Detecting Obfuscated Function Clones in Binaries using Machine Learning

Michael Pucher
University of Vienna, Austria
m.pucher@univie.ac.at

Christian Kudera
SBA Research
ckudera@sba-research.org

Georg Merzdovnik
SBA Research
gmerzdovnik@sba-research.org

Abstract—**The complexity and functionality of malware is ever-increasing. Obfuscation is used to hide the malicious intent from virus scanners and increase the time it takes to reverse engineer the binary. One way to minimize this effort is function clone detection. Detecting whether a function is already known, or similar to an existing function, can reduce analysis effort. Outside of malware, the same function clone detection mechanism can be used to find vulnerable versions of functions in binaries, making it a powerful technique.**

This work introduces a slim approach for the identification of obfuscated function clones, called OFCI, building on recent advances in machine learning based function clone detection. To tackle the issue of obfuscation, OFCI analyzes the effect of known function calls on function similarity. Furthermore, we investigate function similarity classification on code obfuscated through virtualization by applying function clone detection on execution traces. While not working adequately, it nevertheless provides insight into potential future directions.

Using the ALBERT transformer OFCI can achieve an 83% model size reduction in comparison to state-of-the-art approaches, while only causing an average 7% decrease in the ROC-AUC scores of function pair similarity classification. However, the reduction in model size comes at the cost of precision for function clone search. We discuss the reasons for this as well as other pitfalls of building function similarity detection tooling.

I. Introduction

Binary similarity analysis is a valuable tool for security analysis. It can assist in finding vulnerable code in software corpora, as well as reduce the complexity for reversing software and malware. One reason for this is the practice of code reuse for similar tasks, also prevalent among malware [42], [59]. The problem intensifies with the use of statically linked binaries, which will not only include the *interesting* parts of the code, but bundle library functions as well. A notable example is Go [56], since it defaults to static linking. The number of Go related malware has increased by a staggering amount [6], potentially sharing a large amount of code with other applications or contain parts of standard library or utility functions. While binaries can include debug information like function names, those are usually stripped before distribution or might even be harmful when analyzing malicious binaries [45].

Workshop on Binary Analysis Research (BAR) 2022
24 April 2022, San Diego, CA, USA
ISBN 1-891562-76-2
https://dx.doi.org/10.14722/bar.2022.23005
www.ndss-symposium.org

Function clone detection is one approach to identify code reuse across binaries. The idea is to identify re-use of known functions (e.g. from libraries) in binaries without symbol/debug information, to re-assign names or identify known vulnerabilities in unknown binaries. The focus of function clone detection lies in first generating a database of function signatures and then comparing unidentified functions with the database to identify code re-use. Numerous approaches for this exist, with machine learning techniques gaining popularity [13], [39], [20]. Machine learning based on binary features is well suited for this task, as it is inherently *fuzzy* in its nature. As function clone detection is intended to find functions that might not be exact matches, learning similarity of functions allows matching functions that share certain features. This is backed by a recent survey on binary similarity [22], showing that approaches based on machine learning performed well in comparison to others. Additionally to searching for similar functions in a dataset, such tools can be used to perform function similarity classification, i.e. determining if two functions are equal. However, such approaches also have their drawbacks. Current function clone detection approaches have shortcomings when dealing with obfuscation. Additionally, training the models takes a significant amount of time and resources. This might not be a problem for e.g. language processing, where a model can be trained once and then be reused. But when we are dealing with function clone detection, specifically for the case of obfuscation, we are lacking the corpus to train one model that fits all applications. Therefore models might need re-training more often to be adapted to specific use-cases.

The aim of this work is to improve upon existing capabilities of identifying function clones in obfuscated binaries, while simultaneously decreasing the model size to allow training and execution with lower requirements for computing resources. The main contributions of this paper are:

- **Development of an end-to-end function clone identification framework.** Existing approaches have not published their whole data processing pipeline, or rely on commercial tools, like IDA Pro [24], for feature extraction. This paper presents OFCI, an open source[1] end-to-end framework, built from publicly accessible and open source tools.
- **A reduced and improved machine learning model for function clone identification.** OFCI aims to reproduce similar performance of related models while

[1] https://github.com/sbaresearch/ofci

reducing their high computational complexity. To improve identification of function clones, OFCI uses API calls, like system or library calls, as an additional iterative feature vector, i.e. detection of a function clone can improve the identification of other functions calling this clone.

- **Matching state-of-the-art performance of function similarity classification in the presence of basic obfuscation.** OFCI can match state-of-the-art solutions for function similarity classification when obfuscations like bogus control-flow, control-flow flattening, instruction substitution etc. are present.
- **Analysis of identifying functions in the presence of virtual machine obfuscation.** OFCI presents the *first* approach to function clone detection, when functions are obfuscated using virtualized code based on recorded execution traces. While the results themselves are not adequate, we identify possible pitfalls and research directions are given to assist future research.
- **Analysis of reduced function clone search performance.** While the ROC-AUC scores reported by OFCI are comparable to the state of the art, the function clone search shows reduced performance. Our work gives an analysis into possible issues and the consequences for other function clone search implementations.

II. BACKGROUND

To identify similar functions in optimized and obfuscated binaries, first a definition of what constitutes *similarity* between two segments of binary code is required. For the purpose of this work, as is also the case in related works [13], [39], [47], [63], we define executable binary code to be similar if it is semantically similar.

A. Function Clone Detection

The binary similarity problem can be constrained to *function clone detection*. By restricting binary similarity to function boundaries, function clone detection can be defined as an information retrieval (IR) problem. Treating binary code as a language then allows application of modern information retrieval algorithms, designed for, e.g., full text search.

B. Obfuscation

Detection of function clones might be hindered by obfuscation, especially when dealing with malware. Obfuscation means transforming code in such a manner that it becomes harder to analyze, while still preserving the execution semantics. Hikari [64] and Tigress [7] are two freely available obfuscation tools, with selected passes used in related work [13], [47], [3], [9], [41] as well.

1) Hikari: is based on LLVM and can be used to obfuscate complete projects. It is a fork of Obfuscator-LLVM [33] including additional obfuscation passes. **Bogus Control Flow (BCF)** changes the control-flow graph (CFG) by inserting new basic blocks to make it harder to find the relevant basic blocks of the original function. **Basic Block Splitting (SPL)** breaks larger basic blocks apart, thereby creating a large number of new basic blocks. **Control Flow Flattening (CFF)** changes the control flow into a flat structure, where basic blocks are called from a central dispatcher. At the end of each block control returns to the dispatcher, which decides which block to execute next. The CFG of every function will then have the same basic structure. In comparison, Hikari's **Register-Based Indirect Branching (IBR)** does not flatten the control-flow, but replaces every existing branch with an indirect one. The **Instruction Substitution (SUB)** does not change the CFG, instead replacing instructions with a chain of more complex ones that produce the same output for all expected inputs.

2) Tigress: is a source-to-source obfuscator that supports more types of obfuscation, but can not as easily be applied to full projects. **Virtualized Code (VIRT)**, also used in commercially available obfuscators [57], [61], transforms the code into bytecode that is run in a virtual machine (VM). While there exist different approaches to VM deobfuscation, there appears to be no prior work regarding binary code similarity of code executed in the VM. **Encode Arithmetic (EA)** strengthens arithmetic operations by generating mixed boolean-arithmetic expressions.

C. Machine Learning

Modern IR relies on complex machine learning architectures. In the context of language processing, words are distributed across a text in a certain manner, with some words more likely to appear in specific parts of a sentence or surrounded by specific words. Embedding vectors, popularized by WORD2VEC [43], are based on this concept and provide a vector representation of words *embedded* in their contexts, thus making it possible to represent simple semantic concepts through operations on vectors. This can be applied on the property of cosine similarity, allowing similar words to be represented by vectors having a high cosine similarity. Word embeddings map simple semantics concerning single words and their contexts, but not sequences. To process sequences, a neural network needs to incorporate memory to keep state from previous and successive words in a sentence to form the embedding for a whole sequence. Recurrent neural networks (RNN) were invented to model such processes, later followed by the Long Short-Term Memory (LSTM) [25] architecture. The ability to process sequences allows a sequence-to-sequence model, or seq2seq model, which transforms an input to an output sequence. A seq2seq model consists of an encoder to convert an input sequence to an internal representation, and a decoder, transforming the internal representation into a sequence in the domain of the output vocabulary. Due to the sequential nature of RNN/LSTM models, training cannot effectively use modern architectures that heavily rely on GPU processing, leading to the introduction of the *Transformer* architecture [60]. A transformer is a seq2seq model designed to work with a fixed maximum input length to optimize the training and inference on GPUs. It consists of a stack of encoders and decoders, which contain self-attention layers and feed-forward networks. BERT [12] was introduced, removing the decoder half of a transformer and stacking several encoder layers. These layers can be pre-trained with a general task and, instead of adding a decoder, a task-specific neural network can be added on top as *head* for fine-tuning. One such general pre-training task is masked language modeling (LM), which is semi-supervised: A sequence is fed into the model and before processing BERT masks out random tokens in the input sequence, training the model to predict the original token at the masked location. ALBERT [35] improves on BERT by reducing the number of parameters, without sacrificing benchmark performance. This is achieved by factorizing the embedding matrix, allowing to keep hidden layer sizes small with a growing vocabulary, and the sharing of parameters across hidden layers.

III. RELATED WORK

As function clone identification is not always wanted or possible, most research is done in the broader field of binary similarity, which is not necessarily limited to the scope of functions.

A. Classical Approaches to Binary Similarity

In the context of this paper, all approaches that do not rely on machine learning techniques are referred to as classical approaches. One category is similarity based on extracted features such as string references, call sequences or CFG matching heuristics, with prime examples for this being BINDIFF [4] or DISCOVRE [16]. Fuzzy hashing can be used to compare these features or byte sequences directly, with different forms being used. The most common form is locality-sensitive hashing (LSH) [28] as part of the pipeline in a large number of approaches such as CACOMPARE [26], BINHASH [31], MULTI-MH [49] and others ([14], [27], [8], [53], [54], [65], [22]). Static analysis approaches, which are prone to simple obfuscation attacks, are complemented by dynamic techniques that make use of traces ([11], [10]) paired with sampling of I/O pairs [5], symbolic execution [49], or monitoring of the environmental changes (e.g. memory of pointer arguments) caused by functions ([15], [40]). Dynamic analysis offers a trade-off between more accurate results and longer analysis runtimes, leading to approaches like EXPOSE [44] pre-filtering the analysis set statically before performing dynamic analysis.

B. Approaches Based on Machine Learning

The rise of computation power and development of modern machine learning techniques opened up new possibilities for information retrieval and natural language processing. While some approaches use machine learning to learn embeddings based on manually selected structural features ([18], [63], [20], [19]), related work has in general shifted away from relying on these features, rather making use of WORD2VEC-like techniques ([13], [39], [37], [65], [51]). One of these approaches is ASM2VEC [13], training embeddings directly on the instruction text. Similar to WORD2VEC, these embeddings would only represent single instructions and not functions, which is solved by adopting a customized PV-DM [36] approach. To capture the control-flow of a function, ASM2VEC is trained on random walks through CFGs of functions. In comparison, INNEREYE [65] and SAFE [39] take a two-step approach and first train a WORD2VEC model on instructions and later process a sequence of instruction embeddings with an LSTM and an RNN respectively. Additionally, SAFE makes use of self-attention features, leading to recent approaches ([47], [48]) incorporating transformers through the BERT architecture. TREX [47] introduces additional data for pre-training the model by recording microtraces to capture program semantics, before fine-tuning the model on the function similarity task. This pre-training can be done on a different dataset than the fine-tuning, which ideally contains a wide range of different instructions on different architectures. In comparison to TREX, OSCAR [48] does not require additional data, but instead achieves good results by customizing BERT for code similarity and lifting binary code to an intermediate representation.

C. Deobfuscation

As machine learning does not provide exact results, it cannot be used directly to recover exact unobfuscated code, but can still be used as part of a general solution. One use case is the restoration of code metadata for use with a human analyst, similar to assigning function names through function clone detection. Examples for this include DEBIN [23] trying to recreate debug information or DEGUARD [2] and MACNETO [55] trying to restore variable names and type information in obfuscated Android applications. Other use cases for machine learning in deobfuscation have appeared in the context of opaque predicates and VM deobfuscation. Tofighi-Shirazi et al. [58] make use of supervised learning to label whether a predicate is opaque or not and SYNTIA [3] and XYNTIA [41] have used machine learning for deobfuscating VM handlers. As the latter approaches make use of program synthesis, machine learning and probabilistic algorithms are used to guide the search space exploration.

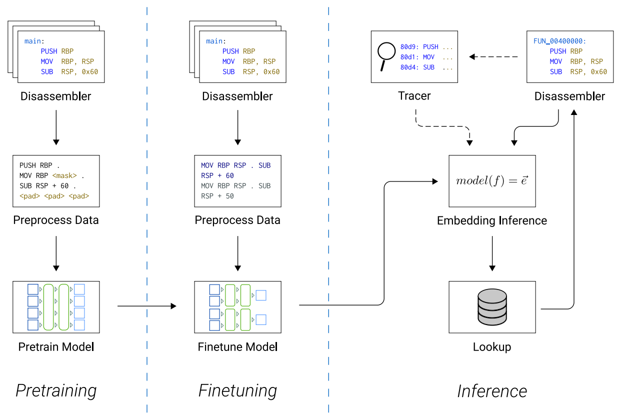
IV. OBFUSCATED FUNCTION CLONE IDENTIFICATION

A review of related approaches has shown that binary similarity has profited from machine learning and that usage of machine learning is viable for deobfuscation tasks. Out of recent state-of-the-art function clone detection approaches, only some approaches deal with obfuscation [47], [13], [30], [34]. None of these approaches have tried to apply binary code similarity algorithms to counter obfuscation by virtualization. In order to improve the state of function clone detection when faced with heavily obfuscated code, this work introduces Obfuscated Function Clone Identification, or OFCI for short.

A. Assumptions and Threat Model

Before discussing the general architecture, it is necessary to highlight the assumptions and constraints under which OFCI was designed to operate. Firstly, we assume that detection of function entry points and boundaries is provided correctly by existing tools/libraries, in our case Ghidra [1]. Since OFCI uses textual disassembly to compare functions, a working disassembler is required; however, compared to other solutions (e.g. [13], [63]), OFCI does not need to reconstruct the CFG. OFCI's model can be trained using a consumer grade GPU, however using the model does not require a GPU.

The threat model of OFCI assumes the author of the binaries under analysis to be an attacker, who makes use of obfuscation techniques to hide information contained in the binary. The attacker is assumed to only apply the obfuscations discussed in subsection II-B, i.e. only obfuscations that do not split or merge functions. However, standard compiler optimizations, like function inlining, can be applied as usual. The attacker may make use of code virtualization as implemented by TIGRESS [7].



```
main:
    PUSH RBP
    MOV  RBP, RSP
    SUB  RSP, 0x60
```

```
PUSH RBP .
MOV RBP <mask> .
SUB RSP + 60 .
<pad> <pad> <pad>
```

```
MOV RBP RSP . SUB
RSP + 60
MOV RBP RSP . SUB
RSP + 50
```

```
80d9: PUSH ...
80d1: MOV  ...
80d4: SUB  ...
```

```
FUN_00400000:
    PUSH RBP
    MOV  RBP, RSP
    SUB  RSP, 0x60
```

Fig. 1: Overview of the OFCI architecture and pipeline.

B. Architecture Overview

The architecture of OFCI and its three basic pipelines are shown in Figure 1: Pretraining, fine-tuning and inference within the reverse engineering environment. It adopts recent trends in natural language processing that have been shown to work well in related work ([47], [48]), causing the model training process to be split into pre-training and fine-tuning. OFCI makes use of this insight to build a simpler framework that does not require the additional model parameters required to accommodate dynamic value information [47] or lifting of the disassembly to an IR [48]. A disassembler is required to extract the textual instructions from the base dataset or a target program under analysis. OFCI then processes these instructions together with their function metadata and stores them for model training or directly uses them for function embedding inference and lookup. Before embeddings can be generated, the model has to undergo a pre-training process where it tries to learn general concepts about the extracted disassembly through masked LM. This is a one-time effort and is run on the complete corpus of binaries. While the pre-training doesn't require any metadata and can operate solely on the extracted instructions, fine-tuning makes use of contrastive learning, where two functions are compared based on the cosine similarity of their embeddings. This requires the selection of function pairs after the function instructions have been extracted and additionally adds a label, marking whether the paired functions are similar or dissimilar. After the training process, the model can be used for detecting function clones on new binaries, by filling a database with embeddings of known functions and detecting the clones in the new binary. In the case of code obfuscated through virtualization, we make use of instruction tracing and use the resulting trace to infer the embedding for lookup.

C. Feature Modeling

Feature extraction happens at every part of the pipeline, providing the necessary input data for the embedding model. When comparing the task of function similarity with sentence similarity in natural language processing, the idea of treating disassembly as text and using it directly seems to appear straightforward. However, further processing is required, largely due to the appearance of numeric constants in the textual instructions that present a challenge in a typical word processing setting. These constants can carry specific meaning, e.g. relative address offsets or special constants used for bitwise operations, which should be preserved to aid identification of a function. Issues arise when trying to tokenize these numbers, as in theory all 64-bit numbers could be represented in the disassembly, resulting in a prohibitively large vocabulary. Therefore those constants need to be handled specifically. Instead of replacing all numeric constants with a single token ([39], [13]) or encoding the constants through additional model support [47], OFCI splits the constants into bytes. Every time it encounters a scalar value or address, it first treats the value as an 8-byte integer in little endian byte order. Then, every byte of the integer is added to the stream of normalized disassembly words as hexadecimal number, increasing the vocabulary by a fixed size of 256 entries. Trailing zeros are omitted in order to reduce the amount of needed tokens and with similar reasoning, negative constants are interpreted with their absolute value and a preceding negation sign, instead of using their actual byte representation. In general, most other information present in the disassembly instructions is left unchanged: All special characters, e.g. brackets or arithmetic operators, are treated as separate tokens, instruction operands are separated by whitespace, and the instructions itself are separated with dots. Lastly, to turn the words of the normalized disassembly into numeric tokens, we make use of a Byte Pair Encoding (BPE) tokenizer.

Another issue is the fixed input size of transformer models, in our case 512 tokens. Functions can be much longer than 512 tokens, in which case we split them into *fragments*: All functions are split into fragments with a maximum size of 512 tokens each. When matching two functions that contain more than one fragment, the surplus fragments of the longer function are ignored and the function fragments matched side-by-side during training. During inference, the embeddings for all fragments of a function are calculated and averaged to produce one single embedding for the whole function.

OFCI tries to incorporate the basic idea of function inlining, without actually inlining the function into the caller. Instead, address references to the entrypoint of a function are recorded and replaced in the normalization phase, a feature we named *Call-ID*. Whenever a binary is passed through the data processing phase of OFCI, the names of all functions are compared against a database of function names, or added to the database, generating a new unique ID. The preprocessor remembers the IDs of these functions for the disassembly normalization phase. During the normalization phase, whenever an address is encountered as numerical constant, the preprocessor checks whether the address points to the start of a function. If not, it is directly normalized through the previously described scalar splitting, but if it turns out to be a function address, the generated ID for the function name is normalized instead. This encodes references to other functions in and outside of the binary, which is especially interesting in the case of API functions or syscalls, as these are usually known from the start. Call-ID therefore allows for iterative embedding generation as more functions used in the binary are identified at each step.

D. Neural Network Architecture

OFCI is mainly inspired by related work ([47], [48]) making use of BERT-like architectures. While these approaches modify the base architecture to some extent, the goal of OFCI is to make use of stock implementations, while also reducing the overall model size and still achieving comparable results to the state of the art. To this end, we adopt a different BERT architecture called ALBERT [35], designed specifically to reduce the model size of BERT. In order to limit the parameter count of BERT significantly, ALBERT introduces two optimization strategies: Factorization of the vocabulary embedding matrix and sharing of parameters between layers. As the vocabulary used for tokenization grows, so does the embedding matrix, which is in turn influenced by the size of the hidden layers. ALBERT factorizes the embedding matrix into smaller matrices to accommodate bigger vocabularies and hidden sizes.

In the context of OFCI, ALBERT is pre-trained using masked language modeling, with the disassembly features that have been previously extracted. After pre-training, a simple feed forward network, a *head* for the base network, has to be added on top of ALBERT. The similarity head is constructed as follows: First, mean pooling of the embeddings is calculated, then passed through a dropout layer, a single feed forward layer with the $tanh$ activation function, through another dropout layer, then through a linear layer and normalization. This follows roughly the architecture used by BERT classification tasks, or the similarity head implemented in SENTENCE-BERT [52] and TREX [47]. For training the similarity head, a contrastive learning approach is used, taking a pair of two functions and a similarity label as input. The goal of the task is to make the embeddings of similar functions have a high cosine similarity, while dissimilar functions should produce a low cosine similarity value. To achieve this, the two functions are first passed through the model with the similarity head, producing embedding variables, which are then combined through a cosine loss function calculating the distance from the label.

E. Virtual Machine Analysis

Research dealing with virtualization in the context of function clone detection is scarce. A recent survey [34] highlights the need for covering interprocedural virtualization obfuscators like Themida [57], or VMProtect [61], as the obfuscations applied by Hikari [64] do not appear to be more complex than cross-optimization binary similarity. OFCI does not solve interprocedural obfuscation either, but is intended as a stepping stone to expand research in this area. To this end, OFCI aims to perform function clone search on functions virtualized with Tigress [7], which works by virtualizing functions separately.

When virtualizing code, the original code of a function is translated into a new language, i.e. bytecode, which is intended to obfuscate the original intention of the instructions. This transformation is in stark contrast to the obfuscations of Hikari, as neither the original instructions, nor control-flow is preserved. However, since the transformation can be interpreted as translation into a new language, BERT-like architectures should still perform reasonably well. M-BERT [21] has shown this and existing work on function clone detection [47], [39] produces good results across architectures. Treating the bytecode as different architecture directly is not straightforward, because a static disassembler might not be able to identify where or how the bytecode is stored and there is no straightforward way to map bytecode operations with their arguments to text. Because our approach relies on disassembly text, we collect instruction traces: The recorded instructions implicitly encode the semantics of the executed bytecode. These traces are processed by our training/inference machinery in the same manner as static disassembly. OFCI does not currently support automatic trace creation. If a trace is required for analysis, it has to be created manually using our tracing tool, as not all inputs will produce traces covering the relevant parts of the program.

V. IMPLEMENTATION

While recent approaches for function clone detection perform well on paper, code is only published in rare cases [39], [48], with others selectively publishing code [63], [47] or relying on expensive proprietary software as part of their pipeline [13]. The main requirement for implementing OFCI is therefore to only rely on recent and freely available software and to publicly release all parts of its own pipeline, including trained models. This is achieved by implementing feature extraction and end-to-end functionality as a Ghidra plugin, training the model with the Hugging Face Transformers library [62] and PyTorch [46], collecting traces with Intel Pin [29] and performing embedding lookup with Faiss [32]. The custom code for interfacing with these tools is kept at a minimum, making it straightforward to swap to different transformer models or use different libraries altogether.

The Ghidra plugin extracts the function symbol names and normalizes opcodes and arguments in the disassembly, before exporting it into a database. The function names are extracted for labeling the fine-tuning dataset and *thunked* functions, e.g. PLT resolving code, are not disassembled. As imported functions are important for the Call-ID feature, the names of *thunked* functions are however still saved. Call-ID is implemented by assigning a unique ID for every function name across the dataset, which is later propagated into the normalized disassembly instead of the original call address. This is simplified by using Ghidra's internal structure of instructions, instead of parsing the textual disassembly: Ghidra represents instruction operands that take memory addresses as *Address* objects. OFCI then performs a lookup in the Call-ID map and, if the address points to a function, replaces it with the Call-ID in the normalized disassembly. If the binary is statically linked, syscalls are included in this process and treated as function calls by our plugin. While Ghidra is generally slower when compared to other disassemblers [50], OFCI can make use of multiple headless instances to process larger datasets in parallel.

After the normalized disassembly has been exported, we tokenize it with the help of the BPE tokenizer implementation of the Transformers library. If this tokenization is done for the first time, a BPE vocabulary is trained. The vocabulary used in our implementation amounts to roughly 800 entries. Due to the small number of vocabulary entries, token lists are stored either as JSON for size advantage, or as 16-bit integer binary dumps for efficient loading. When generating the training datasets, token lists are split and padded into chunks of size 512, the chosen input size of the used ALBERT transformer. For pre-training, all chunks of the full dataset are written into a binary dump file. The fine-tuning dataset generation is more involved: Due to the contrastive learning approach, function pairs need to be selected from the dataset. We define functions as similar when they have the same name and therefore select similar pairs of functions with the same name, but different optimization/obfuscation levels. TREX generates 5 and SAFE generates 1 dissimilar function pair on 1 similar function pair; for our approach, we chose the middle ground and used a 1:2 ratio for similar and dissimilar function pairs. When matching function pairs, the differing number of chunks for each function has to be taken into account and we match the chunks by *zipping* the two functions, i.e. we use the number of chunks of the shorter function and then match one chunk of the first function with the opposite chunk in the second function. The goal of the fine-tuning dataset generation is to generate roughly 50k chunks for each obfuscation category.

Pre-training can make use of the unmodified ALBERT implementation of the Transformers library, together with masked LM as training objective. As we already performed tokenization beforehand, the only relevant missing field is the attention mask, where tokens that should not be considered for attention calculation are masked, which is the padding token in our case. For fine-tuning, a similarity head is needed for the transformer, which is not directly provided by Transformers and implemented by us as described in our description of OFCI, based on the other task-specific heads already implemented in the Transformers library for ALBERT. For the similarity task, each chunk of a function chunk pair is passed through the model and then combined with the label through the cosine embedding loss function provided by PyTorch.

While Ghidra itself provides simple tracing capabilities we utilize Intel Pin, which is a dedicated tool for dynamic analysis, to create instruction trace files. We implemented a pintool with the intention of recording the address of every executed instruction, importing the recording into Ghidra and then iterating over each address to disassemble and normalize the instruction at this address in the same manner as the static disassembly. This allows reusing the normalization procedures we implemented in the Ghidra plugin, avoiding having to duplicate this code into the pintool. As an optimization, not all instruction addresses are recorded, but only start and end addresses of a pin basic block, which can be passed as an address range to Ghidra, allowing the disassembly in bulk. Only the pin basic blocks located in the main binary image are logged; while traces for external libraries could make use of Call-ID, the tracing is only performed on intraprocedural virtualized code The trace file addresses also subtract the main binary image base address to keep addresses position-independent. The pintool does not address function boundaries in a special way and solely logs addresses. The Ghidra plugin will then decide during analysis which portions of the log are assigned to which function, assuming that there is only at most one trace of each function present in the trace file. Therefore, repeated traces have to be split into different trace files.

Function clone search is performed through Faiss [32], which provides several efficient embedding index lookup techniques. In general, after fine-tuning OFCI, the embeddings of all functions are stored in an index. For evaluating OFCI, we use the flat index making use of the L2 distance between vectors, which is equivalent to cosine similarity search in our case, as we normalize the embedding vectors. While the L2 index provides exact results, Faiss also provides more efficient indexing techniques, which come with the cost of some precision loss. For performing clone search, the embeddings generated from exported Ghidra functions are used as query tensor for the index. Faiss returns the first k ranked results, where we defined $k = 1$ for our purposes, retrieving only the most similar function label.

VI. EVALUATION

To evaluate OFCI's performance, we compare it to the two underlying approaches TREX [47] and SAFE [39], as well as recent approaches with the same goal, such as ASM2VEC [13]. We furthermore discuss other implications rarely covered in existing work, e.g. practical applicability and training time.

A. Experiment Setup

Pre-training of the model is performed on a *server* running Ubuntu 18.04.5 LTS with an NVIDIA GTX Titan X, 94 GB of RAM and an Intel Xeon X6580 CPU. All other tasks use a mid-range *developer workstation* based on Arch Linux with a 5.14 kernel, an AMD Ryzen 1800x, 32 GB of RAM, and an NVIDIA RTX 2070 Super. The server does not offer better training performance, but was used to avoid potential stability problems as pre-training requires several days of runtime. Composition of the dataset is described in detail in section A.

Function similarity is a classification problem and therefore the performance across different thresholds is measured using the *Receiver Operating Characteristic* (ROC), together with the corresponding *Area Under the Curve* (AUC). Function clone search is a ranked retrieval problem instead: Querying for function clones returns a list of candidates with decreasing similarity, with the first entry in this list, i.e. the function at rank 1, having the highest similarity. The metric *Precision at Rank 1* (P@1) therefore measures whether the function at rank 1 has the same label as the query function. For single queries, this value is either 0 or 1 and we report the P@1 as averaged across all queries.

B. Feature Extraction and Training Performance

Table I shows the analysis times for the Ghidra-based feature extraction for different categories, together with the obfuscation abbreviations used throughout the evaluation. The listed times are split in analysis time and export time, with the latter being the time needed to extract the normalized disassembly. These measurements offer insight into the expected analysis times on different program categories within Ghidra. Indirect branching and virtualization is the most time intensive w.r.t analysis in Ghidra, as Ghidra tries to resolve target locations of indirect jumps. Export times across the categories are expectedly similar, with the exception of the virtualization categories: The exporter does not parse functions here but traces, which are longer than normal functions.

After export from Ghidra, a vocabulary has to be trained and the disassembly has to be tokenized. The process for vocabulary generation took 10 minutes and the tokenization

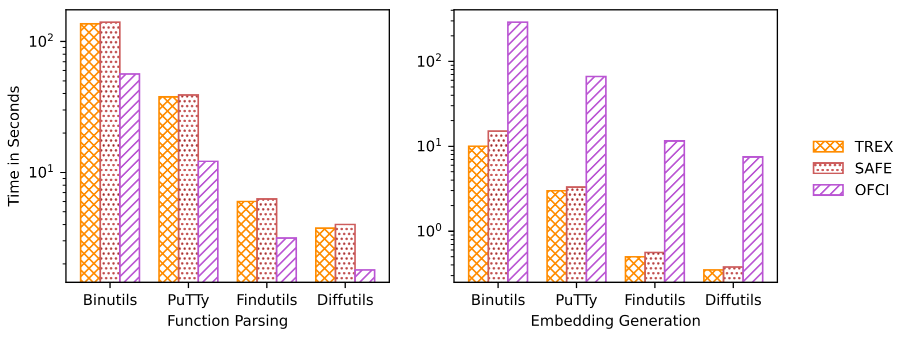


Fig. 2: Function parsing and embedding generation performance.

Category	Analysis	Export
O0	88	8
O1	95	6
O2	90	6
O3	103	7
Bogus Control-Flow (BCF)	102	6
Control-Flow Flattening (CFF)	36	7
Indirect Branching (IBR)	410	5
Basic Block Splitting (SPL)	124	7
Instruction Substitution (SUB)	57	6
EncodeArithmetic (EA)	42	11
Virtualize (VIRT)	248	17
Virtualize + EncodeArithmetic (VIRT-EA)	271	17

TABLE I: Dataset processing times, in minutes.

took 50 minutes, both of which have been executed across the complete dataset. After selection of functions for fine-tuning, the pre-training and fine-tuning datasets are generated, which happens by dumping the tokens from the database into HDF5 files. The pre-training on the remote server was run for exactly one week, during which it achieved 8 epochs on the pre-training dataset. The fine-tuning was executed on the local developer machine and ran for 30 epochs within 20 hours. OFCI is faster than both SAFE and TREX on function parsing; as TREX specifically mentions function parsing, the Ghidra analysis time is not taken into account. Exporting and normalizing the disassembly takes 16s, 4s, 1.5s and 0.9s for Binutils, PuTTy, Findutils and Diffutils respectively. Additional 40s, 8s, 1.6s and 0.9s are required for tokenization; adding these measurements results in the function parsing performance shown in Figure 2. Embedding generation is fast in SAFE compared to OFCI and TREX seemingly only requires a fraction of OFCI's embedding generation time. The main factor for this difference is the measurement of embedding generation on the GPU: The reported values are measured on a system with 8 Nvidia RTX 2080-TI GPUs, while OFCI is evaluated on a machine with only one RTX 2070 Super.

C. Comparison of Model Size

A goal in the creation of OFCI is to cut down on model complexity and decrease the size of the model compared to other recent approaches. The model size in terms of on-disk size and number of trainable model parameters is compared to TREX and SAFE in Table II. We compare against SAFETORCH [17], a PyTorch implementation of the SAFE neural network, which allows easier integration of instruction embeddings into the overall training process. OFCI has 9M trainable parameters, multiplied by 4 (for 32-bit floating point numbers) results in 36 MB of model data. The parameters are stored almost entirely without meta-data or transformation.Performing the same calculation does not hold up for TREX, which is also using PyTorch to store its model, but reaches a size of almost 700 MB, where it should only have 240 MB. This does put OFCI as the leanest approach in comparison to SAFE and TREX, at worst requiring only 17% of the disk space and trainable parameters.

Approach	Size on Disk	Number of Parameters
TREX	696 MB	60.606.229
SAFETORCH	210 MB	55.043.500
OFCI	**35 MB**	**9.136.000**

TABLE II: Comparison of model size of different approaches.

While a significant part of this parameter reduction is due to usage of ALBERT, OFCI also removes specifically designed model parts, as its goal is to evaluate the capabilities of stock NLP models when used in this context. In comparison, SAFE makes use of a specifically modeled RNN, while TREX passes its input streams through a CNN before using ROBERTA. Due to our modeling of the input data as one single token stream, OFCI can represent the full dataset in 3.5GB, whereas TREX requires roughly 40GB for the same dataset due to the required model inputs. When compared to SAFE, OFCI only has roughly 900 tokens in its vocabulary, while the approach taken by SAFE requires 500k tokens; in the case of an unknown instruction, OFCI could still make use of the instruction operands, while SAFE would classify the whole instruction as out-of-vocabulary.

D. Performance on Unobfuscated Data

Due to the lack of available code and models from related approaches we use the results reported by the authors of TREX and other related work for comparison. The robustness of OFCI embeddings on function pairs sampled from different optimization levels can be seen through the ROC curve in Figure 3. On its own, the curve shows good results for the embeddings generated by OFCI, with a high ROC-AUC of 0.95 and no obvious skewing towards true/false positive rates. To put the ROC-AUC score into perspective, Table III compares the AUC scores against TREX. The ROC-AUC for one project

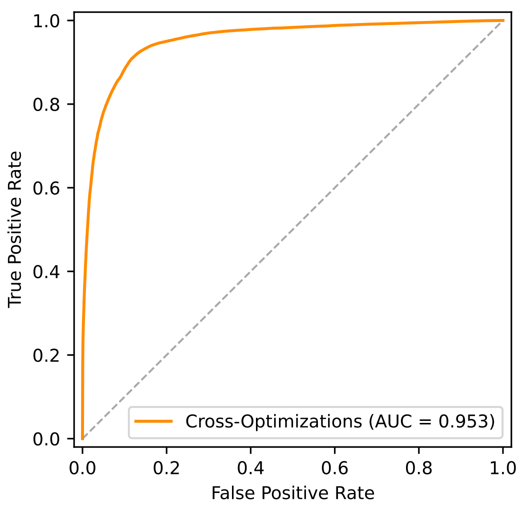


Fig. 3: ROC of function pair similarity across optimizations.

is calculated by sampling 1.000 similar and the same amount of dissimilar pairs from the database, with the standard deviation across 10 runs of sampling being smaller than 0.01 in all cases. The AUC scores of OFCI are worse, as the number of trainable parameters has been significantly reduced and it does not use microtraces introduced by TREX. OFCI performs the worst at LibTomCrypt, with an AUC score of 0.849. The drop in AUC is roughly 15%, while the average decrease is at 7%, which is minor when compared to the 85% reduction of model parameters in OFCI compared to TREX.

Project	TREX	OFCI
Binutils	0.993	0.933
Coreutils	0.992	0.968
Curl	0.993	0.929
Diffutils	0.992	0.951
Findutils	0.992	0.939
GMP	0.993	0.929
ImageMagick	0.993	0.924
Libmicrohttpd	0.994	0.901
LibTomCrypt	0.994	0.849
OpenSSL	0.992	0.952
PuTTY	0.995	0.938
SQLite	0.994	0.916
Zlib	0.991	0.892
Average	0.990	0.925

TABLE III: Comparison of unobfuscated AUC scores.

As the ROC-AUC only shows the performance of classifying functions as similar, it does not show how well OFCI performs on function search. Table IV provides the comparison of Precision@1 reported by TREX and ASM2VEC to OFCI. While the function similarity classification performance of OFCI is on par with other approaches, function search is not. When searching similar functions from binaries with optimization levels O2 and O3, the average performance of OFCI is below TREX and ASM2VEC, at around 50%. While OFCI does not perform well in function search, the precision values are still above the performance of random embeddings. A drop in performance is expected, as OFCI reduces model complexity and the number of trainable parameters, but the reduction margin should be closer to the results on ROC-AUC scores. One potential problem could be the **reduction of model complexity**. With the intention of building on TREX, a model too computation heavy to train on a budget, the focus of OFCI was set to transformer-based models. But in comparison to TREX, which uses ROBERTA [38], OFCI is based on ALBERT [35]. Within the ALBERT paper, the authors devise certain measures to reduce the model complexity (discussed in subsection IV-D), and can achieve on-par performance with BERT [12] on several widely used natural language processing benchmarks, including semantic textual similarity (STS). ROBERTA performs better than simple BERT and thus better than ALBERT in theory. The performance differences discussed in these papers however are in line with the ROC-AUC score differences discussed here, and not with the drastic differences in precision scores. Other points are **reporting of metrics** and **description of evaluation sets** in related work. While there are a variety of popular benchmarks for natural language processing, these benchmarks do not exist for function clone detection and most fields of binary analysis in general, making it hard to objectively compare different approaches on a certain task. This is highlighted by the authors of TREX in a striking manner: For every approach they compare against, i.e. ASM2VEC [13], SAFE [39], GEMINI [63] and BLEX [15], they compare results with different metrics, because with the exception of SAFE, none of the other approaches provides the full source-code or trained model. While there is recent work trying to tackle this issue [34], this benchmark is not yet widely used.

	O2 and O3			O0 and O3		
Project	TREX	ASM2VEC	OFCI	TREX	ASM2VEC	OFCI
Coreutils	0.955	0.929	0.137	0.913	0.781	0.025
Curl	0.961	0.951	0.680	0.894	0.850	0.159
GMP	0.974	0.973	0.748	0.886	0.763	0.219
ImageMagick	0.971	0.971	0.457	0.891	0.837	0.066
LibTomCrypt	0.991	0.991	0.611	0.923	0.921	0.040
OpenSSL	0.982	0.931	0.469	0.914	0.792	0.082
PuTTy	0.956	0.891	0.248	0.926	0.788	0.049
SQLite	0.931	0.926	0.551	0.911	0.776	0.117
Zlib	0.890	0.885	0.465	0.902	0.722	0.329
Average	0.957	0.939	0.485	0.907	0.803	0.121

TABLE IV: Comparison of Precision@1 across optimizations.

E. Performance on Hikari Obfuscated Binaries

The ROC curves of classifying similar function pairs between obfuscations and between all functions in the dataset, including the different optimization levels, are shown in Figure 4. The AUC score on the obfuscated function pairs is 0.963, which puts the classification performance above the previously discussed classifying of function pairs across optimization levels. OFCI performs slightly better on the obfuscation part of the dataset, due to two reasons: The obfuscations provided by Hikari appear to be less drastic than difference between O0 and O3, and the AUC score is calculated between obfuscated function pairs only, meaning there are less differences between the different obfuscations itself.

The performance on the obfuscated data when compared to other approaches can be seen in Table V. First, OFCI is directly compared to TREX on obfuscated function pairs only,

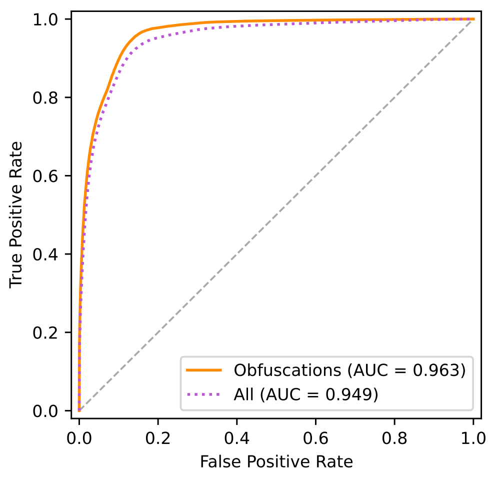


Fig. 4: ROC of obfuscated function pairs and all function pairs.

as the results come from the same dataset. When comparing the AUC scores of obfuscated pairs, Ofci is expectedly outperformed by Trex and again shows a maximum drop of 15% in AUC score. This is again a trade-off for an 85% reduction in the number of trainable model parameters. The second category, the ROC-AUC of all function pairs in the dataset, is taken from the ablation study performed by Trex. It includes the AUC scores calculated for Safe on the Trex dataset and the Trex results when the model is not pre-trained with microtraces, but only with static data, similar to Ofci. This comparison is not entirely objective, as these numbers also include cross-architecture function pairs that are not present in Ofci; however, the comparison is presented here to contextualize the results produced by Ofci. Keeping this in mind, Ofci manages to achieve a higher AUC score than Trex on two projects, manages to generally outperform the version of Trex that does not make use of microtraces and produces results in the same range as Safe, therefore producing good results when only considering ROC-AUC.

	Obfuscated Pairs		All Pairs			
Project	Trex	Ofci	Trex	(w/o microt.)	Safe	Ofci
Binutils	**0.991**	0.947	**0.952**	0.871	0.918	0.942
Coreutils	**0.991**	0.966	0.951	0.900	0.910	**0.965**
Curl	**0.991**	0.940	**0.953**	0.919	0.931	0.924
Diffutils	**0.990**	0.959	**0.952**	0.931	0.918	0.945
Findutils	**0.990**	0.967	**0.961**	0.889	0.910	0.934
GMP	**0.990**	0.869	**0.953**	0.931	0.930	0.852
ImageMagick	**0.989**	0.910	**0.962**	0.889	0.935	0.902
Libmicrohttpd	**0.991**	0.905	**0.951**	0.910	0.917	0.874
LibTomCrypt	**0.991**	0.836	**0.953**	0.900	0.911	0.844
OpenSSL	**0.989**	0.946	**0.952**	0.858	0.925	0.941
PuTTy	**0.990**	0.976	0.941	0.840	0.900	**0.951**
SQLite	**0.993**	0.956	**0.953**	0.850	0.929	0.947
Zlib	**0.990**	0.911	**0.960**	0.810	0.931	0.888
Average	**0.990**	0.929	**0.953**	0.884	0.920	0.916

TABLE V: Comparison of obfuscated AUC scores.

Analyzing the function search Precision@1 results yields the values presented in Table VI, compared to Trex and Asm2Vec. The use case of function search for an obfuscated function is the determination of the original unobfuscated function. To this end, the embeddings of the unobfuscated functions are stored in an index, and the index is queried with the obfuscated functions. This is done for the Bogus Control-Flow (BCF), Control-Flow Flattening (CFF) and Instruction Substitution (SUB) obfuscations, as these are the ones listed by Asm2Vec and Trex. The results are slightly worse compared to searching functions across O0 and O3, highlighting that Hikari obfuscations are still able to increase the complexity of the function beyond the typical capabilities of standard compiler optimizations. Instruction substitution is among the simpler obfuscations with an average Precision@1 of 0.229, while CFF and BCF are on the more complicated end with 0.136 and 0.149 average Precision@1 respectively. Similar to the precision values of the unobfuscated function search, the results are however far behind the reported values of Trex and Asm2Vec. The reasons for this being the same as already described for the unobfuscated version.

Obf.	Approach	GMP	LibTomCrypt	ImageMagick	OpenSSL	Average
Bogus Control-Flow	Trex	0.926	0.938	0.934	0.898	0.924
	Asm2Vec	0.802	0.920	0.933	0.883	0.885
	Ofci	0.158	0.121	0.224	0.093	0.149
Control-Flow Flattening	Trex	0.943	0.931	0.936	0.940	0.930
	Asm2Vec	0.772	0.920	0.890	0.795	0.844
	Ofci	0.169	0.178	0.156	0.043	0.136
Instruction Substitution	Trex	0.949	0.962	0.981	0.980	0.968
	Asm2Vec	0.940	0.960	0.981	0.961	0.961
	Ofci	0.249	0.214	0.283	0.169	0.229

TABLE VI: Comparison of obfuscated P@1 scores.

F. Ablation Study

One additional feature provided by Ofci is the *Call-ID*, where references to another call are added to the call instruction, with the intention of improving the results of the embedding. Within this ablation experiment, the performance of Ofci with and without the Call-ID feature is tested to see whether the call references offer an improvement of the results. The ROC curves of function pairs selected from the obfuscation dataset, grouped by different obfuscation passes, can be seen in Figure 5. Functions that do not perform any calls are excluded, as the Call-ID feature will not have any impact on them.

Project	Call-ID	w/o Call-ID
Binutils	0.131	0.116
Coreutils	0.059	0.057
Curl	0.462	0.500
Diffutils	0.471	0.462
Findutils	0.360	0.326
GMP	0.857	0.714
ImageMagick	0.377	0.333
Libmicrohttpd	0.857	0.714
LibTomCrypt	0.296	0.296
OpenSSL	0.049	0.052
PuTTY	0.108	0.100
SQLite	0.372	0.283
Zlib	0.857	0.857
Average	0.404	0.370

TABLE VII: Comparison of P@1 with and without Call-ID.

The ROC curves in Figure 5 show no visible difference, the robustness of the embeddings appears to be just as good as without the Call-ID feature. However, there are small differences in the ROC-AUC scores of function pairs in favor of

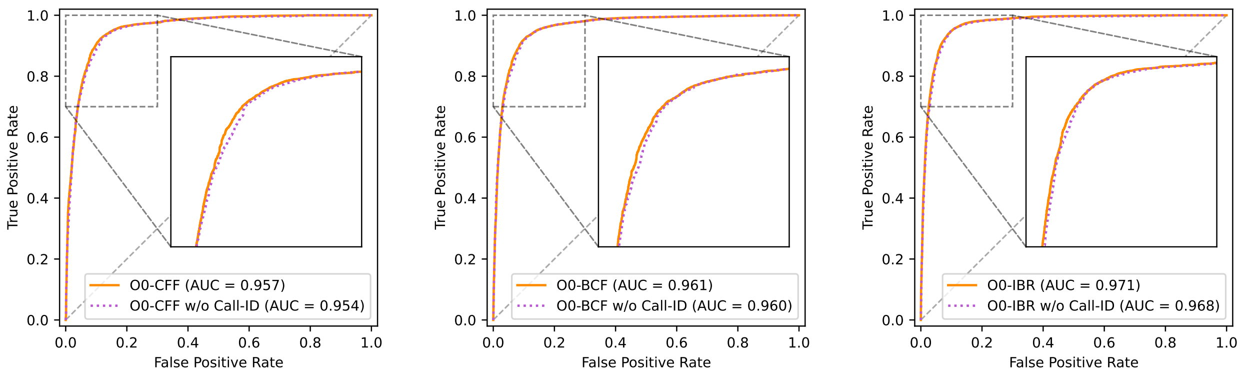


Fig. 5: ROC of various obfuscated function pairs with and without Call-ID.

having the Call-ID feature. For the group of O0-BCF function pairs, this difference is as low as 0.001 difference in AUC, while the O0-CFF and O0-IBR pairs show a bigger difference. While the differences in the ROC-AUC scores are small, the effect of Call-ID can be observed in the Precision@1 scores as well, shown in Table VII for all projects when searching O0 from CFF. The dataset is the same subset used for the ROC curve O0-CFF in Figure 5. While the differences are again small, they are more distinct than the difference in ROC-AUC scores and with the exception of Curl/OpenSSL they are showing that Call-ID can indeed improve the function clone search. The ablation testing shows how the Call-ID feature does appear to be beneficial, with differences in ROC-AUC scores being small, but consistently positive across all tests.

G. *Performance on Tigress Virtualized Examples*

As the second major new addition on top of existing work, OFCI approaches analysis of code obfuscated through virtualization. At the time of writing there are only two references to virtualized code in related work regarding function clone detection: A survey [22] citing virtualized code as an unsolved problem, and ASM2VEC [13] performing analysis on the static code generated by Tigress. They showed that they were still able to detect vulnerabilities from the static code generated by Tigress virtualization, with a true positive rate of 35.8% on a small dataset, but do not discuss function clone detection across virtualization. To the best of our knowledge, OFCI is the first machine learning based approach trying to tackle function clone detection across virtualization through dynamic analysis, i.e. through generating instruction traces of the virtualized code and comparing with these traces; the results can therefore not directly be compared with existing approaches.

The ROC curve for function pair embeddings across this whole dataset can be seen in Figure 6. The results show that the classification of function pairs across virtualized code is not good, but also not completely random. Precision@1 scores are omitted here, as they show results similarly close to the performance of a random classifier. Only in the case of searching virtualized functions against obfuscated or other virtualized functions does OFCI produce results that are slightly above random performance. Therefore, a possible solution to produce better results would be to obfuscate known functions with the same virtualization technique and only then compare them with unknown virtualized function traces.

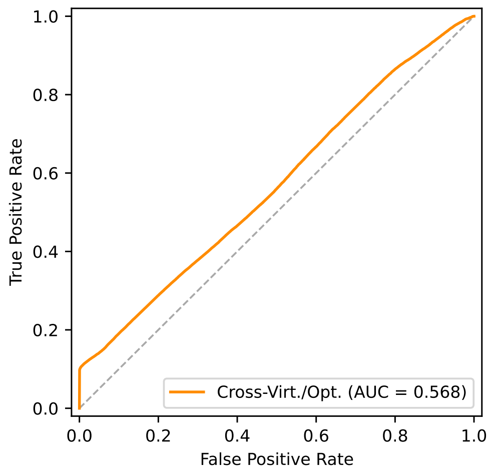


Fig. 6: ROC of all function pairs across optimizations and virtualization.

We identified three main issues when analyzing the performance of our approach when faced with virtualization: The length of the traces, only small differences across traces and the synthetic dataset. Traces represent the longest inputs in the dataset, potentially consisting of hundreds of fragments. Through the analysis of the number of fragments on function pairs obfuscated with Hikari, as shown in Figure 7, we realized that the performance of function similarity classification decreased with the number of fragments per function; a trace made up of hundreds of fragments would therefore take a significant hit in similarity classification performance. This insight proves especially interesting, as most function clone detection approaches simply approach the function length problem by cutting the function at a certain threshold.

Another significant contribution to the performance on virtualized functions is the usage of synthetically created functions used as dataset. As seen in Figure 8, classification between O2 and O3 of the functions shows a perfect classifier, meaning the functions are not affected by optimizations after O2 at all. On the other hand, the model cannot classify pairs between O0 and O3 *at all*, which translates to the similarly bad performance in the virtualization benchmarks. The model

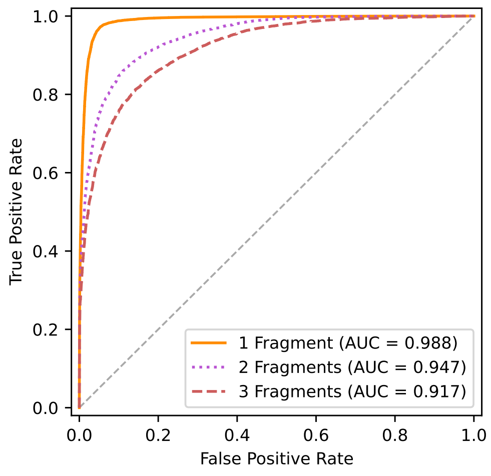


Fig. 7: ROC of obfuscated function pairs by fragment count.

appears to learn certain structural information about the code, which is not present in our simple synthetic functions. As Tigress expects a single source file for compilation, coming up with the quantities of functions needed for machine learning approaches is not simple, as merging real-world C programs into a single source file is far from trivial.

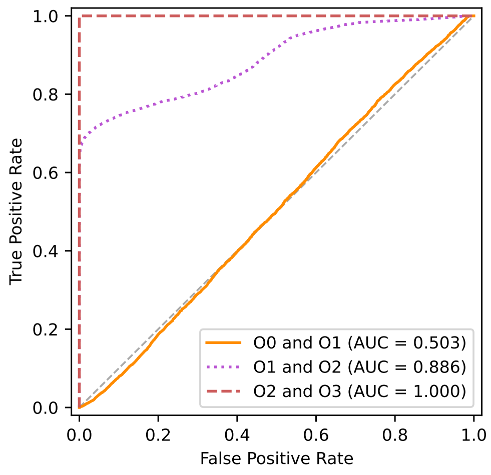


Fig. 8: ROC of source functions in the virtualization dataset.

VII. Conclusion and Future Work

Function clone detection remains a hard problem, with a growing set of machine learning approaches trying to provide new solutions at the cost of being too computation-heavy for production use. Therefore, building on TREX [47], we introduce OFCI to focus on a more lightweight approach to identify obfuscated function clones. OFCI trimmed down the model into a slimmer form of state-of-the-art architectures and implemented the approach as an efficient framework. Every part of this framework makes use of openly available technologies, with Ghidra as core reverse engineering environment and PyTorch as central machine learning library. That makes it possible to have an open end-to-end pipeline for function clone detection, whereas existing approaches rely on proprietary disassemblers like IDA. In addition to openness, the evaluation of OFCI has shown that the framework is able to scale to hundreds of thousands of functions, facilitated by Ghidra's headless analysis modes, outperforming existing approaches in terms of function processing. OFCI has significantly reduced the number of trainable parameters when compared to the most recent and promising related approaches. This does not heavily affect the ability to classify function pairs based on their similarity, which is highlighted in the ROC-AUC scores of the evaluation.

Unfortunately, not all aspects of OFCI have been as successful and the precision of function clone search leads to mixed results in comparison to other approaches. The reasons for these results have been discussed, but further scrutiny and trials are required. Similar issues can be found in the Call-ID and virtualized clone detection features of OFCI. The evaluation showed that Call-ID, i.e. adding function call identification info into the tokenized disassembly, can slightly improve function search performance, while virtualized clone detection through traces does not work in the current form as implemented by OFCI. By extensively analyzing these mixed results, OFCI was able to highlight some general issues in the way related work tackles the function clone detection problem. As future approaches need to tackle these issues, OFCI provides a mature framework for rapid prototyping and testing of new models.

The development of OFCI highlighted the difficulty of working with modern machine learning models. To achieve the reported performance, those approaches rely on large amounts of computing resources, making those approaches harder to reproduce and less viable to be used in production. Therefore, one important issue is to reduce the complexity of new function clone search approaches, while keeping the search performance at the same level. OFCI has already reduced the number of trainable parameters by a large margin, but is still a computation heavy transformer model. Generation of viable datasets is another problem. While existing work is mostly based on using Hikari as an obfuscator, creating large datasets utilizing virtualization based obfuscation is not as easily achievable, since Tigress requires a high level of manual effort. However, generating good datasets is a vital part for every approach based on machine learning, for both training and verification. Since OFCI also supports instruction traces through Intel Pin, another interesting prospect is the integration of additional information from dynamic traces, similar to the microtraces used by TREX. And finally, it is worth investigating whether using intermediate representations or raw bytes instead of disassembly is viable as input for binary code similarity models.

Acknowledgment

We would like to thank our shepherd Dipanjan Das for helping us incorporate the reviewer's comments. This research was funded by the FFG Bridge 1 iSaFe and FFG Bridge Young Scientists AutoHoney grants, the FFG Industrial PhD project AIM@IoT, the FiBack project funded by the Austrian defense research program FORTE, as well as the competence center SBA Research (SBA-K1) funded by COMET.

References

[1] N. S. Agency, "Ghidra," https://ghidra-sre.org/, accessed: 2021-04-04.

[2] B. Bichsel, V. Raychev, P. Tsankov, and M. Vechev, "Statistical Deobfuscation of Android Applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 343–355. [Online]. Available: https://doi.org/10.1145/2976749.2978422

[3] T. Blazytko, M. Contag, C. Aschermann, and T. Holz, "Syntia: Synthesizing the Semantics of Obfuscated Code," in *Proceedings of the 26th USENIX Conference on Security Symposium*, ser. SEC'17. USA: USENIX Association, 2017, pp. 643–659.

[4] C. Blichmann, "BinDiff now available for free," https://security.googleblog.com/2016/03/bindiff-now-available-for-free.html, accessed: 2021-05-12.

[5] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "BinGo: Cross-Architecture Cross-OS Binary Search," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, pp. 678–689. [Online]. Available: https://doi.org/10.1145/2950290.2950350

[6] C. Cimpanu, "Go malware is now common, having been adopted by both APTs and e-crime groups," https://www.zdnet.com/article/go-malware-is-now-common-having-been-adopted-by-both-apts-and-e-crime-groups/, accessed: 2021-05-15.

[7] C. Collberg, "The Tigress C Obfuscator," https://tigress.wtf, accessed: 2021-05-13.

[8] J. Crussell, C. Gibler, and H. Chen, "AnDarwin: Scalable Detection of Semantically Similar Android Applications," in *Computer Security – ESORICS 2013*, J. Crampton, S. Jajodia, and K. Mayes, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 182–199.

[9] R. David, L. Coniglio, and M. Ceccato, "QSynth-A Program Synthesis based Approach for Binary Code Deobfuscation," in *Workshop on Binary Analysis Research*, 2020.

[10] Y. David, N. Partush, and E. Yahav, "Similarity of Binaries through Re-Optimization," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: Association for Computing Machinery, 2017, pp. 79–94. [Online]. Available: https://doi.org/10.1145/3062341.3062387

[11] Y. David and E. Yahav, "Tracelet-Based Code Search in Executables," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 349–360. [Online]. Available: https://doi.org/10.1145/2594291.2594343

[12] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 2019, pp. 4171–4186.

[13] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 472–489.

[14] S. H. Ding, B. C. Fung, and P. Charland, "Kam1n0: MapReduce-Based Assembly Clone Search for Reverse Engineering," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 461–470. [Online]. Available: https://doi.org/10.1145/2939672.2939719

[15] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket execution: Dynamic similarity testing for program binaries and components," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 303–317.

[16] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code." in *NDSS*, vol. 52, 2016, pp. 58–79.

[17] Facebook, "SafeTorch," https://github.com/facebookresearch/SAFEtorch, accessed: 2021-12-06.

[18] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable Graph-Based Bug Search for Firmware Images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 480–491. [Online]. Available: https://doi.org/10.1145/2976749.2978370

[19] J. Gao, X. Yang, Y. Fu, Y. Jiang, H. Shi, and J. Sun, "VulSeeker-pro: Enhanced Semantic Learning Based Binary Vulnerability Seeker with Emulation," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, pp. 803–808. [Online]. Available: https://doi.org/10.1145/3236024.3275524

[20] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "VulSeeker: A Semantic Learning Based Vulnerability Seeker for Cross-Platform Binary," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: Association for Computing Machinery, 2018, pp. 896–899. [Online]. Available: https://doi.org/10.1145/3238147.3240480

[21] Google, "M-Bert," https://github.com/google-research/bert/blob/master/multilingual.md, accessed: 2022-02-26.

[22] I. U. Haq and J. Caballero, "A Survey of Binary Code Similarity," *ACM Comput. Surv.*, vol. 54, no. 3, Apr. 2021. [Online]. Available: https://doi.org/10.1145/3446371

[23] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. Vechev, "Debin: Predicting Debug Information in Stripped Binaries," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 1667–1680. [Online]. Available: https://doi.org/10.1145/3243734.3243866

[24] Hex-Rays, "IDA Pro," https://www.hex-rays.com/products/ida/, accessed: 2021-04-04.

[25] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 11 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[26] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Binary Code Clone Detection across Architectures and Compiling Configurations," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, 2017, pp. 88–98.

[27] H. Huang, A. M. Youssef, and M. Debbabi, "BinSequence: Fast, Accurate and Scalable Binary Code Reuse Detection," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 155–166. [Online]. Available: https://doi.org/10.1145/3052973.3052974

[28] P. Indyk and R. Motwani, "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality," in *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, ser. STOC '98. New York, NY, USA: Association for Computing Machinery, 1998, pp. 604–613. [Online]. Available: https://doi.org/10.1145/276698.276876

[29] Intel, "Pin," https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html, accessed: 2021-10-08.

[30] J. Jiang, G. Li, M. Yu, G. Li, C. Liu, Z. Lv, B. Lv, and W. Huang, "Similarity of Binaries Across Optimization Levels and Obfuscation," in *Computer Security – ESORICS 2020*, L. Chen, N. Li, K. Liang, and S. Schneider, Eds. Cham: Springer International Publishing, 2020, pp. 295–315.

[31] W. Jin, S. Chaki, C. Cohen, A. Gurfinkel, J. Havrilla, C. Hines, and P. Narasimhan, "Binary Function Clustering Using Semantic Hashes," in *2012 11th International Conference on Machine Learning and Applications*, vol. 1, 2012, pp. 386–391.

[32] J. Johnson, M. Douze, and H. Jégou, "Billion-Scale Similarity Search with GPUs," *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2021.

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

[34] D. Kim, E. Kim, S. K. Cha, S. Son, and Y. Kim, "Revisiting Binary Code Similarity Analysis using Interpretable Feature Engineering

and Lessons Learned," *CoRR*, vol. abs/2011.10749, 2020. [Online]. Available: https://arxiv.org/abs/2011.10749

[35] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "ALBERT: A Lite BERT for Self-supervised Learning of Language Representations," *CoRR*, vol. abs/1909.11942, 2019. [Online]. Available: http://arxiv.org/abs/1909.11942

[36] Q. Le and T. Mikolov, "Distributed Representations of Sentences and Documents," in *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ser. ICML'14. JMLR.org, 2014, pp. 1188–1196.

[37] Y. Lee, H. Kwon, S.-H. Choi, S.-H. Lim, S. H. Baek, and K.-W. Park, "Instruction2vec: Efficient Preprocessor of Assembly Code to Detect Software Weakness with CNN," *Applied Sciences*, vol. 9, no. 19, 2019. [Online]. Available: https://www.mdpi.com/2076-3417/9/19/4086

[38] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "RoBERTa: A Robustly Optimized BERT Pretraining Approach," *CoRR*, vol. abs/1907.11692, 2019. [Online]. Available: http://arxiv.org/abs/1907.11692

[39] L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni, "SAFE: Self-Attentive Function Embeddings for Binary Similarity," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, R. Perdisci, C. Maurice, G. Giacinto, and M. Almgren, Eds. Cham: Springer International Publishing, 2019, pp. 309–329.

[40] D. McKee, N. Burow, and M. Payer, "Software Ethology: An Accurate and Resilient Semantic Binary Analysis Framework," *CoRR*, vol. abs/1906.02928, 2019. [Online]. Available: http://arxiv.org/abs/1906.02928

[41] G. Menguy, S. Bardin, R. Bonichon, and C. de Souza Lima, "AI-based Blackbox Code Deobfuscation: Understand, Improve and Mitigate," *CoRR*, vol. abs/2102.04805, 2021. [Online]. Available: https://arxiv.org/abs/2102.04805

[42] T. Micro, "URSNIF, EMOTET, DRIDEX and BitPayme Linked by Loader," https://www.trendmicro.com/en_us/research/18/l/ursnif-emotet-dridex-and-bitpaymer-gangs-linked-by-a-similar-loader.html, accessed: 2021-01-15.

[43] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient Estimation of Word Representations in Vector Space," in *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2013. [Online]. Available: http://arxiv.org/abs/1301.3781

[44] B. H. Ng and A. Prakash, "Expose: Discovering Potential Binary Code Re-use," in *2013 IEEE 37th Annual Computer Software and Applications Conference*, 2013, pp. 492–501.

[45] J. Oakley and S. Bratus, "Exploiting the Hard-Working DWARF: Trojan and Exploit Techniques with No Native Executable Code." *WOOT*, vol. 10, pp. 2 028 052–2 028 063, 2011.

[46] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[47] K. Pei, Z. Xuan, J. Yang, S. Jana, and B. Ray, "Trex: Learning Execution Semantics from Micro-Traces for Binary Similarity," *CoRR*, vol. abs/2012.08680, 2020. [Online]. Available: https://arxiv.org/abs/2012.08680

[48] D. Peng, S. Zheng, Y. Li, G. Ke, D. He, and T.-Y. Liu, "How could Neural Networks understand Programs?" in *Proceedings of the 38th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. Meila and T. Zhang, Eds., vol. 139. PMLR, 18–24 Jul 2021, pp. 8476–8486. [Online]. Available: https://proceedings.mlr.press/v139/peng21b.html

[49] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-Architecture Bug Search in Binary Executables," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 709–724.

[50] Quarkslab, "An Experimental Study of Different Binary Exporters," https://blog.quarkslab.com/an-experimental-study-of-different-binary-exporters.html, accessed: 2021-04-04.

[51] K. Redmond, L. Luo, and Q. Zeng, "A Cross-Architecture Instruction Embedding Model for Natural Language Processing-Inspired Binary Code Analysis," in *NDSS Workshop on Binary Analysis Research (BAR)*, 2019.

[52] N. Reimers, I. Gurevych, N. Reimers, I. Gurevych, N. Thakur, N. Reimers, J. Daxenberger, I. Gurevych, N. Reimers, I. Gurevych *et al.*, "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2019.

[53] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, "Detecting Code Clones in Binary Executables," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ser. ISSTA '09. New York, NY, USA: Association for Computing Machinery, 2009, pp. 117–128. [Online]. Available: https://doi.org/10.1145/1572272.1572287

[54] P. Shirani, L. Wang, and M. Debbabi, "BinShape: Scalable and Robust Binary Library Function Identification Using Function Shape," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, M. Polychronakis and M. Meier, Eds. Cham: Springer International Publishing, 2017, pp. 301–324.

[55] F.-H. Su, J. Bell, G. Kaiser, and R. Baishakhi, "Deobfuscating Android Applications through Deep Learning," https://mice.cs.columbia.edu/getTechreport.php?techreportID=1632, accessed: 2021-01-15.

[56] T. G. D. Team, "The Go Project," https://golang.org/project/, accessed: 2021-05-15.

[57] O. Technologies, "Themida - Advanced Windows Software Protection System," https://oreans.com/themida.php, accessed: 2021-05-13.

[58] R. Tofighi-Shirazi, I.-M. Asavoae, P. Elbaz-Vincent, and T.-H. Le, "Defeating Opaque Predicates Statically through Machine Learning and Binary Analysis," in *Proceedings of the 3rd ACM Workshop on Software Protection*, ser. SPRO'19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 3–14. [Online]. Available: https://doi.org/10.1145/3338503.3357719

[59] J. Upchurch and X. Zhou, "Malware provenance: code reuse detection in malicious software at scale," in *2016 11th International Conference on Malicious and Unwanted Software (MALWARE)*, 2016, pp. 1–9.

[60] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, u. Kaiser, and I. Polosukhin, "Attention is All You Need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, pp. 6000–6010.

[61] VMProtect, "VMProtect Software Protection," http://vmpsoft.com/, accessed: 2021-05-13.

[62] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, "Transformers: State-of-the-Art Natural Language Processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. [Online]. Available: https://www.aclweb.org/anthology/2020.emnlp-demos.6

[63] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural Network-Based Graph Embedding for Cross-Platform Binary Code Similarity Detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 363–376.

[64] N. Zhang, "Hikari Obfuscator," https://github.com/HikariObfuscator/Hikari, accessed: 2021-10-05.

[65] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural Machine Translation Inspired Binary Code Similarity Comparison beyond Function Pairs," in *Proceedings of the 2019 Network and Distributed Systems Security Symposium (NDSS)*, 2019.

Project	O0	O1	O2	O3	BCF	CFF	IBR	SPL	SUB	*Total*
Binutils	57.527	43.901	42.166	39.096	62.981	62.981	62.981	62.981	62.981	*497.595*
Busybox	3.321	2.108	1.831	1.854	3.160	3.271	3.483	3.282	3.282	*25.592*
Coreutils	96.696	74.505	73.013	69.207	17.331	17.331	17.343	17.331	17.331	*400.088*
Curl	5.390	742	727	661	1.002	1.002	1.002	1.002	1.002	*12.530*
Diffutils	3.959	2.500	2.829	2.614	848	848	850	848	848	*16.144*
Findutils	5.055	2.671	3.625	3.286	1.400	1.400	1.404	1.400	1.400	*21.641*
GMP	789	698	690	665	782	782	782	782	782	*6.752*
ImageMagick	4.456	2.389	2.380	2.308	4.447	4.447	4.447	4.447	4.447	*33.768*
Libmicrohttpd	200	176	171	161	200	200	204	200	200	*1.712*
LibTomCrypt	794	749	743	726	794	794	795	794	794	*6.983*
OpenSSL	12.178	11.197	11.077	10.755	10.745	10.871	12.176	10.749	11.476	*101.224*
PuTTy	8.104	5.741	5.666	5.387	8.154	8.154	8.154	8.154	8.154	*65.668*
SQLite	2.192	1.525	1.367	1.181	2.183	2.183	2.183	2.183	2.183	*17.180*
Zlib	154	139	124	115	154	154	154	154	154	*1.302*

TABLE VIII: Extracted functions from the TREX dataset.

APPENDIX A
DATASET

Our dataset consists of two distinct parts, which are trained together but evaluated separately. These parts are:

1) The amd64 part of the dataset published by the authors of TREX [47].
2) An additional set of functions used for evaluating the performance on VM-obfuscated functions.

The first part, the dataset taken from TREX, is already discussed in detail in their paper. We selected the amd64 part of their dataset, as this part additionally contains binaries obfuscated with Hikari. From what we gathered through manual analysis, the obfuscated binaries are based on the O0 version of the respective binary or have been compiled with O0. Through extraction with our Ghidra exporter, the retrieved function counts differ from the numbers reported by TREX, with the number of extracted functions shown in Table VIII.

While this accounts for smaller differences, the differences in Coreutils, Binutils, etc., can be explained by TREX focussing on one specific version of these utils, while the dataset actually contains multiple versions. OFCI uses all versions in the dataset and marks a function as similar even if the version is different. Whenever we refer to a project that consists of multiple binaries, e.g. Coreutils and Binutils, we refer to *all* binaries contained in these projects, for OpenSSL we use the shared library itself.

The second part, the dataset used for evaluating performance on VM-obfuscated functions, is created synthetically, as it is not trivial to apply Tigress VM-obfuscations to real-world programs. We use a Python script to generate functions containing `if`-conditions and arithmetic/bitwise operations, similar to programs generated in VM deobfuscation research [9], [3].

An example program can be seen in Listing 1; the constants are picked in a way that a trace will always capture the longest path, i.e. evaluate the `if`-condition to true, based on fixed input parameters. A total of 7k functions is generated this way, compiled on all optimization levels, obfuscated using Tigress *Virtualization*, *EncodeArithmetic* and the combination of *Virtualization* and *EncodeArithmetic* together, resulting in a total of 49k binary functions.

```
uint64_t fn_00000849 (uint64_t a, uint64_t b,
        uint64_t c,uint64_t d, uint64_t e) {
    if ((((((((~e)^e)&2528754653ULL)
            &c)-e)+c)*d)> 706126599ULL) {
        if ((-(((-e)^(~a))&b)) > 4055914716ULL) {
            return (~(-((((-(-3380300720ULL))^e)
                &1282233943ULL)+a)));
        }
        return 2;
    }
    return 1;
}
```

Listing 1: Generated program for VM obfuscation analysis.

For pre-training the whole function database is used and 1.5M function fragments are randomly selected. As our definition of similarity is based on function names, for fine-tuning we first select a subset of function names, to ensure that the functions and the similar functions we use for the evaluation have never been used in the fine-tuning process. Our database contains 190k unique function names, out of which 56k (30%) are randomly selected for fine-tuning. In the next step, 500k function fragments are sampled from function pairs using the selected names. To this end, one positive and two negative function pairs are randomly selected, their fragments counted and added to the total number of fragments for their respective optimization/obfuscation category. The next function pair will be selected based on the already generated number of fragments per category, with the goal of distributing the number of fragments equally across categories until 500k fragment pairs have been generated. Out of these 500k fragment pairs, 200k are used for training and 100k for validation during training.