Mathematical Problems of Computer Science 48, 64--73, 2017.

Effective and Accurate Binary Clone Detection

Hayk K. Aslanyan

Ivannikov Institute for System Programming of the RAS
e-mail: hayk@ispras.ru

Abstract

Software developers usually copy and paste a particular piece of code as they prefer to use a pre-written or a partial solution as a basis for solving their problem. However, it can lead to various errors, as well as increase the size of the source and binary code. Finding similar parts of code (clones) in binary code becomes more applicable when the source code is not available. Additionally, a compiler can copy some parts of code during various transformations and create code clones, which do not exist in the source code. Detection of binary code clones is used for malware analysis, finding semantic errors, detecting copyright violation, etc. This article discusses the existing methods of binary code clones detection and introduces a new method for binary clone detection. It consists of three main stages. The first stage is based on Binnavi platform [1] and generates program dependence graphs for each binary function. Graphs are generated based on REIL [2] (Reverse Engineering Intermediate Language) platform-independent language. REIL representation is supported for several architectures (x86, x86-64, ARM, MIPS, PPC), thus ensuring the independence of the tool from the target architecture. The second stage detects clones based on previously created graphs. A polynomial heuristic algorithm is suggested for finding the maximum common subgraph of two program dependence graphs. At the third stage, the obtained clones are visualized for manual analysis.

Keywords: Binary code clone, Program static analysis, Program dependence graph.

1. Introduction

The reuse of code fragments (a continuous sequence of code lines) is often encountered in software development process. There are several approaches for finding code clones, which are based on text [3], lexical [4], syntactic [5]—[7] and semantic [8]—[14] analysis of the program. However, all these methods analyze source code and the task of finding clones in binary code is less studied. These approaches are not applicable for binary code as binary code is platform dependent and registers and direct memory access are used, while in source code only variables are considered. Detection of binary code clones has many practical applications such as finding functionally similar parts of the program, malware, semantic errors and copyright violations.

Binary code clones are divided into three main types. The first type of binary code clones are code fragments that completely match. The second type of binary code clones are code fragments,

which can differ in types, values, names of data and registers. The third type of binary code clones are code fragments that can differ in types, names of data and registers, and may also differ in some instructions (in a particular fragment some instructions may be added or removed).

Examples of assembly code clones (for x86 architecture) are shown in Fig.1. The clone of the first type is exactly matched fragments. The clone of the second type uses ecx register instead of eax. The clone of the third type uses ecx register instead of eax and has one deleted instruction (imul eax, ebp+var_4]).

Code fragment	Type 1 clone	Type 2 clone	Type 3 clone

public main
main proc near
 	
var_4= dword ptr -4
argc= dword ptr 8
argv= dword ptr 0Ch
envp= dword ptr 10h
 	
push ebp
mov ebp, esp
mov [ebp+var_4], 5
mov eax,[ebp+var_4]
imul eax,[ebp+var_4]
leave
retn
main endp
	public main
	main proc near
	
	var_4= dword ptr -4
	argc= dword ptr 8
	argv= dword ptr 0Ch
	envp= dword ptr 10h
	
	push ebp
	mov ebp, esp
	mov [ebp+var_4], 5
	mov eax,[ebp+var_4]
	imul eax,[ebp+var_4]
	leave
	retn
	main endp
		public main
		main proc near
		
		var_4= dword ptr -4
		argc= dword ptr 8
		argv= dword ptr 0Ch
		envp= dword ptr 10h
		
		push ebp
		mov ebp, esp
		mov [ebp+var_4], 10
		mov ecx, [ebp+var_4]
		imul ecx, [ebp+var_4]
		leave
		retn
		main endp
			public main
			main proc near
			
			var_1= dword ptr -4
			argc= dword ptr 8
			argv= dword ptr 0Ch
			envp= dword ptr 10h
			
			push ebp
			mov ebp, esp
			mov [ebp+var_1], 15
			mov ecx, [ebp+var_1]
			
			leave
			retn
			main endp

Fig.1. Examples of code clones (x86 assembler).

2. Binary Code Clone Detection Approaches

Text-based approach. Jang et al. [15] proposed a fingerprinting algorithm called BitShred based on bloom filters to cluster malware samples. BitShred consists of three phases: shredding a file, creating a fingerprint, and comparing fingerprints. In the shredding phase, BitShred divides all executable code sections into fragments. Then for each fragment fingerprint is calculated based on Bloom filters [16]. At the last stage it compares fingerprints by the following ratio:
$J(A,B) = S(BF_A \wedge BF_B) / S(BF_A \vee BF_B)$, where S(BF) is the count of set bits in the BF. Finally, fragments, having a higher similarity score, are clustered together. The algorithm finds clones of only the first type.

Token-based approach. A. Schulman [17] proposed a system that creates a hash for each function in a binary file. Matched hashes that occur in more than one file indicate a clone of the code. Hashes are based on opcodes and location labels of the opcodes.

Karim et al. [18] algorithm also considers hashes on instructions opcodes, but they are calculated using n-grams. The algorithms of the approach allow finding clones of the first and second types.

Metrics-based approach. The system, created by D. Bruschi et al. [19], finds clones in binary files to detect malicious programs. First, the binary file is disassembled and normalized, then the dead code is deleted and the code is split into fragments. For each fragment, a metric is constructed based on the control flow graph. At the last stage, clones are detected by comparing the obtained metrics.

Sæbjørnsen et al. [20] after obtaining the assembler from the binary file, create intermediate representations of the assembly code. Then, a binary code is partitioned into overlapping code segments that consist of a block of contiguous assembly from a function. Then they create a normalized instruction sequence, abstracting the information of memory location and registers. Next, it performs clone detection on the normalized sequence. They define two methods for creating clone clusters. The first method is an exact match that uses a hash for each code region, and a clone exists if there are any repeated hash values. The second method is an inexact match, which extracts a set of features from a code region and looks for other code regions with the same feature set. They count the number of occurrences of each feature to create a feature vector for each region. Next, they use locality-sensitive hashing (LSH) (Andoni and Indyk, [21]) on each region and perform a distance calculation for clustering based on features for inexact matching. Based on this work, M. Farhadi et al. [22] created a system for detecting clones of malicious code in programs.

Algorithms of the approach detect all three types of clones.

Structural-based approach. T. Dullien et al. [23] proposed a system for comparing binary files based on structural analysis, to search for malicious code. The algorithm consists of two stages: the generation of several hashes of malicious code and the recognition of similarity between different sections of code based on the control flow graph.

Y. David and E.Yahav [24] detect similarity among functions based on decomposition of functions into tracelets, which are continuous, partial traces of execution and are obtained from control flow graph. To measure similarity between two tracelets, they define a set of simple rewrite rules and measure how many rewrites are required to reach from one tracelet to another. They do this step by encoding the problem as a constraint-solving problem and measure distance using the number of constraints that have to be violated to reach a match.

Algorithms of the approach detect all three types of clones.

Behavior-based approach. In [25] D.E. Krutz and E.Shahab propose a new approach for code clone detecting, which also detects semantic clones. They use concolic analysis, which combines concrete and symbolic values in order to traverse all possible paths.

3. Tool Architecture for Binary Code Clone Detection

The proposed model takes into account the following requirements:

- finding all types of clones;
- independence from the target architecture;
- scalability: the size of the analyzed programs can reach hundreds of MB;
- a large percentage of true positives (> 90%).

It is allowed to specify values of two variables: the minimum number of instructions for clones (MN) and the minimum percentage of similarity (MP) of clones.

The work of the tool is divided into three main stages:

1. The first stage is based on Ida pro disassembler [26] and Binnavi static analysis platform [1]. The Ida Pro disassembler is used as a tool for restoring structures and the control flow of the program. At this stage, machine code is translated to REIL representation, then, PDG (program dependence graph) is generated for each function.
2. At the second stage, the assembler code clones are detected taking into account the parameters of the MN and MP.
3. At the third stage code clones and their corresponding PDGs are visualized.

The main advantage of the proposed tool is that it is based on a semantic approach, which is more correct, than other approaches.

3.1 Generation of PDGs

Binnavi platform is used to generate program dependence graphs for each function. Binnavi provides an interface for generating and using various intermediate representations of the program based on REIL, including the generation of the control flow graph, the generation of the call graph and data dependency graph. As a part of the tool, a new functionality has been added to the Binnavi platform, which allows each function to automatically generate a control flow graph and a data flow graph and merge them into program dependence graph (Fig. 2). The vertices of PDG correspond to REIL instructions, and the edges are data and control dependencies between instructions. Each vertex has ID, which is the opcode of its REIL instruction.

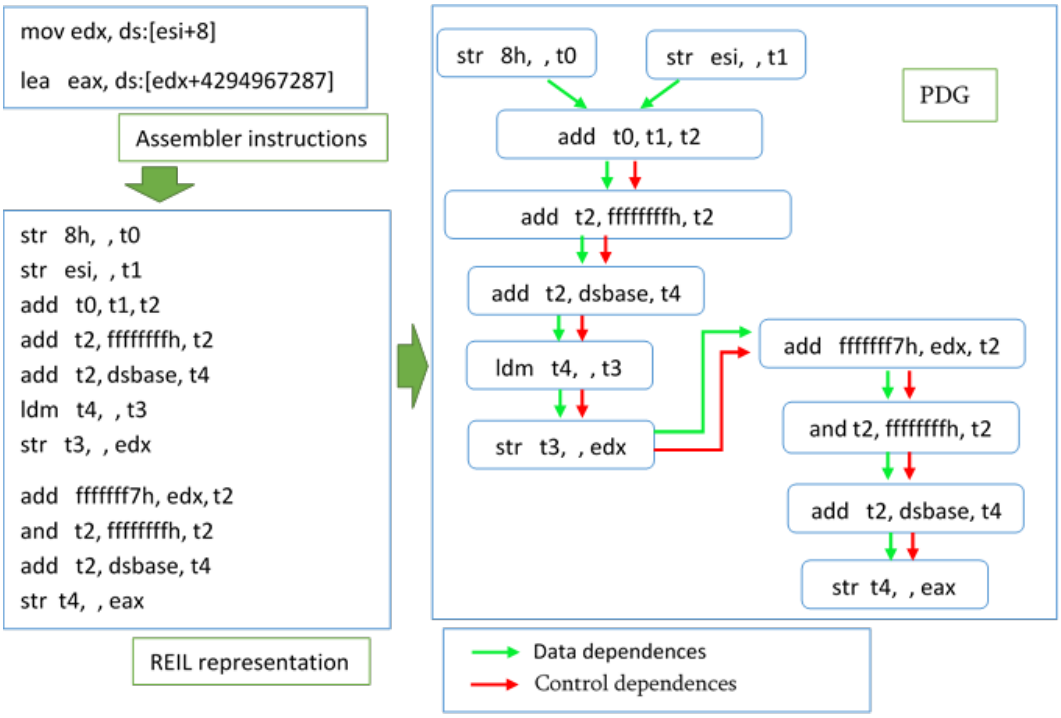

Fig. 2. Example of PDG.

3.2 Heuristic Algorithm for Maximum Common Subgraph Detection of Two PDGs

At this stage a maximum common subgraph is calculated for each pair of PDGs with heuristic algorithm. This algorithm is named Tracebasedslice. It uses several procedures, which are defined below. Let $G(V, E)$ be a PDG and X and Y be any sets of PDG vertices, such that $X \subseteq V$. Two vertices can be matching candidates, if the first vertex's ID, predecessors' and successors' amounts are equal to the second vertex's ID, predecessors' and successors' amount.

Definition 1: ***getPredecessors(X,Y)*** *procedure returns empty set if Y is empty, otherwise returns vertices from X, which are not in Y and are predecessors for Y's vertices.*

Definition 2: ***getSuccessors(X,Y)*** *procedure returns empty set if Y is empty, otherwise returns vertices from X, which are not in Y and are successors for Y's vertices.*

Definition 3: ***sortVertices(X)*** *procedure sorts vertices of X by their IDs, predecessors count, successors count and binary address.*

Definition 4: ***makeCorrespondence (X, Y)*** *procedure returns pairs of vertices from sorted X and Y sets, which are matching candidates. It considers vertices' IDs, predecessors, successors count and based on merging algorithm.*

Definition 5: ***makeOneCorrespondence(X, Y)*** *procedure returns a pair of vertices from sorted X and Y sets, which are matching candidates.*

Definition 6: *For any* $m \in X$ *and* $n \in Y$ ***checkPredecessors(X,Y,m,n)*** *condition is satisfied, if predecessors of m from X and predecessors of n from Y have the same set of IDs.*

Definition 7: *For any* $m \in X$ *and* $n \in Y$ ***checkSuccessors(X,Y,m,n)*** *condition is satisfied, if successors of m from X and successors of n from Y have the same set of IDs.*

Definition 8: ***inducedSubGraph(X, G)*** *procedure returns induces subgraph of X in G.*

Procedure Tracebasedslice

INPUT: Pair of PDGs *G1 (V1, E1), G2 (V2, E2)*

OUTPUT: Maximum common subgraph of G1 and G2

1. *matchedNodes1* $\subseteq V1$, *matchedNodes2* $\subseteq V2$
2. *matchedNodes1* $\leftarrow \emptyset$, *matchedNodes2* $\leftarrow \emptyset$
3. *noPredecessor1*$\leftarrow$*{n*$\in$*V1:n hasn't predecessor}*
4. *noPredecessor2*$\leftarrow$*{n*$\in$*V2:n hasn't predecessor}*
5. *continueMatching* $\leftarrow$ *true*
6. *while (continueMatching)*
7. *continueMatching*$\leftarrow$*false*
8. *tempMatching*$\subseteq V1 \times V2$
9. *tempMatching*$\leftarrow \emptyset$
10. *sortedNeighbours1*$\leftarrow$ *sortVertices* (*getPredecessors (V1, matchedNodes1))*
11. *sortedNeighbours2*$\leftarrow$ *sortVertices (getPredecessors (V2, matchedNodes2))*
12. *tempMatching*$\leftarrow$*makeCorrespondence(sortedNeighbours1, sortedNeighbours2)*
13. *sortedNeighbours1*$\leftarrow$ *sortVertices* (*getSuccessors (V1, matchedNodes1))*
14. *sortedNeighbours2*$\leftarrow$ *sortVertices (getSuccessors (V2, matchedNodes2))*
15. *tempMatching*$\leftarrow$*tempMatching* $\cup$ *makeCorrespondence(sortedNeighbours1, sortedNeighbours2)*
16. *if tempMatching is empty*
17. *tempMatching*$\leftarrow$*makeOneCorrespondence (noPredecessor1, noPredecessor2)*
18. *if tempMatching is not empty*
19. *continueMatching*$\leftarrow$*true*
20. *for all (v1, v2)* $\in$ *tempMatching*
21. *if checkPredecessors*(*matchedNodes1*, *matchedNodes2, v1, v2*) *and checkSuccessors(matchedNodes1*, *matchedNodes2, v1, v2)*
22. *matchedNodes1* $\leftarrow$*matchedNodes1* $\cup$ *{v1}*
23. *matchedNodes2*$\leftarrow$*matchedNodes2* $\cup$ *{v2}*
24. *return inducedSubGraph(matchedNodes1, G1), inducedSubGraph(matchedNodes2, G2)*

After detecting the maximum common subgraph, the Tracebasedslice procedure returns the common part of two functions. If it satisfies MN and MP, then the obtained results are saved and visualized.

3.3 Visualization of Binary Code Clones

The last stage of the tool is visualization of clones. The purpose of the created graphical interface is to demonstrate the assembler code of the obtained clones, the percentage of their similarity, the corresponding graphs and maximum common subgraph (Fig. 3).

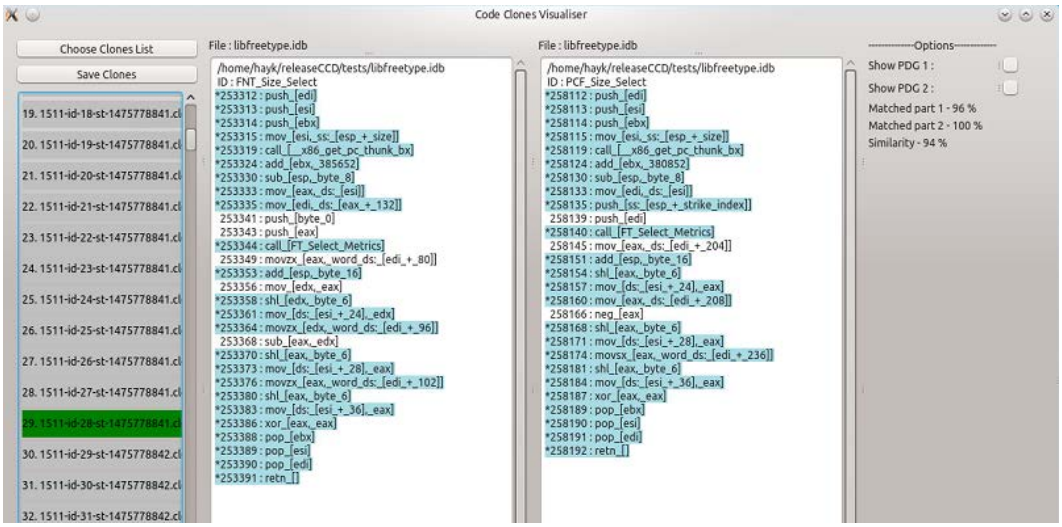

Fig. 3. Visualization of detected clones

4. Results

To assess the effectiveness of the tool, it is tested on various real projects. Two successive versions are analyzed for each project. The tool analyzes the pairs of functions, which have the same name in old and new versions of projects. As the same function most likely will change at a few instructions in new version, then they should be clones. If the function is not changed, then they should be clones of the second type. The target machine for testing is Intel i5, 4 cores, RAM - 16 GB. The results on several programs are shown in Table 1.

Table 1: Results

Project name	Version		Binaries sizes (MB)		Functions count with the same name	Analyze time	Detected clones count (MP = 50%)	Detected clones count (MP = 90%)
	old	new	old	new				
grep	3.0	3.1	0.74	0.74	308	10s	307	299
gdb	8.0	8.1	49	66	12707	6m 14s	11894	10683
findutils	4.4.1	4.4.2	1.1	1.1	484	12s	484	482
gcc	4.9.0	5.4.0	3.2	3.4	1041	50s	956	662
git	2.6.0	2.9.5	9.4	9.8	3257	1m 15s	3098	2627
bison	2.3	2.4	1.3	1.5	498	19s	498	497

5. Conclusion

In this paper, the main approaches for finding binary code clones are observed. Developed a multiplatform, scalable tool that allows finding all three types of code clones. The results on real world programs prove high accuracy and scalability of the tool.

References

[1] [Online]. Available: https://www.zynamics.com/binnavi.html

[2] [Online]. Available: https://www.zynamics.com/binnavi/manual/html/reil_language.htm

[3] S. Ducasse, M. Rieger and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proceedings of the 15th International Conference on Software Maintenance*, pp. 109-118, 1999.

[4] T. Kamiya, S. Kusumoto and K. Inoue, "CCFinder: A multilinguistic tokenbased code clone detection system for large scale source code," in *IEEE Transactions on Software Engineering*, vol 28, no. 7, pp. 654-670, Jul 2002.

[5] I. Baxter, A. Yahin, L. Moura and M. Anna, "Clone detection using abstract syntax trees," in *Proceedings of the 14th IEEE International Conference on Software Maintenance*, pp. 368-377, 1998.

[6] R. Tairas and J. Gray, "Phoenix-based clone detection using suffix trees," in *Proceedings of the 44th Annual Southeast Regional Conference*, pp. 679-684, 2006.

[7] L. Jiang, G. Misherghi, Z. Su and S. Glondu, "DECKARD : Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th International Conference on Software Engineering*, pp. 96-105, 2007.

[8] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Proceedings of the 8th International Symposium on Static Analysis*, pp. 40-56, 2001.

[9] J. Krinke, "Identifying similar code with program dependence graphs," in *Proceedings of the 8th Working Conference on Reverse Engineering*, pp. 301-309, 2001.

[10] M. Gabel, L. Jiang and Z. Su, "Scalable detection of semantic clones," in *Proceedings of 30th International Conference on Software Engineering*, pp. 321-330, 2008.

[11] S. Sargsyan, S. Kurmangaleev, A. Baloian and H. Aslanyan, "Scalable and Accurate Clones Detection Based on Metrics for Dependence Graph," *Mathematical Problems of Computer Science*, vol. 42, pp. 54-62, 2014.

[12] A. Avetisyan, S. Kurmangaleev, S. Sargsyan, M. Arutunian and A. Belevantsev, "LLVM Based code clone detection framework," in *10th International Conference on Computer Science and Information Technologies*, pp. 100-104, 2015.

[13] S. Sargsyan, S. Kurmangaleev, A. Belevantsev and A. Avetisyan, "Scalable and accurate code clone detection," *Programming and Computer Software*, vol. 6, pp. 9-17, 2015.

[14] S. Sargsyan, S. Kurmangaleev, A. Belevantsev, H. Aslanyan and A. Baloian , "Scalable tool for code clone detection based on semantic analysis of program," *Trudy. ISP RAS*, vol. 1, pp. 39-50, 2015.

[15] J. Jang and D. Brumley, "Bitshred: Fast, scalable code reuse detection in binary code," *CyLab*, p. 28, 2009.

[16] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, pp. 422-426, 1970.

[17] A. Schulman, "Finding binary clones with opstrings function digests: Part III," *Dr. Dobb's Journal*, pp. 56-61, 2005.

[18] M.E. Karim, A. Walenstein, A. Lakhotia and L. Parida, "Malware phylogeny generation using permutations of code," *Computer Virology*, pp. 13-23, 2005.

[19] D. Bruschi, L. Martignoni and M. Monga, "Code normalization for self-mutating malware," *IEEE Security & Privacy*, pp. 46-54, 2007.

[20] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan and Z. Su, "Detecting code clones in binary executables," in *Proceedings of the 18th International Symposium on Software Testing and Analysis*, pp. 117-128, 2009.

[21] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," in *47th Annual IEEE Symposium on Foundations of Computer Science* , pp. 459-468, 2006.

[22] M. R. Farhadi, B. C. M. Fung, P. Charland and M. Debbabi, "BinClone: Detecting Code Clones in Malware," in *2014 Eighth International Conference on*, San Francisco, CA, pp. 78-87, 2014.

[23] T. Dullien, E. Carrera, S. Eppler and S. Porst, "Automated attacker correlation for malicious code," DTIC Document, 2010.

[24] 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*, pp. 349-360, 2014.

[25] D. E. Krutz and E. Shihab, "CCCD: Concolic Code Clone Detection," *WCRE*, 2013. D. E. Krutz and E. Shihab, "CCCD: Concolic code clone detection," *2013 20th Working Conference on Reverse Engineering*, Koblenz, 2013, pp. 489-490.

[26] https://www.hex-rays.com/products/ida.

[27] H. K. Aslanyan, S. F. Kurmangaleev, V. G. Vardanyan, M. S. Arutunian and S. S. Sargsyan, "Platform-independent and scalable tool for binary code clone detection," *Trudy ISPRAN/Proc. ISP RAS*, vol. 1, no. 2, pp. 215-226, 2016.

Submitted 02.08.2017, accepted 23.11.2017.

Երկուական կոդի կլոնների արդյունավետ և ճշգրիտ որոնում

Հ. Ասլանյան

Ամփոփում

Ծրագրային ապահովման մշակողները հաճախ պատճենում և տեղադրում են որոշակի կոդ, քանի որ նրանք նախընտրում են օգտագործել նախապես եղած լուծումը կամ մասնակի լուծումը որպես այլ խնդրի լուծման հիմք: Այնուամենայնիվ, դա կարող է հանգեցնել տարբեր սխալների, ինչպես նաև նախնական և երկուական կոդերի մեծացմանը: Կոդի նմանատիպ մասերը (կլոններ) երկուական կոդում գտնելու խնդիրը դառնում է ավելի կիրառելի, երբ նախնական կոդը հասանելի չէ: Բացի այդ, տարբեր ձևափոխությունների ընթացքում կոմպիլյատորը կարող է պատճենել կոդի որոշ մասեր և ստեղծել կոդի կլոններ, որոնք գոյություն չունեն նախնական կոդում: Երկուական կոդի կլոնների հայտնաբերումը կարող է օգտագործվել վնասակար կոդի, սիմվոլիկ սխալների, հեղինակային իրավունքի խախտումների հայտնաբերման համար: Հոդվածում քննարկվում է երկուական կոդերի հայտնաբերման գոյություն ունեցող մեթոդները և ներկայացվում է երկուական կոդի կլոնների հայտնաբերման նոր մեթոդ: Այն բաղկացած է երեք հիմնական փուլերից: Առաջին փուլը հիմնված է Binnavi համակարգի վրա և կառուցում է ծրագրի կախվածության գրաֆներ յուրաքանչյուր երկուական ֆունկցիայի համար: Գրաֆները ստեղծվում են՝ հիմնվելով REIL (Reverse Engineering Intermediate Language) ճարտարապետությունից անկախ լեզվի վրա: REIL ներկայացումը կարելի է ստանալ մի քանի ճարտարապետությունների ասեմբլերների համար (x86, x86-64, ARM, MIPS, PPC), դրանով ապահովելով գործիքի անկախությունը կոնկրետ ճարտարապետությունից: Երկրորդ փուլը հայտնաբերում է կլոններ՝ նախկինում ստեղծված գրաֆների հիման վրա: Երկու ծրագրի կախվածությունների գրաֆների առավելագույն ընդհանուր ենթագրաֆի հայտնաբերման համար առաջարկվում է բազմանդամային մոտարկող ալգորիթմ: Երրորդ փուլում ստացված կլոնները ցուցադրվում են հետագա վերլուծության համար:

Эффективное и точное обнаружение клонов бинарного кода

А. Асланян

Аннотация

Разработчики программного обеспечения обычно копируют и вставляют определенный фрагмент кода, поскольку предпочитают использовать предварительно написанное решение или частичное решение в качестве основы для решения своей проблемы. Однако это может привести к различным ошибкам, а также увеличить размер исходного и бинарного кода. Поиск похожих частей кода (клонов) в двоичном коде становится более применимым, если исходный код недоступен. Кроме того, компилятор во время различных преобразований может скопировать некоторые части кода и создать клоны кода, которых нет в исходном коде. Обнаружение клонов бинарного кода используется для нахождения вредоносного кода, поиска семантических ошибок, обнаружения нарушения авторских прав и т. д. В этой статье обсуждаются существующие методы обнаружения клонов бинарных кодов и вводится новый метод их обнаружения. Он состоит из трех основных этапов. Первый этап основан на платформе Binnavi и генерирует графы зависимостей программы для каждой двоичной функции. Графы создаются на основе независимого от платформы языка REIL (Reverse Engineering Intermediate Language). Представление REIL поддерживается для нескольких архитектур (x86, x86-64, ARM, MIPS, PPC), что обеспечивает независимость инструмента от целевой архитектуры. Второй этап обнаруживает клоны на основе ранее созданных графов. Предлагается полиномиальный эвристический алгоритм для нахождения максимального общего подграфа двух графов зависимостей программы. На третьем этапе полученные клоны визуализируются для ручного анализа.