How Machine Learning Is Solving the Binary Function Similarity Problem (Additional Technical Information)

Andrea Marcelli
Cisco Systems, Inc.

Mariano Graziano
Cisco Systems, Inc.

Xabier Ugarte-Pedrero
Cisco Systems, Inc.

Yanick Fratantonio
Cisco Systems, Inc.

Mohamad Mansouri
EURECOM

Davide Balzarotti
EURECOM

This document provides additional technical details and context for the paper "How Machine Learning Is Solving the Binary Function Similarity Problem," published at USENIX Security 2022 [14]. In particular, we cover additional technical details about the datasets and the approached we considered for our measurement study. Additional supporting material, the actual datasets, and other various artifacts can be found at https://github.com/Cisco-Talos/binary_function_similarity.

1 Dataset details

Table 2 contains the complete list of projects that have been selected for Dataset-1 and Dataset-2, including the project versions and the link to the source code. The binaries in Dataset-1 have been compiled using `GCC-4.8.5` (June 23, 2015), `GCC-5.5.0` (Oct 10, 2017), `GCC-7.4.0` (Dec 6, 2018), `GCC-9.2.1` (Aug 12, 2019), `CLANG/LLVM-3.5.2` (Apr 02, 2015), `CLANG/LLVM-5.0.1` (Dec 21, 2017), `CLANG/LLVM-7.0.0` (Sept 19, 2018), and `CLANG/LLVM-9.0.0` (Sept 19, 2019). Dataset-2 is a subset of the one used in Trex [22], and it was compiled with `GCC-7.5` (Nov 14, 2019).

Table 1: Vulnerability test details.

Query functions	CVE	Netgear R7000	TP-Link Deco-M4
CMS_decrypt	CVE-2019-1563	x	x
PKCS7_dataDecode	CVE-2019-1563	x	x
MDC2_Update	CVE-2016-6303	x	
BN_bn2dec	CVE-2016-2182	x	x
X509_NAME_oneline	CVE-2016-2176		x
EVP_EncryptUpdate	CVE-2016-2106		x
EVP_EncodeUpdate	CVE-2016-2105		x
SRP_VBASE_get_by_user	CVE-2016-0798		x
BN_dec2bn	CVE-2016-0797		x
BN_hex2bn	CVE-2016-0797		x

Table 1 contains the details of the functions we selected for the vulnerability discovery use case, including the CVEs and the affected firmware versions.

Figure 1 on the left shows an histogram with the number of basic blocks for the selected functions, limited to 100 for figure clarity. Only 3.5% of functions have more than 100 BBs. On the right, the histogram shows the number of instructions for the selected functions, limited to 500 for figure clarity. Only 4.5% of the functions had more than 500 assembly instructions.

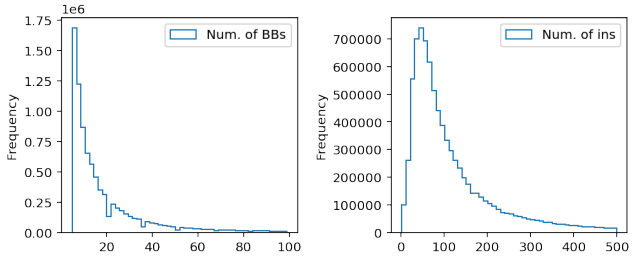

Figure 1: Histogram of the number of basic blocks (left) and instructions (right) for the selected functions.

2 Implementation details for the selected approaches

Catalog1. In order to compute the Catalog1 signature, we used the implementation from the FCatalog IDA plugin [28], in particular we chose the one written in C, as it provides a substantial speed-up compared to the Python version. We wrote an IDA plugin to iterate over each basic block of a function and to extract the corresponding bytes via the `get_bytes` API. The Catalog1 algorithm implements the MinHash Locality Sensitive Hashing [1], and outputs a set of 32-bit integers. The size of the set depends on the number of hash functions, limited to a maximum of 128 in the FCatalog code. The bigger the set, the more accurate the results are likely to be, but the similarity computation and the storage are more expensive. The similarity between two Catalog1 signatures is computed via the Jaccard similarity.

FunctionSimSearch. The official code is available on GitHub [7]. For the feature extraction we used the official IDA plugin, but we updated the code to use Capstone as a

Table 2: Details on specific versions we used for each project in our dataset.

Dataset	Project	Version	URL
Dataset 1	UnRAR	5.5.3	https://www.rarlab.com/rar/unrarsrc-5.5.3.tar.gz
	ClamAV	0.102.0-devel	https://github.com/Cisco-Talos/clamav-devel.git (branch: dev/0.102) (commit: ee5a160840309eb933e73f4268a1e67f9e77961d)
	Curl	7.67.0	https://github.com/curl/curl (branch: master) (commit: d81dbae19f8876ad472e445d89760970c79cceaa)
	Nmap	7.80	https://nmap.org/dist/nmap-7.80.tar.bz2
	OpenSSL	3.0	https://github.com/openssl/openssl.git (branch:master) (commit: 187753e09ceab4c85a0041844e749658e8f712d3)
	Zlib	1.2.11	https://github.com/madler/zlib (branch:master) (commit: cacf7f1d4e3d44d871b605da3b647f07d718623f)
	Z3	4.8.7	https://github.com/Z3Prover/z3 (branch: master) (commit: 0b486d26daea05f918643a9d277f12027f0bc2f6)
Dataset 2	Binutils	2.34	https://github.com/CUMLSec/trex
	Coreutils	8.32	
	Diffutils	3.7	
	Findutils	4.7.0	
	GMP	6.2.0	
	ImageMagick	7.0.10	
	Libmicrohttpd	0.9.71	
	LibTomCrypt	1.18.2	
	PuTTy	0.74	
	SQLite	3.34.0	

disassembler like the rest of the approaches. The project leverages the SimHash algorithm [2] to compute the function signature, which works by hashing individual features and then aggregating them into a single vector. For the SimHash calculation, we used the FunctionSimSearch C++ toolkit with Python bindings. Since the C++ software was originally written to work with the Dyninst binary analysis framework [23], we updated the regular expression that extracts the immediates to be compatible with the Capstone syntax. The implementation uses two sets of weights: the first one (hardcoded) determines the contribution of each type of feature (graphlets, mnemonics and immediates), while the other determines per simhash-bit weights. The second one can be also learnt using a custom loss function that is specifically engineered to maximize the SimHash similarity learning. Nevertheless, we decided to not include this training in the evaluation as its implementation is not optimized and the model suffers from limited generalization capabilities, as described in the original blog post [8]. The output is a 128-bit fuzzy-hash and the hamming distance is used as a similarity metric.

Gemini. The Gemini neural network is made available by the authors on GitHub [29]. However, we decided to use a similar but more comprehensive re-implementation released by Massarelli et al. [13], which allowed us to minimize the evaluation differences related to the implementations of models based on the same network architecture [3]. Unfortunately, the feature extraction part, which is a crucial component and it is the same as in Genius [10], is not available in the GitHub repository. We contacted the authors, but we received no answer. Gemini extracts seven numerical features: six at basic block level, and one at CFG level. Those at basic block level require to classify the instruction set in six main categories. We found several repositories online and partial implementations that attempt to replicate this phase, but after a detailed analysis we found those classifications to be either incomplete or wrong. We also realized that the proposed classification is often subjective and the same instruction may fall in different categories. We opted for a best-effort manual classification of 2,962 instructions for the three main instruction sets (x86, ARM, MIPS) in the above categories. The similarity between two function embeddings is computed via cosine distance.

SAFE and Massarelli et al. (BAR 2019). Massarelli et al. published two papers on the topic [13, 16] and they released the Tensorflow implementation of the two models on GitHub [15, 17]. Both models take as input normalized assembly instructions: SAFE [16] uses a self-attentive neural network to produce a function level embedding, while in the other work [13] the authors combine instructions at the basic block level using different strategies (arithmetic, weighted mean and RNN) and then a GNN learns the function embedding. In the neural network the normalized assembly instructions are translated into instruction embeddings using a pre-trained word2vec [18] model, alternatively they can be learned end-to-end at the cost of a longer training phase. We implemented the feature extraction via IDA scripts, the assembly normalization on top of Capstone, and we used Gensim [26] to train the word2vec model. The Tensorflow implementation of the two models was integrated in our common codebase. We also contacted the authors to clarify some implementation details and we received full support. Both models produce one embedding per function and use the cosine distance to compute the function similarity.

Li et al. 2019. The implementation of two models presented by Li et al. [12], the Graph Neural Network (GNN) and the Graph Matching Network (GMN), is available via an IPython notebook in the DeepMind repository [9]. The two neural networks are implemented on top of Sonnet [4], a wrapper library around Tensorflow, but the implementation is generic and does not include the evaluation for the function similarity use case. We contacted the first author and he gave us some guidance on how to implement the missing parts. For the feature extraction, we relied on an IDA script using the Capstone disassembler and IDA's API to retrieve the function CFG. In order to compute the bag of words of the basic block opcodes,

we selected the list of the 200 most common opcodes from the training and validation dataset, then the same list was applied to the testing dataset. The GNN model uses the Euclidean distance to compute the similarity between two function embeddings, while the GMN model uses the inner product.

Zeek. We contacted the authors of Zeek [24], but we were not able to get access to the code or to get any additional information. We re-implemented the Zeek approach starting from the details presented in the paper. We lifted each binary to the VEX IR (using PyVex [25]) and we implemented the dataflow analysis, backward slicer, strand extraction, and normalization on top of it. The neural network has been implemented using Tensorflow, with the same codebase as the other machine-learning approaches. Throughout the implementation we used the same parameters as described in the paper, but the optimizer was not specified, thus we used the AdamOptimizer with a learning rate of 0.001, similarly to the others. From each basic block, different strands are extracted, normalized using some syntactic rules, and converted into a numerical representation using the bag of words (BoW) of the hashed strands (MD5 modulo 1024). In order to learn the cross-architecture similarity, Zeek uses a 2 hidden layers fully-connected neural network. The network has 2048 input neurons, that is the concatenation of the two 1024 BoW vectors, and 2 output neurons. The output is a softmax probability of the function being similar (top neuron), or being different (bottom neuron).

Asm2Vec. The model described in the Asm2Vec [5] paper is a variant of the PV-DM (paragraph2vec [11]) model. The public repository of the Kam1n0-Community [6] claims to include an implementation of the Asm2Vec model, as described in their paper. However, after carefully inspecting the code, we only found the PV-DM and PV-DBOW variants of paragraph2vec. We contacted the first author to highlight the issue, but we did not receive any answer. We finally reimplemented the Asm2Vec model in Gensim [26], which is the reference implementation for all embedding models that derive from word2vec. We further improved the framework and implemented a multi-process inference to speed up the evaluation phase. In order to address the out-of-vocabulary problem (OOV), each assembly instruction is split in instruction mnemonic and operands, then each token is associated with a unique embedding. Finally the model is trained using a rolling window over the function assembly instructions and the same negative sampling loss presented in word2vec [18]. The cosine distance is used to compute the similarity between two function embeddings.

Trex. The Trex [22] code to pre-train and fine-tune the model is available on GitHub [21], together with a small example dataset. Since the model was written on top of Fairseq [19], a sequence modeling toolkit for PyTorch [20], it was not possible to rewrite it in Tensorflow similarly to the other machine-learning implementations. Moreover, due to the large number of model parameters, it is very expensive to pre-train and fine-tune Trex. Indeed the authors used a workstation with 8 Nvidia RTX 2080-Ti GPUs. As an example, fine tuning the model for 30 epochs on Google Colab Pro would have required more than a week. We contacted the authors and they shared the fine tuned model that they have used for the tests in the paper. For the inference, we were able to use the single Nvidia RTX 2080-Ti GPU, which gives about a 10x speedup compared to running the model on the CPU. We implemented the feature extraction on top of IDA, the Capstone disassembler, and we used the Trex preprocessing code to create the function traces in the expected format. Similarly to other models, Trex uses the cosine distance to compute the similarity between two function embeddings.

Order Matters and CodeCMR. The model implementation for the papers Order Matters [30] and CodeCMR [31] is not public. We contacted the authors asking for the code, but they could not share it. However, they informed us that they had a new approach (CodeCMR) that was outperforming Order Matters, although CodeCMR was originally created to solve the problem of matching binary functions with the corresponding source code. They also revealed that the model proposed in CodeCMR was powering the new BinaryAI platform [27]. Given the complexity of the solution proposed in Order Matters, we were not confident in the faithfulness of any reproduction attempt of this work, and it would have been challenging to draw any meaningful conclusion. We finally agreed that they would have retrained and tested their model on our datasets, sharing with us the list of function embeddings for the testing data. In order to compute the function similarity, Order Matters and CodeCMR use the cosine distance between function embeddings.

References

[1] Andrei Z Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*, pages 21–29. IEEE, 1997.

[2] Moses S Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388, 2002.

[3] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In *International conference on machine learning*, pages 2702–2711, 2016.

[4] Google DeepMind. Sonnet. https://github.com/deepmind/sonnet.

[5] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe 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)*, pages 472–489, San Francisco, CA, USA, May 2019. IEEE.

[6] Steven H. H. Ding and Miles Q. Li. The Kam1n0 Assembly Analysis Platform Topics. https://github.com/McGill-DMaS/Kam1n0-Community.

[7] Thomas Dullien. FunctionSimSearch. https://github.com/googleprojectzero/functionsimsearch.

[8] Thomas Dullien. Searching statically-linked vulnerable library functions in executable code . https://googleprojectzero.blogspot.com/2018/12/searching-statically-linked-vulnerable.html.

[9] Li et al. graph_matching_networks.ipynb. https://github.com/deepmind/deepmind-research/tree/master/graph_matching_networks.

[10] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable Graph-based Bug Search for Firmware Images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 480–491, Vienna Austria, October 2016. ACM.

[11] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196, 2014.

[12] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. Graph matching networks for learning the similarity of graph structured objects. In *International conference on machine learning*, pages 3835–3845. PMLR, 2019.

[13] Luca Massarelli, Giuseppe A. Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. Investigating Graph Embedding Neural Networks with Unsupervised Features Extraction for Binary Analysis. In *Proceedings 2019 Workshop on Binary Analysis Research*, San Diego, CA, 2019. Internet Society.

[14] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. How Machine Learning Is Solving the Binary Function Similarity Problem. In *USENIX Security Symposium*, 2022.

[15] Luca Massarelli. Code for the paper Investigating Graph Embedding Neural Networks with Unsupervised Features Extraction for Binary Analysis. https://github.com/lucamassarelli/Unsupervised-Features-Learning-For-Binary-Similarity.

[16] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. Safe: Self-attentive function embeddings for binary similarity. In *Proceedings of Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2019.

[17] Luca Massarelli and Giuseppe Antonio Di Luna. Code for the paper SAFE: Self-Attentive Function Embeddings for binary similarity. https://github.com/gadiluna/SAFE.

[18] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26:3111–3119, 2013.

[19] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling. In *Proceedings of NAACL-HLT 2019: Demonstrations*, 2019.

[20] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[21] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. Code for the paper TREX: Learning Execution Semantics from Micro-Traces for Binary Similarity. https://github.com/CUMLSec/trex.

[22] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. Trex: Learning execution semantics from micro-traces for binary similarity. *arXiv preprint arXiv:2012.08680*, 2020.

[23] Dyninst Project. DyninstAPI: Tools for binary instrumentation, analysis, and modification. https://github.com/dyninst/dyninst.

[24] Noam Shalev and Nimrod Partush. Binary Similarity Detection Using Machine Learning. In *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security - PLAS '18*, pages 42–47, Toronto, Canada, 2018. ACM Press.

[25] Yan Shoshitaishvili. PyVEX. https://github.com/angr/pyvex.

[26] RARE Technologies. Gensim: Topic modelling for human. https://radimrehurek.com/gensim/index.html.

[27] Tencent. BinaryAI Python SDK. https://github.com/binaryai/sdk.

[28] xorpd. FCatalog. https://www.xorpd.net/pages/fcatalog.html.

[29] Xiaojun Xu. DNN Binary Code Similarity Detection. https://github.com/xiaojunxu/dnn-binary-code-similarity.

[30] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou ouang, and Shi Wu. Order Matters: Semantic-Aware Neural Networks for Binary Code Similarity Detection. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(01):1145–1152, April 2020.

[31] Zeping Yu, Wenxin Zheng, Jiaqi Wang, Qiyi Tang, Sen Nie, and Shi Wu. Codecmr: Cross-modal retrieval for function-level binary source code matching. *Advances in Neural Information Processing Systems*, 33, 2020.