Binary Code Similarity Detection

Zian Liu

Department of Computer Science and Software Engineering Swinburne University of Technology Data61, CSIRO Melbourne, Australia 102516622@student.swin.edu.au

Abstract-Binary code similarity detection is to detect the similarity of code at binary (assembly) level without source code. Existing works have their limitations when dealing with mutated binary code generated by different compiling options. In this paper, we propose a novel approach to addressing this problem. By inspecting the binary code, we found that generally, within a function, some instructions aim to calculate (prepare) values for other instructions. The latter instructions are defined by us as key instructions. Currently, we define four categories of key instructions: calling subfunctions, comparing instruction, returning instruction, and memory-store instruction. Thus if we symbolically execute similar binary codes, symbolic values at these key instructions are expected to be similar. As such, we implement a prototype tool, which has three steps. First, it symbolically executes binary code; Second, it extracts symbolic values at defined key instructions into a graph; Last, it compares the symbolic graph similarity. In our implementation, we also address some problems, including path explosion and loop handling.

Index Terms-Binary code, code analysis, symbolic execution

I. INTRODUCTION

My PhD goal aims at analyzing binary code to improve binary code security. This includes similarity detection, code diffing, bug finding, etc., at binary level. I plan to achieve this goal by implementing novel approaches in a binary code similarity detecting tool. The current process is that the tool's framework has finished. I am fine-tuning this tool to conducting on a large scale dataset.

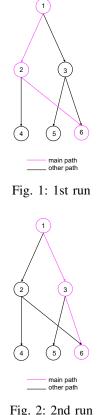
With the rapid development of the software industry, binary code similarity detection is playing an increasingly critical role. Software producers release their products as binary code mainly to protect their source code. Binary code similarity detection can be used in many fields such as bug search, malware detection, malware clustering, malware linage, patch generation and analysis, porting information, software theft detection [1], etc. However, binary code similarity detection is challenging, which is mainly due to, 1) The same source code can be compiled on different architectures, resulting in different binary codes using different instruction sets; 2) During compiling process, different compilers and compiling options may produce significantly different binary code for the same source code; 3) Producers can use many code protecting tools to obfuscate the code. There are some existing solutions to this problem. These works can be divided into three categories with different standards. In Section II, we mainly introduce these works according to their comparison type: syntactic, semantic, and structural similarity. A typical syntactical approach represents the code by statisticizing the occurrence of specific strings in the code. Alternatively, one can use machine learning based method to learn the vectorized representation of one or more sentences in the code. A semantic approach evaluates whether two codes have similar functionality or impact. Structural approach compares graph features such as control flow graph and call graph. However, these solutions still have some limitations. For example, syntactic similarity focus on the 'appearance' of binary code thus sometimes failing to explore semantic similarity. Semantic similarity, however, is not scalable and may not cover all of the code or situations. Moreover, Ren et al. pointed out that existing binary code similarity detection overlooked the impact of compiling options [2]. They claimed that the testing set used in current work lacked enough compiling options mutation. They tested several state-of-the-art binary similarity detection tools with their crafted dataset. The detecting result all turned out to be less accurate. Structural similarity is vulnerable for cross-architecture comparison since they assume that basic blocks remain their feature and relationship with other basic blocks under all circumstances. However, by utilizing Ren el al's method, one can tune the compiling configuration to have less structural similarity. Therefore, our research question is how can we detect binary code similarity even when these compiling options are deployed.

By analyzing the binary code, we observed that some instructions calculate value for some other instructions. The latter instructions are more important for deciding the similarity while the former instructions should be given less consideration. This is because the symbolic value of the former can propagate to latter instructions. Only inspecting the symbolic value of these latter instructions can reserve the main symbolic meaning of the function. We thus define these important instructions as Key Instructions. In our tool, they are translated to Intermediate Representation (IR) and connected based on control flow to form a Key IR graph. We implemented our idea in a prototype tool. This tool is able to lift different architectures' binary code into Key IR graphs and compare their similarity. This tool consists of three phases: 1) symbolic execution, 2) Key IR graph construction, and 3) Key IR graph similarity detection. In summary, our contributions are: 1) We proposed a Key IR graph to abstract the semantic of binary code. 2) We implemented a prototype tool, including the symbolic execution engine, IR lifting engine, and comparison engine. This tool currently supports two architectures, i.e., x86-64 and ARM. 3) We proposed a novel method to compare the similarity of Key IR graphs. Currently we have finished the implementation process and on our way to evaluation.

II. BACKGROUND

As [1] mentioned in their survey, existing works on binary code similarity can be classified according to different standards. In terms of the comparison granularity, these works can be divided into instruction level, basic block level, function level, and whole program level. In the binary similarity comparison, one input is used as a query to compare with the target. Thus according to input-target number, they can be classified into one-to-one, one-to-many, and manyto-many comparisons. According to supported architectures, they can be divided into a single architecture and crossarchitecture. Single architecture only supports one kind of assembly instruction set, while cross-architecture supports more than one. According to analysis type, they can be categorized into static analysis, static analysis, and hybrid analysis. Static analysis analyzes the code without running the code, while dynamic analysis executes the code with some input. The advantage of static analysis is scalability and code coverage. Dynamic analysis is resource consuming but helpful to gain semantic information. According to comparison type, they can be classified into syntax, structural, and semantics similarity comparison. Syntax similarity captures instruction representation similarity, while structural similarity focuses on similarity in terms of graph representation of the binary code. Semantic similarity highlights the similarity of code's impact.

We introduce various techniques used according to different comparison types. For syntax similarity, common strategies include hashing, embedding, and alignment. [3]-[5] all use hashing technique to output various instructions sequences into fixed length of output hashing value. Equivalent output value implies syntax similarity. [6]-[9] generate an embedding from sequences. [10]–[13] automatically learn the embeddings for each instruction and use them to produce the basic-level or function-level embedding. [14]-[16] align two sequences and decide their similarity. For structural similarity, common methods are optimization solution, k-subgraph matching, path similarity, and graph embedding. [3], [17]-[19] transform the problem into finding the mapping between two CFGs with minimum cost (optimization solution). [18], [20]-[22] divide the graph into k subgraphs, so that each one has k connected nodes. Matched number of those subgraphs indicates the extent of the similarity. [16], [23], [24] determine similarity based on paths. [19], [25], [26] extract features from graphs into feature vectors and determine the vector similarity. For semantic similarity, general techniques are instruction classification, input-output pairs, symbolic execution, theorem prover, and semantic hashes. [20], [22], [24] classified instruction based on their semantic in terms of arithmetic, logic or data transfer purpose. [5], [27]–[31] check whether output are the same to the input. [8], [23], [27], [28], [32], [33] use symbolic



- -8. _. _....

formulas to represent the binary code. With the symbolic formula, [32] uses Theorem Prover to check whether two different symbolic formula has similar output. [27], [34], [35] use symbolic hashes as an alternative to the theorem prover. [36], [37] determine the edit distance of the tree/graph of the symbolic formula.

III. METHODOLOGY

The comparison granularity of our tool is function level. Given two functions from two different input binary codes, our tool consists of three phases: 1. symbolic execution, 2. Key IR graph construction, and 3. Key IR graph similarity comparison. We firstly symbolically execute the binary code several times to get their possible symbolic values. Then for the most important instructions for comparison, we lift them and their symbolic values into Key IRs. Those Key IRs are then combined to produce the Key IR graph. Lastly, we compare the similarity of two IR graphs.

A. Symbolic execution

In this section, we randomly select paths and symbolically execute the function to infer the results of each operand in each instruction. As shown in Figure. 1 and Figure. 2, a function may contain many paths and symbolic execution might have the path explosion problem. Thus we symbolically execute the function many times. For each run, we randomly select a main path by using deep-first algorithm to symbolic execute this path. However, if we only execute one main path each run, we

mov mov call	<pre>rcx, [rsp+1D8h+var_1C8] edx, 20h ; ' ' esi, 19h EVP_CIPHER_CTX_ctrl</pre>
mov sub lea mov cmp jnb	<pre>(a) calling subfunction rbp, r13 rbp, r14 rax, ds:0[rbx*4] [rsp+1D8h+var_1D0], rax rbp, rax short loc 1C88</pre>
	(b) comparing instruction
mov	<pre>eax, 0FFFFFFFh</pre>
mov xor	<pre>rsi, [rsp+1D8h+var_40] rsi, fs:28h</pre>
add pop pop pop pop pop retn	rsp, 1A8h rbx rbp r12 r13 r14 r15
mov lea mov	<pre>(c) returning instruction rax, [rsp+1D8h+var_1D8] r13, [rax+r14] [rsp+1D8h+var_170], r13 (d) memory address writing</pre>

Fig. 3: Four types of Key Instructions

still may miss many instructions after many times of run (e.g., we may miss instructions 3, 4, 5 in Figure.1, and instructions 2, 4, 5 in Figure.2). To cover all the instructions in each run, except the main path, we also select other paths to cover all the instructions. We do not propagate to the instructions we already encountered to avoid the path explosion problem in each run.

Since we randomly select paths in each run, some instructions demonstrate different symbolic values. In fact, not each instruction can have different states. Only the instructions at the joint of some paths can hold different symbolic values (e.g., instruction 6 in Figure.1 and 2). Other instructions can only hold one possible symbolic value (e.g., all other instructions in Figure.1 and 2). The percentage of these possible symbolic values revealed is dependent on the times of run. Generally, the more time the run, the more chances we can reveal all the possible values.

B. Key IR graph construction

As we observed in the binary code, many instructions are making preparations for some other instructions, such as calling subfunction, comparing instructions, as shown in Figure. 3. In Figure.3a, parameters are firstly loaded into registers *rcx* and *edx* before calling the EVP_CIPHER_CTX_ctrl subfunction. In Figure.3b, values in register rbp are loaded from register *r13* and subtracted with value in register *r14*. Value in register *rax* is loaded from some memory address before comparing with *rbp*. In Figure.3c, value in register eax

is set to 0FFFFFFFF before stack balancing instructions (i.e., pop instructions) and return instruction. In Figure.3d, value in somewhere of the memory address is loaded to register rax and added to the value in register r14. Then this value is written to some other memory location [rsp+1D8h+var_170]. As we can see in all these examples, some instructions are preparing values for some later instructions. We define these preparing instructions as non-Key Instructions while the other instructions as Key Instructions. From our observation, during execution, the non-Key Instructions should propagate their values into the Key Instructions. And the Key Instructions better describe the behavior of the binary code. Similar binary code should contain similar Key Instructions with similar values (e.g., calling the same subfunction with similar parameters or comparing similar values to similar values). We define the Key Instructions to contain two types: control-flow impacting and control-flow irrelevant. Specifically, the control-flow impacting include three categories: 1. calling subfunction, 2. comparing instruction, and 3. returning instruction. We select them because Calling subfunctions leads the execution flow into other parts of the binary code. The result of comparing instructions decides which branches to take next. Returning instructions leads execution flow back to the caller function. The control-flow irrelevant instructions refer to the memory writing instructions since it does not affect the control flow.

With the result of the first module, symbolic execution, we can translate Key Instructions into Key IRs with symbolic values. Each instruction in the binary code corresponds to one Key IR node. Then we connect the Key IRs based on their control flow to form the Key IR graph. It is important to note that some nodes only contain one possible symbolic value among those Key IR nodes, while the other nodes might contain multiple symbolic values because they are the joint of multiple paths.

C. Key IR graph comparison

Because of the impact of cross-architecture, different compiling options, and other factors, the Key IR graph might contain reordered nodes, inserted new nodes, and duplicated nodes. Sometimes Key IR graph might also lose some nodes and have other mutations because of the compilation options described in [2]. We propose a fuzzy method to match two Key IR graphs. To compare the similarity of two Key IR graphs, we check how many similar nodes in two graphs. To compare the similarity of a pair of node, we divide this process into two phases: 1. single node textual similarity, and 2. context similarity. In the first phase, we pick the most similar node pair from two Key IR graphs as potentially similar pairs if these two nodes' symbolic value has high textual similarity. The symbolic values have been simplified beforehand by msynth ¹. Next, we compare the similarity of this node pairs' context. The context here refers to the neighbor nodes of a node within a given boundary. Again, we compare these neighbors' simplified symbolic value textual similarity. An example of

¹https://github.com/mrphrazer/msynth

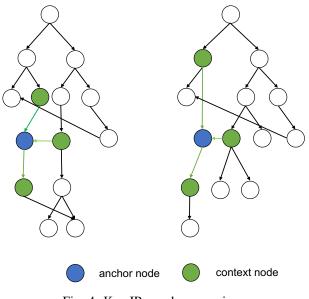


Fig. 4: Key IR graph comparison

this graph comparison is shown in Figure.4. Suppose we find the potentially similar node pair (blue circle) with high similarity, and the context boundary is 1. Thus we examine the neighbor nodes within this bound (green circle) and check their similarity. When detecting the context similarity, the existence of similar nodes within the context indicates the similarity. The more similar nodes are, the more similar these binaries will be. We ignore the relations between these nodes because the complexity of compilation options can significantly mutate the relations.

IV. IMPLEMENTATION

Our implementation is an IDA Pro plugin written in C++ and python. Our tool is based on IDA pro's control flow graph and its API. We have finished our implementation and on our way for evaluation.

A. Symbolic execution

1) Symbolic values: At the beginning of each function, we assign VAR 0 to VAR n to parameters 1 to N of this function. With the execution of each instruction, those initial tags will be represented in various expressions and are propagated to different instructions as their operands' values. The similarity of the expressions reveals similar semantics. Thus we transform binary code similarity detection into expression similarity detection.

2) Handling loops: During the execution, we might encounter loops. Since it is often difficult to decide the time of execution of a loop, it is challenging to propagate the values of instructions in the loop to some Key Instructions. Also, it is non-trivial to analyze which instructions' operands are invariant during loops and updated in each loop. To address these problems, we symbolically execute the instructions in a loop twice. The invariant operands keep their constant symbolic value, while the updated operands add an 'ITER()' notation outside their value to highlight that their values are changed in the loop.

B. Key IR graph construction

We recover both control-flow impacting and control-flow irrelevant instructions into Key IRs. We identify those instructions if they match some patterns. The identification is implemented as a rule in the tool. For control-flow impacting instructions, we recover three types, including 1. calling subfunction instructions, 2. comparing instruction, and 3. returning instructions. For Type 1, we find the corresponding parameters for the subfunction. The patterns for matching Type 2 instruction are more complicated than other types since some compiler options can translate comparison instructions from source code to equivalent codes in an implicit way as in [2]. For Type 3, we find the last modification of register rax in x86-64 or register R0 in ARM before the function returns. For control-flow irrelevant instruction, i.e., memory writing instruction, we identify them if the mnemonic is 'mov' in x86-64 or 'STR' in ARM, and the destination operand is a memory address.

Each Key Instruction is then transformed to a node in the Key IR graph. It is important to note that each Key Instruction might have several possible symbolic values. Thus we reserve all of them for each Key Instruction. After we recovered all the nodes, we connect them based on the control flow provided by IDA pro. The result Key IR graph has the same control flow as IDA pro's control flow. The difference is that all the non-Key Instructions are removed.

C. Key IR graph comparison

To compare the similarity of two symbolic formulas, we firstly simplify them using msynth [?]. Msynth is originally used as a binary code deobfuscation tool. It aims to simplify very complex binary code formulas into simple formulas. We use it here to facilitate our comparison. Moreover, various compilation options can be mitigated since the resulting formula will be more similar, even the original formula may be different.

V. EVALUATION

We have collected many open-source benchmarks used widely in existing works, i.e., OpenSSL, Coreutils, SPEC CPU2006, and SPEC CPU 2017. We will answer two research questions in our experiment: 1. Can we effectively detect cross-architecture binary codes? 2. Can we mitigate the influence of compiling options? To answer the first question, we aim to compile our benchmark dataset on ARM and x64 architectures with different compilers such as GCC and Clang. Then we will prepare random pairs of similar and dissimilar function pairs with labels indicating their similarity. To answer the second question, we will compile benchmark dataset by using the compilation options described in [2] and prepare similar and dissimilar function pairs. In both experiments, the percentage of correctly detected similar and dissimilar function pairs will be regarded as the accuracy, which is to indicate the performance of our work.

REFERENCES

- I. U. Haq and J. Caballero, "A survey of binary code similarity," ACM Comput. Surv., vol. 54, Apr. 2021.
- [2] X. Ren, M. Ho, J. Ming, Y. Lei, and L. Li, "Unleashing the hidden power of compiler optimization on binary code difference: An empirical study," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, (New York, NY, USA), p. 142–157, Association for Computing Machinery, 2021.
- [3] X. Hu, T.-c. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proceedings of the 16th ACM Conference* on Computer and Communications Security, CCS '09, (New York, NY, USA), p. 611–620, Association for Computing Machinery, 2009.
- [4] M. R. Farhadi, B. C. Fung, P. Charland, and M. Debbabi, "Binclone: Detecting code clones in malware," in 2014 Eighth International Conference on Software Security and Reliability (SERE), pp. 78–87, 2014.
- [5] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song, "Spain: Security patch analysis for binaries towards understanding the pain and pills," in 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp. 462–472, 2017.
- [6] I. Santos, F. Brezo, J. Nieves, Y. Penya, B. Sanz, C. Laorden, and P. G. Bringas, "Idea: Opcode-sequence-based malware detection," in *ESSoS*, 2010.
- [7] B. Kang, T. Kim, H. Kwon, Y. Choi, and E. G. Im, "Malware classification method via binary content comparison," in *Proceedings of the 2012 ACM Research in Applied Computation Symposium*, RACS '12, (New York, NY, USA), p. 316–321, Association for Computing Machinery, 2012.
- [8] B. H. Ng and A. Prakash, "Expose: Discovering potential binary code re-use," in 2013 IEEE 37th Annual Computer Software and Applications Conference, pp. 492–501, 2013.
- [9] S. H. H. Ding, B. Fung, and P. Charland, "Kam1n0: Mapreduce-based assembly clone search for reverse engineering," *Proceedings of the 22nd* ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2016.
- [10] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou, "αdiff: Cross-version binary code similarity detection with dnn," in *Proceedings* of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, (New York, NY, USA), p. 667–678, Association for Computing Machinery, 2018.
- [11] F. Zuo, X. Li, Z. Zhang, P. Young, L. Luo, and Q. Zeng, "Neural machine translation inspired binary code similarity comparison beyond function pairs," *ArXiv*, vol. abs/1808.04706, 2019.
- [12] S. H. H. Ding, B. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," 2019 IEEE Symposium on Security and Privacy (SP), pp. 472–489, 2019.
- [13] L. Massarelli, G. A. D. Luna, F. Petroni, L. Querzoni, and R. Baldoni, "Safe: Self-attentive function embeddings for binary similarity," *ArXiv*, vol. abs/1811.05296, 2019.
- [14] B. Baker and R. Muth, "Compressing differences of executable code," 2012.
- [15] 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, PLDI '14, (New York, NY, USA), p. 349–360, Association for Computing Machinery, 2014.
- [16] 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*, ASIA CCS '17, (New York, NY, USA), p. 155–166, Association for Computing Machinery, 2017.
- [17] M. Bourquin, A. King, and E. Robbins, "Binslayer: Accurate comparison of binary executables," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, PPREW '13, (New York, NY, USA), Association for Computing Machinery, 2013.
- [18] S. Cesare, Y. Xiang, and W. Zhou, "Control flow-based malware variantdetection," *IEEE Transactions on Dependable and Secure Computing*, vol. 11, no. 4, pp. 307–317, 2014.
- [19] 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*, CCS '16, (New York, NY, USA), p. 480–491, Association for Computing Machinery, 2016.

- [20] M. Lindorfer, A. Di Federico, F. Maggi, P. M. Comparetti, and S. Zanero, "Lines of malicious code: Insights into the malicious software industry," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, (New York, NY, USA), p. 349–358, Association for Computing Machinery, 2012.
- [21] W. M. Khoo, A. Mycroft, and R. Anderson, "Rendezvous: A search engine for binary code," in 2013 10th Working Conference on Mining Software Repositories (MSR), pp. 329–338, 2013.
- [22] S. Alrabaee, P. Shirani, L. Wang, and M. Debbabi, "Fossil: A resilient and efficient system for identifying foss functions in malware binaries," *ACM Trans. Priv. Secur.*, vol. 21, Jan. 2018.
- [23] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, (New York, NY, USA), p. 389–400, Association for Computing Machinery, 2014.
- [24] S. Alrabaee, P. Shirani, L. Wang, and M. Debbabi, "Sigma: A semantic integrated graph matching approach for identifying reused functions in binary code," *Digit. Investig.*, vol. 12, pp. S61–S71, 2015.
- [25] 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*, ASE 2018, (New York, NY, USA), p. 896–899, Association for Computing Machinery, 2018.
- [26] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural networkbased graph embedding for cross-platform binary code similarity detection," *Proceedings of the 2017 ACM SIGSAC Conference on Computer* and Communications Security, 2017.
- [27] W. Jin, S. Chaki, C. Cohen, A. Gurfinkel, J. Havrilla, C. Hines, and P. Narasimhan, "Binary function clustering using semantic hashes," in *Proceedings of the 2012 11th International Conference on Machine Learning and Applications - Volume 01*, ICMLA '12, (USA), p. 386–391, IEEE Computer Society, 2012.
- [28] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Crossarchitecture bug search in binary executables," *it - Information Technol*ogy, vol. 59, pp. 83 – 91, 2017.
- [29] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "Bingo: cross-architecture cross-os binary search," *Proceedings of the* 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016.
- [30] U. Kargén and N. Shahmehri, "Towards robust instruction-level trace alignment of binary code," 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 342–352, 2017.
- [31] S. Wang and D. Wu, "In-memory fuzzing for binary code similarity analysis," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, p. 319–330, IEEE Press, 2017.
- [32] S.-C. Wang, C. Liu, Y. yu Li, and W. Xu, "Semdiff: Finding semtic differences in binary programs based on angr," 2017.
- [33] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," SIGPLAN Not., vol. 51, p. 266–280, June 2016.
- [34] A. Lakhotia, M. D. Preda, and R. Giacobazzi, "Fast location of similar code fragments using semantic 'juice'," in *Proceedings of the 2nd* ACM SIGPLAN Program Protection and Reverse Engineering Workshop, PPREW '13, (New York, NY, USA), Association for Computing Machinery, 2013.
- [35] Y. David, N. Partush, and E. Yahav, "Similarity of binaries through reoptimization," *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017.
- [36] Q. Feng, M. Wang, M. Zhang, R. Zhou, A. Henderson, and H. Yin, "Extracting conditional formulas for cross-platform bug search," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, (New York, NY, USA), p. 346–359, Association for Computing Machinery, 2017.
- [37] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow, "Leveraging semantic signatures for bug search in binary programs," in *Proceedings* of the 30th Annual Computer Security Applications Conference, ACSAC '14, (New York, NY, USA), p. 406–415, Association for Computing Machinery, 2014.