Binary Similarity Detection Using Machine Learning

Noam Shalev
Cornell Tech
New York, NY
noam.shalev@cornell.edu

Nimrod Partush
Forah Inc.
Tel-Aviv, Israel
nimrod@partush.email

ABSTRACT

Finding similar procedures in stripped binaries has various use cases in the domains of cyber security and intellectual property. Previous works have attended this problem and came up with approaches that either trade throughput for accuracy or address a more relaxed problem.

In this paper, we present a cross-compiler-and-architecture approach for detecting similarity between binary procedures, which achieves both high accuracy and peerless throughput. For this purpose, we employ machine learning alongside similarity by composition: we decompose the code into smaller comparable fragments, transform these fragments to vectors, and build machine learning-based predictors for detecting similarity between vectors that originate from similar procedures.

We implement our approach in a tool called *Zeek* and evaluate it by searching similarities in open source projects that we crawl from the world-wide-web. Our results show that we perform 250X faster than state-of-the-art tools without harming accuracy.

ACM Reference Format:
Noam Shalev and Nimrod Partush. 2018. Binary Similarity Detection Using Machine Learning. In *The 13th Workshop on Programming Languages and Analysis for Security (PLAS'18), October 19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3264820.3264821

1 INTRODUCTION

Similarity detection between binary code samples has various use cases in the fields of cyber security and intellectual property. Open source code vulnerability discovery rate is on the rise [11] with the number of reported vulnerabilities having more than doubled during 2017. Whenever a new vulnerability is discovered, it may apply to multiple versions of the package, which have already been compiled and distributed to end-users. These end-users, e.g., individuals and companies which purchased a network router, may only access the code their router is running in *binary form*. They are left exposed, not knowing whether the newly discovered vulnerability applies to them, waiting for a patch which may or may not be released.

Furthermore, cloud providers would like to protect their clients by scanning their virtual machines for vulnerabilities, however refrain from doing so for privacy reasons. Having the ability to scan binary codes allows for the direct search of memory content for vulnerabilities without infringing upon privacy or requiring any knowledge of the VM operations. Moreover, such an approach focuses the scope of the results on software that is assured to be loaded, and allows one to ignore irrelevant executables in the system, that do not run. Finding similarity in binary codes also has implications in the software IP domain, allowing the scanning of released binaries for IP theft.

The main challenge in identifying binary similarities is that the same source code can be compiled using different compilers, different optimization levels or targeting different architectures, thus producing syntactically different binary codes. Though this challenge can be solved with near-perfect accuracy using SMT solvers [4], such approaches are infeasible due to their low throughput, making them irrelevant in cases with big code corpora. On the other hand, faster approaches achieve lower accuracy and generate false positives, which waste human resources, as the results are later manually reviewed for assurance.

In this paper, we propose an approach for binary similarity detection which is highly accurate yet faster than any previous binary similarity search technique, thus allows for scanning real-world workloads in practical time. It is based on the similarity by composition principle [3] alongside machine learning. We introduce the *proc2vec* method for representing procedures (or code sections) as vectors. In proc2vec we decompose each procedure to smaller segments, translate each segment to a canonical form and transform its textual representation to a number, thus finding an embedding in the vector space for each procedure. Next, we design a neural network classifier that detects similarity between

PLAS'18, October 19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5993-1/18/10…$15.00
https://doi.org/10.1145/3264820.3264821

vectors that originate from semantically equivalent procedures. In order to train our classifier, we build a database with dozens of millions of examples, generated by compiling various open-source renowned projects. We compile each project using different compilers, with different compiler versions, optimization levels and for different architectures.

Finally, we evaluate our approach by predicting similarities in open-source projects that are not included in the training set. We show that our proposed classifier achieves high accuracy and executes more than 250X faster than current state-of-the-art tools.

2 RELATED WORK

Similarity Search. Previous work in the binary code-search domain mostly employed syntactic techniques, and were not geared to handle the challenges of a cross-compiler search [10, 18]. Others require dynamic analysis in addition to static analysis [6, 14] or do not scale [4, 14] due to computationally heavy techniques.

David et al. [4, 5] presented an approach for reasoning about similarity of binaries based on segmenting code sections into strands. We build upon this approach, however in contrast to [4] which uses a heavyweight SMT solver, we employ machine learning techniques for predicting similarity. Moreover, unlike GitZ [5], we forgo the need for lengthy translations between IR representations and costly statistical reasoning; thus we achieve better throughput of about two orders of magnitude.

ML and Similarity Search. Previous work [2, 7, 8, 12, 17] proposed using ML for code clone detection and vulnerability search; however, they all operate on the source code level and not on the binaries. Specifically, Li et al. [8] and Peng et al. [12] proposed techniques for building program vector representations, so these can be fed into deep learning models. In this paper, we wish to do the same for binaries.

3 PROBLEM STATEMENT

Problem definition. Given a query procedure q and a large collection T of (target) procedures, in a stripped binary form (without debugging information), our goal is to quantitatively define the similarity of each procedure $t \in T$ to the query q. We require a method that can operate without information about the source code and/or tool-chain used in the creation of the binaries. The main challenge is to devise a prediction method that is precise enough to avoid false positives, flexible enough to allow finding the code in any compilation configuration, and fast enough so similarity search can be performed on huge corpora or memory regions in a timely manner.

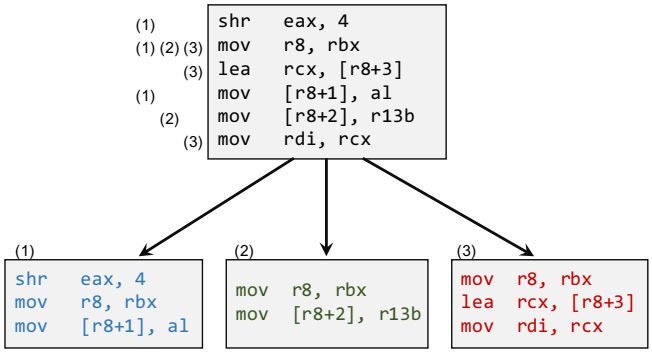

Figure 1: Decomposition to strands.

Metrics. For evaluation, we conduct all vs. all classification experiments. That is, we compile each project P in our test set using multiple compilation configurations, thus generate the binary procedures $\{p_1, p_2, ..., p_n\}$, where n equals to the number of procedures in P times the number of compilation configurations. Next, we perform n^2 predictions; each prediction corresponds to the probability that procedure p_i is similar to procedure p_j. Overall, for each project we output n lists; each list corresponds to a procedure and contains n probabilities. The accuracy of each of these lists is measured by Concentrated ROC (*CROC*) [16], a standard metric for evaluation of classifiers on early retrieval problems.

4 ZEEK OVERVIEW

In this section we provide an overview of our algorithm and system design. We begin in Section 4.1 by reviewing the concept of a code strand and giving intuition about our approach. Next, in Section 4.2, we describe proc2vec, our algorithm for representing code sections as vectors. Finally, in Sections 4.3 and 4.4 we cover our data generation and ML classifier model design, respectively.

4.1 Strands as Features

4.1.1 Strands. We adopt the notion of *strands* introduced in [4] as a building block in our algorithm. A strand is the set of instructions from a code block that are required to compute the value of a certain variable. Figure 1 shows an example of a decomposition of a code block into its composing strands. Note that a single instruction can be associated with multiple strands within a code section. For example, the second instruction of the code block in Figure 1 belongs to all three strands that compose the code section. Note that two syntactically different strands can be equivalent, and that a strand is not necessarily syntactically contiguous.

4.1.2 Intuition. The intuition behind our proc2vec algorithm is based on the similarity by composition principle [3], according to which two signals are similar if it is easy to

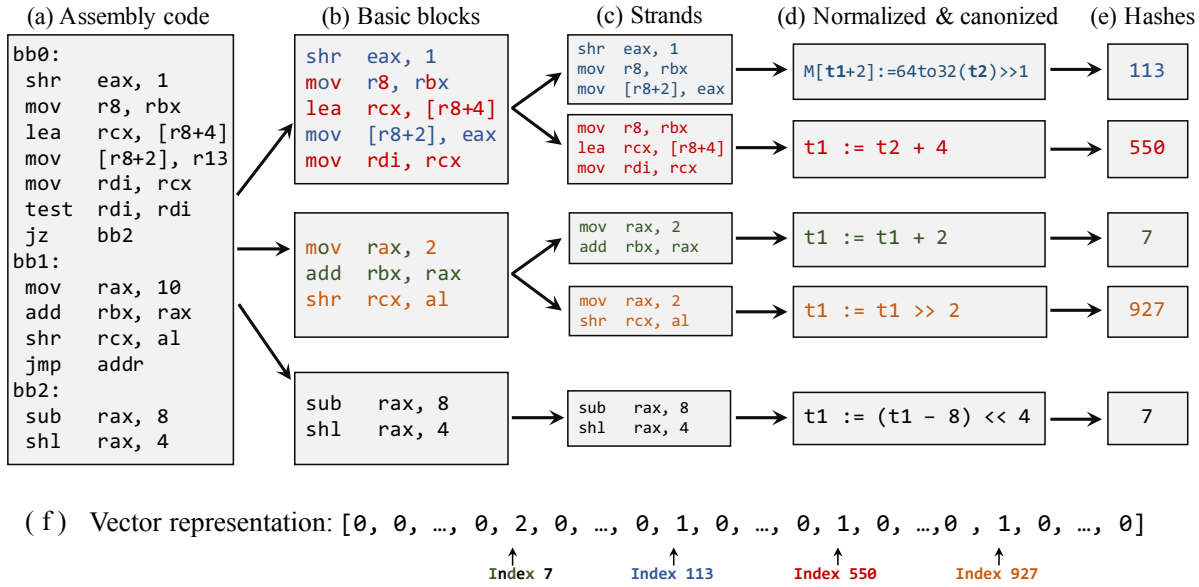

Figure 2: Procedure to vector transformation stages.

compose one signal from large contiguous chunks of the second signal. The similarity by composition principle has been proven to be valuable for similarity detection in image processing [3] and program analysis [4].

Therefore, in this work we wish to use the strands that compose a code section as its feature set. We transform strands to numbers, and assemble those numbers to form a vector that represents the corresponding code. In order to learn the prevalence and co-occurrence of virtually contiguous chunks of code (strands), we train a machine learning model. Thus, code sections that contain rare strands or rare combinations of strands, are likely to be more unique and get a higher similarity score. Likewise, common strand combinations are likely to have the opposite effect. For example, compiler-induced strands (e.g. stack handling operations) are very prevalent and indeed irrelevant for similarity matters.

Previous work [5, 13] has put effort in detecting compiler-induced binary code and artifacts. In our work, we achieve this by employing ML as detailed in this section.

4.2 prov2vec

In this section we describe proc2vec, our algorithm for transforming procedures or code sections to vectors. As described in Figure 2, given the assembly code of a procedure (Figure 2a), the technique for transforming it to a vector is comprised of five steps:

(1) First, we split the procedure to basic blocks (Figure 2b). A basic block is determined according to the `jmp` instruction placements in the procedure binary code.
(2) As depicted in Figure 2c, we further decompose each basic block into *strands*. In the figure, different strands have different colors, while instructions that belong to few strands are marked with all the associated colors.
(3) Next, we bring syntactically different strands with same semantic meaning to the same textual representation. For that purpose, we use techniques introduced in previous work [5] and optimize and canonize each of the strands. As shown in Figure 2d, this step changes the strands' representation to a canonical one, so subsequent additions are grouped, multiplications with power-of-two are replaced by shifts, arithmetic operations are reordered to a consistent representation, etc.
(4) We apply b-bit MD5 hashing on the textual representation of the strands; thus translating each strand to an integer (Figure 2e) in the range $\{0, 2^b - 1\}$. Note that if b is small then collisions are likely to happen, thus different strands can be mapped to the same hash value.
(5) Finally, as depicted in Figure 2f, we use the resulting integer set as indexes and build a vector of length 2^b whereby each element equals to the number of times that the index of the element appears in the set. Hence, the vector elements can be larger than one, and the sum of the vector elements is equal to number of strands that the corresponding procedure contains.

For implementing the above algorithm, we use the PyVEX [15] open-source library. We employ its binary lifter in order to lift the assembly code into `VEX-IR` and slice it to strands (step 2). We further exploit the VEX optimizer on each of the strands for bringing them to a normalized representation (step 3).

Training Set	
Application Name	**Version**
binutils	2.3
OpenSSL	1.0.1
bash	4.3
httpd	2.4.33
ntp	4.2.8
cURL	7.60.0
Snort	2.9.11.1
Git	2.9.5
util-linux	2.32
Apache Mesos	1.5.0
QEMU	2.12.0

Test Set	
Application Name	**Version**
tar	1.30
FFmpeg	2.7.1
Wireshark	1.10.10
coreutils	8.29
bzip2	1.0.6
wget	1.15

(a) Open source projects used for data generation.

Compiler	Versions	Architecture	Optimization Levels
gcc	4.{7,8,9}	x86_64	-O{s,0,1,2,3}
icc	14, 15	x86_64	-O{s,0,1,2,3}
Clang	3.{5,6,7,8}	x86_64	-O{s,0,1,2,3}
gcc	4.8	AArch64	-O{s,0,1,2,3}
Clang	4.0	AArch64	-O{s,0,1,2,3}

(b) Compilers and versions used for data generation.

Figure 3: Data set properties.

4.3 Data Generation

For generating the data, we take various open-source projects (Table 3a) that we find in the wild and compile them for different architectures using different compilers types, versions and optimization levels (Table 3b). Next, we apply our proc2vec algorithm on each of the resulting binary procedures, thus creating a list of vectors, each of which represents a procedure in some specific compilation setting. By stacking all these vectors we build the matrix M.

Using our prior knowledge about the origin of each vector, we build a *match-list*. The match-list contains pair tuples that represent indexes of matching rows in M. That is, if the pair (i, j) appears in the match list, then the corresponding vectors M_i (i'th row of M) and M_j are originated from the source code of the same procedure. Furthermore, in order to teach our classifier that every procedure is similar to itself, and that the input order of a pair has no importance we augment each matching pair (x, y) to a foursome consisting of the pairs (x, y), (x, x), (y, y), (y, x). We label these pairs by 1 (match) and use them to generate our positively labeled dataset.

For generating negative examples, we generate random pairs of vectors that originate from non-equivalent procedures. We make sure that each of these pairs does not appear in the match-list, label the pairs by 0 (non-match) and use them as our negatively labeled data-set.

A similarity between code sections is relatively a rare event and we address that in two ways: (1) we use the CROC metric, which is designed for evaluating unbalanced classification problems, for estimating the classifier's performance. (2) We generate a strongly unbalances dataset and examine different ratios between the number of the negatively and positively labeled examples. After exploring various ratios, we set the ratio to 6.

Finally, we build our full database by shuffling the union of the positively and negatively labeled data-sets. Overall, we get a database with about twenty million examples.

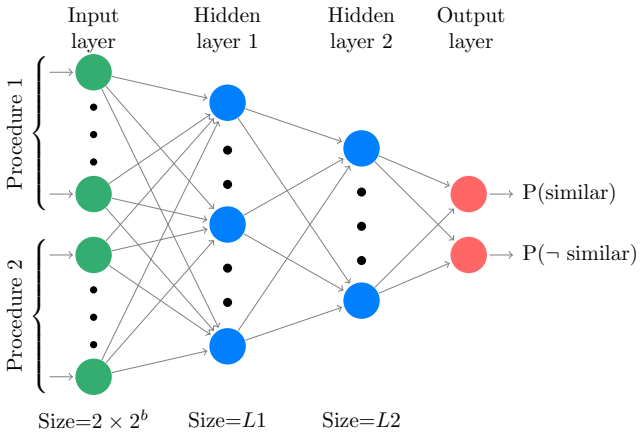

Figure 4: Our neural network skeleton.

4.4 Designing an NN Classifier

For estimating the similarity score of a pair of vectors, we build a deep learning classifier using TensorFlow [1]. Our model is comprised of an input layer, two fully-connected hidden layers, and a softmax output layer, as shown in Figure 4. The input layer is comprised of 2×2^b neurons, corresponding to the two vectors that represent the two input procedures. The first and the second hidden layers are of sizes $L1$ and $L2$, respectively. Finally, the output is a 2-neuron softmax function, representing the probability of similarity between the code sections that generated the input vectors.

Except for the output layer, all the activation functions in the network are tanh. We train the model using cross-entropy cost function, dropout regularization of 0.1, batch size of 32 and 3 passes over the data (epochs).

5 EVALUATION

We evaluate our approach and compare our results with GitZ [5], the fastest state-of-the-art binary similarity search tool [9]. We start in Section 5.1 by exploring various hash size values and fit for each value a corresponding classifier. Next, in Sections 5.2 and 5.3 we present the accuracy and throughput results of our cross platform similarity predictor, respectively. All experiments were performed on a machine with two Intel Xeon E5-2699 processors, 368 GB of RAM, running Ubuntu 16.04.1 LTS.

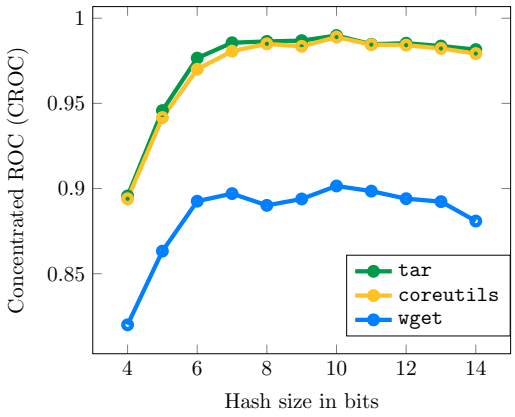

Figure 5: Hash size exploration results.

5.1 Hash Size

MD5 processes text into a fixed-length output of 128 bit; however, a 2^{128}-length vector is much longer than we need. Therefore we fold the hash values (using modulo) thus de-facto reducing the size of the hash. The hash size determines a trade-off: high values (longer input vectors) mean longer representation for each procedure, which may cost in over-fitting and increased training time. On the other hand, low values (shorter input vectors) might create collisions in the procedure representation and omit imperative information from the classifier (underfit).

For finding the optimal hash size, we explore various values and fit a classifier that performs best for the each of the examined values on a toy dataset. The toy dataset consists of code drawn from four applications (OpenSSL, git, util-linux and Apache Mesos) which we compile using all our `x86_64` compilers (see Figure 3b) with the `-O2` optimization level. The parameters that we fit are $L1$, $L2$, and dropout rate.

We test each classifier on three test projects (coreutils, wget, tar) by conducting an all vs. all experiment and present the results in Figure 5. Note that the hash size axis is in units of bits; namely, a value of h implies a vector representation of length 2^h. As can be inferred from the figure, for all of the projects in our test set, using 10 bits for performing the MD5 hashing yields the best results; namely, 2^{10} is the optimal length for a vector representation of a procedure under such a neural network setting.

5.2 Cross Platform Similarity Predictor

Given the results of the previous section, we choose a 10-bit MD5 hash size, configure our neural network with the parameters that optimized the performance of 2^{10} procedure representation ($L1 = 512$ and $L2 = 128$), use data gathered from all of the projects listed in Table 3a, and train a cross-{compiler, version, optimization} binary similarity predictor.

For evaluation, we conduct an all vs. all experiment for each of the projects in out test set and present the results

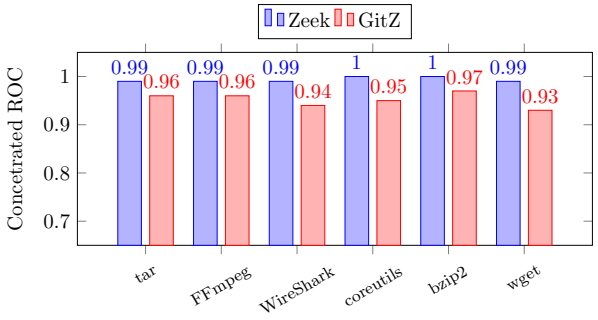

Figure 6: CROC score of the all vs. all experiments, for each of the projects in the test set.

in Figure 6. We repeat these experiments using GitZ and present the results alongside the corresponding Zeek results. As can be seen, Zeek achieves higher average CROC score in all of the tested projects. Note that, compared to the previous section, the results for `wget` are improved. We attribute that to the larger and more diverse training set used in the current fully-trained model.

5.3 Throughput and Latency

Our neural network can output approximately 7000 predictions per second on a single core (excluding the training time, which is performed only once). Furthermore, since the procedure vectors are independent and the model is small enough to fit in memory, the prediction generation can scale with the number of available processors. GitZ, on the other hand, is able to output about 25 predictions per second, hence Zeek introduces a speedup of more than 250X over the state-of-the-art fast similarity detection tools. Note that approaches that achieve near-perfect accuracy, like [4], need about 5 seconds for a single comparison.

In terms of latency, Zeek also has a shorter preprocessing pipeline than GitZ, as it does not require lengthy translation from `VEX-IR` to `LLVM-IR`, thus cutting the required time to output the first prediction by about 15%.

6 CONCLUSIONS

We present an approach for detecting binary similarity using the similarity by composition principle alongside machine learning. We devise proc2vec, an algorithm for representing code sections by vectors, without applying human-crafted feature extraction processes. We show that our approach achieves high accuracy and throughput, thus it is practical for use in real world scenarios.

7 ACKNOWLEDGMENTS

This research was funded in part by the Technion Hiroshi Fujiwara Cyber Security Research Center and the Israel Cyber Bureau.

REFERENCES

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283. https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[2] Upul Bandara and Gamini Wijayrathna. 2012. Detection of Source Code Plagiarism Using Machine Learning Approach. In *International Journal of Computer Theory and Engineering (IJCTE 2012 Volume 4, Number 5)*. https://doi.org/10.7763/IJCTE

[3] Oren Boiman and Michal Irani. 2006. Similarity by Composition. In *Advances in Neural Information Processing Systems 19, Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 4-7, 2006*. 177–184.

[4] Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical Similarity of Binaries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2016)*. ACM, New York, NY, USA, 266–280. https://doi.org/10.1145/2908080.2908126

[5] Yaniv David, Nimrod Partush, and Eran Yahav. 2017. Similarity of Binaries Through Re-optimization. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 79–94. https://doi.org/10.1145/3062341.3062387

[6] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 303–317. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/egele

[7] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder. 2017. CCLearner: A Deep Learning-Based Clone Detection Approach. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 249–260. https://doi.org/10.1109/ICSME.2017.46

[8] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. *CoRR* abs/1801.01681 (2018).

[9] Asuka Nakajime. 2017. Similarity Calculation Method for Binary Executables. (2017).

[10] B. H. Ng and A. Prakash. 2013. Expose: Discovering Potential Binary Code Re-use. In *2013 IEEE 37th Annual Computer Software and Applications Conference*. 492–501. https://doi.org/10.1109/COMPSAC.2013.83

[11] NIST. 2018. *National Vulnerability Database*. Available at https://nvd.nist.gov/.

[12] Hao Peng, Lili Mou, Ge Li, Yuxuan Liu, Lu Zhang, and Zhi Jin. 2015. Building Program Vector Representations for Deep Learning. In *Proceedings of the 8th International Conference on Knowledge Science, Engineering and Management - Volume 9403 (KSEM 2015)*. Springer-Verlag, Berlin, Heidelberg, 547–553. https://doi.org/10.1007/978-3-319-25159-2_49

[13] Nathan E. Rosenblum, Barton P. Miller, and Xiaojin Zhu. 2010. Extracting Compiler Provenance from Program Binaries. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '10)*. ACM, New York, NY, USA, 21–28. https://doi.org/10.1145/1806672.1806678

[14] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2013. Data-driven Equivalence Checking. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. ACM, New York, NY, USA, 391–406. https://doi.org/10.1145/2509136.2509509

[15] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. 138–157. https://doi.org/10.1109/SP.2016.17

[16] S. Joshua Swamidass, Chloé-Agathe Azencott, Kenny Daily, and Pierre Baldi. 2010. A CROC Stronger Than ROC: Measuring, Visualizing and Optimizing Early Retrieval. *Bioinformatics* 26, 10 (2010), 1348–1356. https://doi.org/10.1093/bioinformatics/btq140

[17] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep Learning Code Fragments for Code Clone Detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 87–98. https://doi.org/10.1145/2970276.2970326

[18] zynamics. 2011. *BinDiff*. Available at http://www.zynamics.com/.