function Pairs Across Architectures, Optimizations, and Obfuscations</th>
<th>Cross-Opt+OBF</th>
<th>ARCH+OPT+OBF</th>
<th>ARCH+OPT</th>
<th>OPT</th>
<th>ARCH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binutils</td>
<td>.991</td>
<td>.999</td>
<td>.988</td>
<td>.959</td>
<td>.947</td>
</tr>
<tr>
<td>Coreutils</td>
<td>.988</td>
<td>.99</td>
<td>.989</td>
<td>.955</td>
<td>.945</td>
</tr>
<tr>
<td>Curl</td>
<td>.991</td>
<td>.993</td>
<td>.99</td>
<td>.967</td>
<td>.956</td>
</tr>
<tr>
<td>Diffutils</td>
<td>.989</td>
<td>.992</td>
<td>.99</td>
<td>.973</td>
<td>.961</td>
</tr>
<tr>
<td>Findutils</td>
<td>.99</td>
<td>.991</td>
<td>.99</td>
<td>.966</td>
<td>.962</td>
</tr>
<tr>
<td>GMP</td>
<td>.99</td>
<td>.989</td>
<td>.989</td>
<td>.967</td>
<td>.964</td>
</tr>
<tr>
<td>ImageMagick</td>
<td>.992</td>
<td>.994</td>
<td>.987</td>
<td>.957</td>
<td>.955</td>
</tr>
<tr>
<td>microftpd</td>
<td>.991</td>
<td>.994</td>
<td>.99</td>
<td>.97</td>
<td>.965</td>
</tr>
<tr>
<td>TomCrypt</td>
<td>.989</td>
<td>.991</td>
<td>.99</td>
<td>.971</td>
<td>.957</td>
</tr>
<tr>
<td>OpenSSL</td>
<td>.991</td>
<td>.99</td>
<td>.989</td>
<td>.967</td>
<td>.957</td>
</tr>
<tr>
<td>PuTTY</td>
<td>.989</td>
<td>.992</td>
<td>.99</td>
<td>.965</td>
<td>.95</td>
</tr>
<tr>
<td>SQLite</td>
<td>.99</td>
<td>.992</td>
<td>.99</td>
<td>.967</td>
<td>.959</td>
</tr>
<tr>
<td>Zlib</td>
<td>.989</td>
<td>.992</td>
<td>.987</td>
<td>.968</td>
<td>.961</td>
</tr>
<tr>
<td>Average</td>
<td>.99</td>
<td>.992</td>
<td>.989</td>
<td>.966</td>
<td>.957</td>
</tr>
</tbody>
</table>
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 T.REX and 14% for Asm2Vec), but T.REX’s AUC score drops much less than that of Asm2Vec.

To compare to Blex, we evaluate T.REX 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 T.REX 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 T.REX is more effective than the hand-coded features in Blex for matching similar binaries.

**Cross-Obfuscation.** We compare T.REX to Asm2Vec on matching obfuscated function binaries. Asm2Vec is evaluated on obfuscations including bcf, cff, and sub – as 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 T.REX’s results on other projects.

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

### 5.3 RQ3: Execution Time

We evaluate the speed of generating function embeddings for computing similarity. We compare T.REX 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 T.REX scales to different number of functions. Since the offline training (i.e., pretraining T.REX) 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 T.REX 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.
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 RO4: 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
We quantify the accuracy of REACTS TO and SAFE in searching functions in the firmware. The queries and the firmware are from (a) both ARM, (b) x64 and ARM, respectively.

Fig. 12. Top-1/3/5/10 error of T\textsubscript{REX} 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

<table>
<thead>
<tr>
<th>CVE</th>
<th>Ubiquiti sunMax</th>
<th>TP-Link Deco-M4</th>
<th>NETGEAR R7000</th>
<th>Linksys RE7000</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2016-6303</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2016-6302</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2016-2842</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2016-2182</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2016-2180</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2016-2178</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2016-2176</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2016-2109</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2016-2106</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2016-2105</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2016-0799</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2016-0798</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2016-0797</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2016-0705</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

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 T\textsubscript{REX}. 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 T\textsubscript{REX}’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 T\textsubscript{REX}, which helps discover real-world vulnerabilities in large-scale firmware databases.

Vulnerability Search Performance. We quantify the accuracy of T\textsubscript{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 5. Specifically, we compile OpenSSL to ARM and x64 with O3, and feed both our compiled and firmware’s binaries to T\textsubscript{REX} 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 T\textsubscript{REX} 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. – T\textsubscript{REX} 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 T\textsc{rex} 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 obfuscations 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, T\textsc{rex} 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 T\textsc{rex} 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 T\textsc{rex}. However, the concrete value space can be extremely large. This leads to the question how T\textsc{rex} 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 T\textsc{rex} 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 Similarity.** 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, T\textsc{rex} 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. T\textsc{rex} 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]. T\textsc{rex} only uses the traces to learn approximate execution semantics of instructions and transfers 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 reasonabled-sized programs, e.g., httpd and sthttpd. As a reference comparison, we run T\textsc{rex} on all the function pairs between httpd (102 functions) and sthttpd (103 functions). T\textsc{rex} 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, T\textsc{rex} 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 T\textsc{rex} 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 pre-training 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, Capital One, Google, JP Morgan, DiDi, or Amazon.

References

A. Maier, H. Gascon, C. Wressnegger, and K. Rieck, "TypeMiner: Detecting

L. Luo, J. Ming, D. Xu, and D. Wu, "BinSim: Trace-based

D. Xu, D. Wu, and Y. Xiang, "Memoized semantics-based binary

D. Mu et al., "RENN: Efficient reverse execution with neural-net-


J. Jiang, M. Woo, and D. Brumley, "Towards automatic software

D. Kim, E. Kim, S. K. Cha, S. Son, and Y. Kim, "Revisiting binary

H. Koo, S. Park, D. Choi, and T. Kim, "Semantic-aware binary


X. Li, Q. Yu, and H. Yin, "PalmTree: Learning an assembly

Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli, "Graph matching

F. Liu, Y. Wen, D. Zhang, X. Jiang, X. Xing, and D. Meng, "Log2vec: A heterogeneous graph embedding based approach


