Program Equivalence Using Neural Networks

Tiago M. Nascimento[1,2], Charles B. Prado[1], Davidson R. Boccardo[1], Luiz F.R.C. Carmo[1], and Raphael C.S. Machado[1]

[1] INMETRO - National Institute of Metrology, Normalization and Industrial Quality Rio de Janeiro - Brazil
[2] UFRJ - Federal University of Rio de Janeiro - Brazil
{tmnascimento,cbprado,drboccardo,lfrust,rcmachado}@inmetro.gov.br

Abstract. Program equivalence refers to the mapping between equivalent codes written in different languages – including high-level and low-level languages. In the present work, we propose a novel approach for correlating program codes of different languages using artificial neural networks and program characteristics derived from control flow graphs and call graphs. Our approach correlates the program codes of different languages by feeding the neural network with logical flow characteristics. Our evaluation using real code examples shows a typical correspondence rate between 62% and 100% with the very low rate of 4% false positives.

1 Introduction

Equivalence between two programs can be characterized by their executing behavior. The behavior equivalence can be established between distinct languages – including high-level and low-level languages. The *equivalence problem* refers to, given two programs, deciding if they present the same executing behavior or not. In the following, we present some scenarios of Software Engineering concerned with the equivalence problem.

1. **Platform migration**. Consider a developer who, for historic reasons, has a small part of a certain system written in a distinct programming language from the rest of the system. For maintainability reasons, this developer may track the homogeneity of the system by rewriting the routines in the predominant language of the system. One way to determine whether the "rewritten routines" were properly coded is to verify if they are equivalent to the "older routines".
2. **Legacy software recovering**. Consider a developer who has a software system in production that, due to difficulties in its software configuration management, needs to "recover the baseline" of the software system in production, *i.e.*, for each software module that has a binary code in execution, the developer must identify among several versions of source code, which corresponds to that binary in execution. One possibility for this task could be recompiling the distinct source code modules, comparing the generated binaries with the ones in the production environment. This approach however, besides being impractical – given the huge amount of versions and

J. Suzuki and T. Nakano (Eds.): BIONETICS 2010, LNICST 87, pp. 637–650, 2012.

compilations that should be conducted, still has the possibility of failure in the case that the setting parameters are not exactly the same as those used in the compilation of the modules currently in the production.

3. **Introduction of non-intentioned behavior in the compilation process**. This scenario refers to the fact that the compilation process is a part of the software development and must be validated. In fact, the compilation process may introduce *bugs* and non-intentioned behavior in the software [1]. Besides, the software code and the compiler may be deliberately corrupted by the insertion of a malicious behavior (backdoor). Hence, it becomes interesting to have tools to directly compare the behavior described by source and binary code, independently of the compilation process.
4. **Software acquisition management**. Assume that a software manufacturer wishes to outsourcing the development of some libraries in a project. For so, the manufacturer gives some specifications to an independent developer, which returns the specified product developed. Naturally, such a product should be submitted to a battery of tests in order to characterize it according to the specifications. However, these tests normally are ineffective with hidden malicious behavior. Such a behavior is typically activated by the insertion of hidden undocumented commands — for instance, by using a counterintuitive sequence of keystrokes[1]. The detection of hidden behavior only can be done through of code analysis. In this case, the manufacturer may require not only the binary code but also the associated source code. Once the source code be analyzed, further step requires the mapping (equivalence) between the source and the binary code.

In the previous examples, we see that frequently the equivalence problem refers to the mapping of a binary code from a source code or of a binary code from another binary code, which we term "traceability". A simple and direct way of performing such "traceability" is to reproduce exactly the development environment of the software developer and to compile the source code, verifying whether the generated binary code is as expected. For this approach we must assure that the compilation environment is the same. Such a hypothesis, however, may be impractical — as the case (2), in which the compilation settings may be lost due to flaws in the configuration management, or as the case (4), which would restrict the developer to a unique environment. Another drawback is related to the cost and the complexity required to keep several software development environments. Moreover, as demonstrated in the Thompson Turing Award Lecture [1]: unless the "language transformation" performed by the compiler can be completely characterized — which would require a binary analysis of the compiler code — it is not possible to guarantee that the compilation process, itself, does not introduce some kind of flaw or malicious behavior into the software (this is closely related to the example in case (3)). Another way of performing the software traceability is to audit the software development environment of the software developer. Such an approach presents disadvantages similar to those described

[1] These sequences of commands are sometimes called *easter eggs*.

in the previous paragraph, and it is likely to be ineffective when dealing with a malicious developer.

Summarizing, binary code verification is the only true way to detect hidden capabilities, as demonstrated by Thompson in his Turing Award Lecture [1]. Lest Thompson's paper be considered theoretical, his ideas have been put into practice by the malware W32.Induc.A [2]. On large and complex systems, however, binary verification can require long and laborious work to integrally track variable manipulations and to perform vulnerability analysis, so that a usual approach is to conduct such verification on the source code. Depending on the architectural complexity of the software, this software can be explicitly submitted to a white-box approach entailing a source code analysis. However, source code verification is not sufficient enough to give any guarantee about the behavior of the related binary code. To certify that the binary code being executed works properly, one must guarantee that such binary code was, in fact, generated from the approved source code through an honest compilation process.

Source code verification does not preclude the verification as to whether the binary code corresponds to its source code. Traceability of executable codes is the process for establishing the correspondence between source and object codes. That is, once the source code of a given software version is analyzed, evaluated and approved, it is necessary to verify whether a given binary code — which will be, in fact, in execution — corresponds to that source code.

In the present work we propose a different strategy to verify whether two programs of different languages (typically source code and binary machine code) describe the same software behavior. This paper presents a novel approach for performing program equivalence of program codes of different languages by using artificial neural network (ANN). More specifically, we collected properties of the control flow graphs and call graph, such as number of edges, number of nodes and number of functions, and we used an ANN to discover the degree of similarity of the program languages based on the collected properties.

The rest of the paper is structured as follows. Section 2 discusses the related works on program equivalence. Section 3 describes our proposed method by characterizing the properties extraction of program languages and by showing the application of an artificial neural network to discover the degree of similarity of the program languages. Section 4 presents the empirical evaluation of the method presented, followed by our concluding remarks.

2 Related Works

There are not many contributions related with program equivalence in the literature. However, there were works for verifying and analyzing of source and binary codes that may assist in achieving the proposed approach.

Quinlan *et al.* [3] proposed a framework for software defects verification (binary or source). However, it does not compare source and binary codes. Hassan *et al.* [4] observed that the architecture of some programs is intrinsically related with the their source and binary codes. They used two types of extractors:

a transfer control extractor of a code binary (LDX) and a label extractor of a C source code (CTAGX). After the extraction process, they conducted a comparison of the obtained results in order to infer the software architecture. Hatton [5] investigated the defect density as a relationship between a binary code and a source code. For so, he used the size (number of rows) of the source code and its defect per 1,000 lines to seek the relationship with the binary code. Neither of these works addresses the program equivalence problem.

Buttle [6] utilizes the program logical structure (control flow graph) of the binary code to match with the program logical structure of the source code. We also use program flow characteristics to match binary codes, however, in our approach other relationships that may coexist between source and binary codes in order to obtain a better matching were considered.

On a tangential direction there has been significant work in binary differing with the intent to review sequential versions of the same piece of software, to analyze malware variants of the same high-level language and to analyze security updates [7,8]. Most of these works use graph matching to compare the binaries. A good summary of these works may be found in [9], which also introduces anti-differing techniques with the intent to thwart algorithms based on graph matching. Research results from differing binaries [7,8,9], may thus be borrowed for the program equivalence problem for analogous constraints.

Some contributions in security use neural networks for cryptography approaches. A new digital image encryption algorithm using neural networks is presented in [10]. Such algorithm employs a hyper-chaotic cellular neural network using chaotic characteristics of dynamic systems.

Artificial intelligence based methods for software validation, verification and reliability can be found on the literature. The approaches in [11] and [12] propose the use neural networks for software reliability prediction. The former uses the prediction for software defects fix effort, while the second the prediction system is based on neural network ensembles. In [13], a neural network is used to predict a fault-prone module in a web application. In [14], a self-organizing system for reliability of modules is constructed.

The use of artificial neural networks was previously considered in [15], which described a method for verifying the correspondence between source and binary codes using artificial neural networks. The approach described in this paper improves upon that work by refining the extracted properties and by applying the method for program equivalence of distinct languages.

3 Proposed Approach

Our approach for program equivalence involves two steps. In the first step, we use software tools to extract characteristics of distinct program languages. Such characteristics could be simple ones, such as size, or more sophisticated ones, such as those derived from the control flow graphs or call graphs. In the second step, we use a nonlinear nondeterministic classifier to determine the correspondence of the program languages. In this section, we show how this approach was put in

practice with the use of four characteristics (size, number of procedures, number of nodes of control flow graph and number of edges of control flow graph) and an artificial neural network as classifier.

3.1 Extraction Process

In the compilation process there is a lot of lost information that should be taken into account when designing a program equivalence approach. The amount of available information of the compiled code is platform and compiler-specific. In the following, we mention some properties regardless of the source and binary codes, which may be explicitly or implicitly available, in order to give insights about the complexity of designing a program equivalence method.

Considering the fact that most embedded softwares are written in imperative language, properties such as variable names, variable types and procedure names may be lost during the compilation process since the compiler goal is to maximize the performance. This process normally decreases the legibility of the binary code, so representing low confidence for the mentioned properties to use in our program equivalence problem.

Data contents are not explicitly available in the source and binary codes. Nonetheless, these contents may be computed by data-flow analysis. The scope of the data-flow analysis for source and binary is faintly different. In the source code, the scope is at the variable level, meanwhile, in the binary code it is at memory and registers. This difference certainly increases the number of instructions contained in the binary in comparison to the respective number of the source code. Besides being positive for compiler data optimizations since it tracks fine-grained transitions, it requires more memory to analyze more code lines. Since the extraction of this property is complex, it is not suitable for our program equivalence problem.

The control sequentiality is certainly kept in the binary, albeit, its tree of execution is not clearly structured as such in the source code. A good summary of algorithms to structure the control sequentiality may be found in [16]. The control sequentiality describes the program logic of a certain code, and it can be characterized by call graphs and the individual function flow graphs. The call graphs show the caller-callee relationship. The individual function flow graphs represent the basic blocks and its flow of information based on conditional and unconditional branches.

The characteristics used in our program equivalence method are based on the program logic of the code (control sequentiality). These characteristics are number of nodes, edges and functions. The sizes in bytes of the codes were also used in our method. The sizes of the codes were obtained in a straightforward manner, however, the number of nodes and edges of the control flow graph for the source code were obtained by using a shell script that counted these properties. Such a control flow graph of the source code was built from the Gnu Compiler Collection with the parameter “fdump-tree-cfg”. For the extraction of the same characteristics from the control flow graph in the binary code, we used an idapython script over the IDA disassembler [17]. The number of the

functions for the source codes and binary codes were extracted from the call graphs, generated by the GNU cflow [18] tool for the source code and by the IDA disassembler for the binary code.

3.2 Artificial Neural Network

The fundamental point in program equivalence is to establish an association of the program languages by linking their intrinsic logical characteristics. For such, it is fundamental to find an efficient approach that combines the four different parameters extracted from logical program code (number of nodes, edges, functions, and the sizes of the codes (in bytes)).

At first analysis, some linear separation methods could be investigated to solve this problem. However, as will be shown in the further sections, this problem is both nonlinear and highly complex to solve using a linear method. For these reasons, we examine the use of artificial neural networks.

Neural networks, with their remarkable ability to derive meaning from complicated or imprecise data, can be used to extract patterns and detect trends that are too complex to be noticed by either humans or other computer techniques. A trained neural network can be thought of as an "expert" in the category of information it has been given to analyze. Obviously, there are many kinds of Back-propagation networks applied to a large set of different problems, like: classification, recognition, prediction and others.

In this work, the main focus will be on neural networks applied to the classification problem. For that, we propose the use of Cascade-Forward Back-propagation Neural Networks, since they are widely used in this context [19,20,21]. In the next section, the details of neural network implementation will be presented.

Neural Network Implementation. To design a neural network, four characteristics (number of nodes, edges, functions, and the sizes of the codes (in bytes)) were fed into the input nodes of the one fully-connected cascade forward network using the Back-propagation training procedure [22]. From the empirical analysis, the best neural configuration was built with two neurons in the hidden layer. For the output layer, only one neuron was used (see Figure 1).

In order to simulate the neural network, the Matlab Neural Network Toolbox [23] was used. The selected activation function for the neurons was the hyperbolic tangent sigmoid. The target vector for the training phase was defined by establishing a target value of 1 when the input parameters represent equivalent codes (called class of true association), otherwise the target value was set to -1 (called class of false association).

4 Experimental Evaluation

We now present the results of an empirical evaluation for mapping equivalent codes written in distinct languages. Three evaluations were performed:

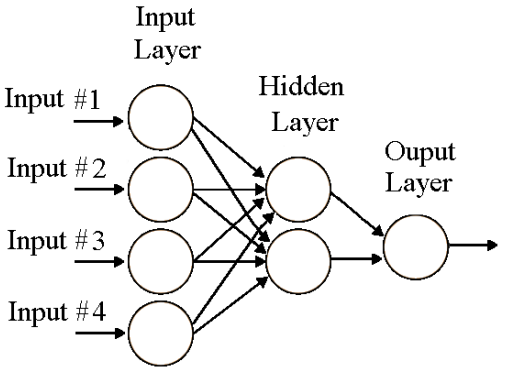

Fig. 1. Neural Network Topology

1) evaluation of malware modified binaries, 2) evaluation of binaries of different platforms and 3) evaluation of binaries generated from different compilers.

For all the evaluations, the percentage used for training phase was ≈70% and the remaining samples were used to verify the capacity of generalization of the ANN. We studied the improvements of our method by comparing it against Support Vector Machine (SVM) — a technique typically applied to the construction of classifiers [24,25]. Our empirical evaluation shows that our method produces more precise results than SVM.

4.1 Evaluation of Malware Modified Binaries

Since most malwares are predominant for Windows environment, we performed the evaluation in this environment. Before starting to infect the system in order to extract the characteristics of the infected codes, we established some security policies to avoid malware dissemination. We setup an isolated machine for the malware infection and characteristics extraction. The extracted characteristics of the infected codes were used to build the set of false association.

For this evaluation, we collected 94 source codes, taken from [26,27] and compiled them using the gcc compiler of Windows platform. Figure 2 shows the training set of true association, *i.e.*, characteristics extracted from 70 pairs of correlated source and binary codes. The 24 remaining were utilized in the evaluation phase. The histograms show the distribution of the normalized variables: edges, nodes, functions and size, respectively, in the x-axis and their frequency in the y-axis. The control flow graph characteristics (edges and nodes) were fed into input #1 and input #2 of the ANN, respectively. The number of functions (extracted from the call graph) was fed into input #3, and the size into input #4. All the characteristics were correlated by subtracting the source code characteristic from the binary code characteristic except for the size characteristic, in which a division was applied. Before inputing such parameters into the ANN,

a normalization step was necessary since our ANN only accepts values in the interval [-1,1]. The normalization of the parameters was calculated by dividing all data of each parameter by the largest value of the same parameter.

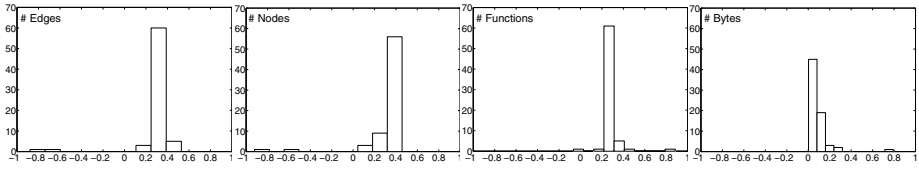

Fig. 2. Training set of the true association using correlated binary and source codes

The false association was created by infecting the 94 codes with four malwares (Virus.Cabanas.a, Virus.Win32.NGVCK.1003, Virus.Win32.Qudos.4250, Virus.Win32.Artelad.2173). Figure 3 shows the training set of false association using characteristics extracted from 280 infected samples (70 for each malware), in which the characteristics were correlated by subtracting the source code characteristic from the infected binary code except for the size characteristic, in which a division was applied. All the characteristics were normalized before inputing into the ANN. The 96 remaining infected samples were utilized in the evaluation phase.

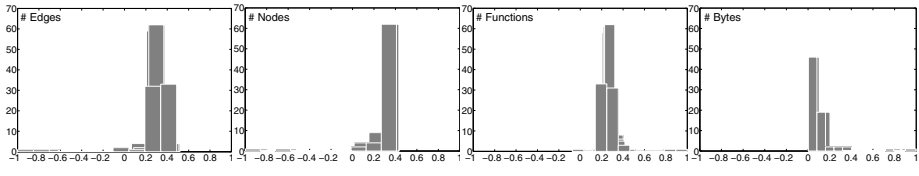

Fig. 3. Training set of the false association using malware modified binaries

Figures 4 and 5 show the sets utilized for the evaluation of the ANN. Table 1 shows the neural network results with respect to the number of hits and mistakes for both classes (true and false association). The data shows ≈4% (4/96) false positives and ≈37% (9/24) false negatives. The results of our approach are promising since the program equivalence is mainly concerned with a low rate of false positives.

Table 1. Neural network results of the evaluation of malware modified binaries

Class	True association	False association
True association	15	9
False association	4	92

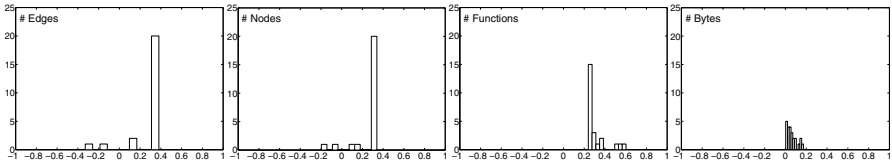

Fig. 4. Evaluation of the ANN using correlated binary and source codes

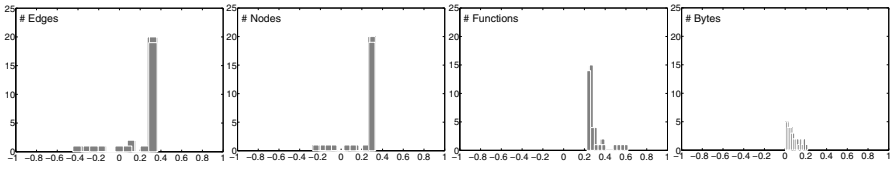

Fig. 5. Evaluation of the ANN using non-correlated binary and source codes

We compare the results of our proposed ANN with the results achieved using a Support Vector Machine (SVM). The classifier used was a two-norm and soft-margin SVM. The kernel function that maps the training data into kernel space was linear and the method to find the separating hyperplane was quadratic programming. However, the SVM did not converge to any separating hyperplane for the training sets exhibited in Figures 2 and 3. However, using only one malware (Virus.Cabanas.a) for the false association we found a division of the 4d-space — that is, a 3-d hyperplane that divides the 4-d space into two semi-spaces. In this experiment, the SVM correctly identified 19 among the 24 pairs of corresponding codes, with 7/23 (≈30%) ratio of false positives. Observe that the SVM method achieved a reasonable accuracy regarding the identification of corresponding codes (exactly the same 19/24 as the ANN), but with a much higher number of false positives (7/23 against the 1/24 ratio of the ANN).

4.2 Evaluation of Binaries of Different Platforms

For this evaluation, we compiled 81 source codes using both gcc compiler on the Windows environment and gcc compiler on a Linux-like environment to build the true association. Figure 6 shows the training set of true association, *i.e.*, characteristics extracted from 60 pairs of correlated Windows binary and Linux binary. The 21 remaining were utilized in the evaluation phase. The normalization and extraction processes are analogous to the ones of the previous evaluation.

Figure 7 shows the training set for the false association, created naively by random values. Figures 8 and 9 show the sets utilized for the evaluation of the ANN. Table 2 shows the neural network results with respect to the number of hits and mistakes for both classes (true and false association). The data shows ≈9% (2/21) false positives and zero false negatives. We used the same training sets exhibited in Figures 6 and 7 to feed the SVM. The SVM correctly identified

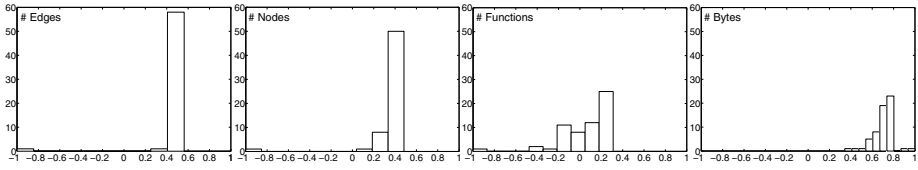

Fig. 6. Training set of true association using correlated binaries of different platforms

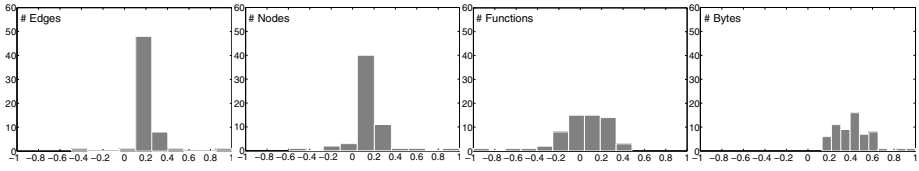

Fig. 7. Training set of the false association using non-correlated binaries of different platforms

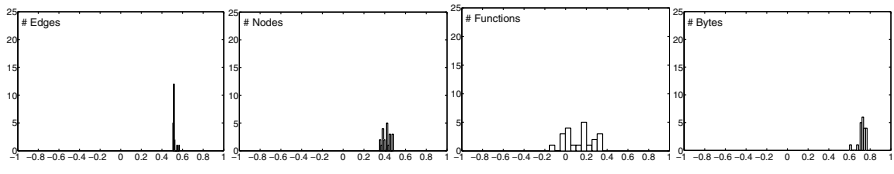

Fig. 8. Evaluation of the ANN using correlated binaries of different platforms

Table 2. Neural network results of the evaluation of binaries of different platforms

Class	True association	False association
True association	21	0
False association	2	19

all the 21 pairs of corresponding codes as the results of our ANN, with 3/21 (≈14%) ratio of false positives against 2/21 ratio of our ANN approach.

4.3 Evaluation of Binaries Generated from Different Compilers

For this evaluation, we compiled 83 source codes using the gcc compiler and the Borland C++ compiler, both on the Windows environment. Figure 10 shows the training set of true association, *i.e.*, characteristics extracted from 63 pairs of correlated binary generated from the compilers above. The 20 remaining were utilized in the evaluation phase. The normalization and extraction processes are analogous to the ones of the previous evaluation.

Figure 11 shows the training set for the false association, created naively by random values. Figures 12 and 13 show the sets utilized for the evaluation of the ANN. Table 3 shows the neural network results with respect to the number of

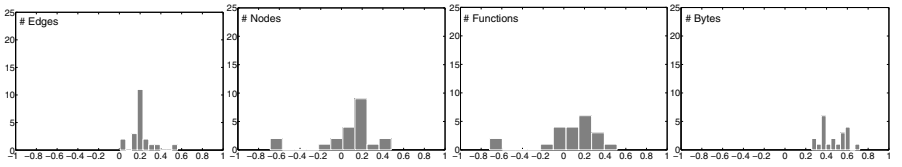

Fig. 9. Evaluation of the ANN using non-correlated binaries of different platforms

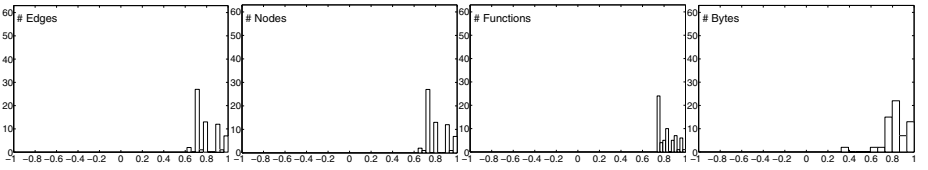

Fig. 10. Training set using correlated binaries generated by different compilers

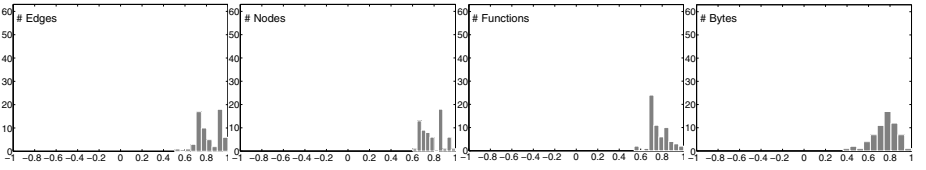

Fig. 11. Training set using different non-correlated compiled binaries

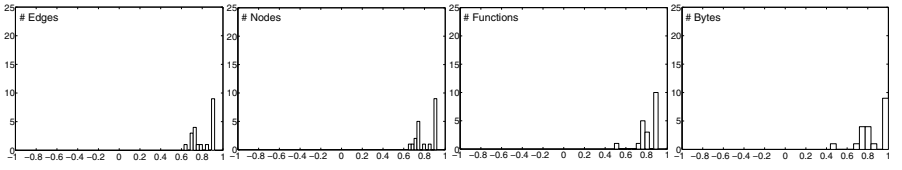

Fig. 12. Evaluation of the ANN using correlated binaries generated by different compilers

Table 3. Neural network results of the evaluation of binaries compiled with distinct compilers

Class	True association	False association
True association	19	1
False association	1	19

hits and mistakes for both classes (true and false association). The data shows 5% (1/20) false positives and 5% (1/20) false negatives. We used the training sets exhibited in Figures 10 and 11 to feed the SVM. The SVM correctly identified 18/20 pairs of corresponding codes against 19/20 of our ANN, with 10/20 (50%) ratio of false positives against 1/20 ratio of our ANN approach.

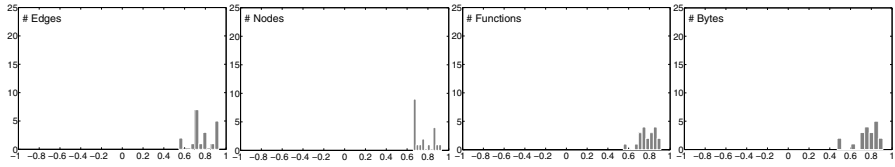

Fig. 13. Evaluation of the ANN using non-correlated binaries generated by different compilers

5 Conclusions

This paper deals with the issue of program equivalence of different languages — including high-level and low-level languages. The problem is fundamental for software validation: since the software evaluation is frequently based on source code analysis, it is important to guarantee that the compilation process did not introduce security flaws, backdoors or unwanted behaviors.

In the present work, we tackle the problem of program equivalence by extracting meaningful characteristics of program codes, obtaining such characteristics from program call graphs and control flow graphs. We use such characteristics to feed a nondeterministic classifier that decides whether a binary code corresponds to a given source code or whether a binary corresponds to another binary of a different platform or compiler. The originality of our approach lies on the extraction of characteristics of the call graphs and control flow graphs of the program codes, and on the use of an artificial neural network to decide the legitimacy of a binary code based on those characteristics. The performance of the proposed approach is well characterized through an experimental evaluation that, besides confirming a very low rate of false positives (considered as a basic requirement), also provides a reasonable amount of false negatives.

It could be argued that the proposed approach would not work in detecting simple binary code modifications that do not alter call graphs and control flow graphs, such as a change of a constant. We observe, however, that such modification would be immediately noted by the conventional functional tests performed on devices under verification. The kind of modification that our approach proposes to encounter is more subtle. For example, a malicious manufacturer could leave a backdoor that when activated transfers the execution control to a malicious behavior. Such a malicious modification would hardly be noticed by functional tests. However, it would not be possible for the manufacturer to include a backdoor without modifying the call graph and the control flow graph of the binary code. This makes the malicious modifications exactly the ones amenable to our approach based on call graph and control flow graph parameters.

An open question in the proposed approach, and the subject of ongoing research, concerns the necessity of also considering obfuscated codes during the training phase, to better understand its implications. For example, control-flow obfuscation alters the flow of control of the application by reordering statements, procedures, loops, obscuring flow of control using opaque predicates and

replacing transfer flow instructions. Using such obfuscation some properties used in our approach may change, so violating our results. Other possible avenues of research involve merging our current technique with data-flow strategies to circumvent attacks such as modification of variable contents. Finally, in future works we plan to investigate which other graph invariants — crossing number, cycle covering, chromatic number etc. — are meaningful to improve the correspondence of codes written in different languages.

References

1. Thompson, K.: Reflections on trusting trust. Commun. ACM 27(8), 761–763 (1984)
2. McDonald, J.: Delphi falls prey (2009), `http://www.symantec.com/connect/blogs/delphi-falls-prey` (last accessed October 2009)
3. Quinlan, D., Panas, T.: Source code and binary analysis of software defects. In: CSIIRW 2009: Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research, pp. 1–4. ACM, New York (2009)
4. Hassan, A.E., Jiang, Z.M., Holt, R.C.: Source versus object code extraction for recovering software architecture. In: WCRE 2005: Proceedings of the 12th Working Conference on Reverse Engineering, pp. 67–76. IEEE Computer Society, Washington, DC (1995)
5. Hatton, L.: Estimating source lines of code from object code. In: Windows and Embedded Control Systems (2005), `http://www.leshatton.org/Documents/LOC2005.pdf`
6. Buttle, D.L.: Verification of Compiled Code. PhD thesis, University of York, UK (2001)
7. Wang, Z., Pierce, K., McFarling, S.: Bmat - a binary matching tool for stale profile propagation. The Journal of Instruction-Level Parallelism (2002)
8. Flake, H.: Structural comparison of executable objects. In: Proc. of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA). IEEE Computer Society (2004)
9. Oh, J.: Fight against 1-day exploits: Diffing binaries vs anti-diffing binaries. In: Blackhat Technical Security Conference (2009)
10. Zhenga, J.: A digital image encryption algorithm based on hyper-chaotic cellular neural network. Journal Fundamenta Informaticae (2009)
11. Zeng, H., Rine, D.: A neural network approach for software defects fix effort estimation. In: IASTED Conf. on Software Engineering and Applications, pp. 513–517 (2004)
12. Zhenga, J.: Predicting software reliability with neural network ensembles. Expert Systems with Applications (36), 2116–2122 (2007)
13. Reddy, C.S., Raju, K.V.S.V.N., Kumari, V.V., Devi, G.L.: Fault-prone module prediction of a web application using artificial neural networks. In: Proceeding (591) Software Engineering and Applications (2007)
14. Lenic, M., Povalej, P., Kokol, P., Cardoso, A.I.: Using cellular automata to predict reliability of modules. In: Proceeding (436) Software Engineering and Applications (2004)
15. Boccardo, D.R., Nascimento, T.M., Machado, R.C., Prado, C.B., Carmo, L.F.R.C.: Traceability of executable codes using neural networks. In: Proceedings of the Information Security Conference (2010) (to appear)

16. Moretti, E., Chanteperdrix, G., Osorio, A.: New algorithms for control-flow graph structuring. In: CSMR 2001: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering, p. 184. IEEE Computer Society, Washington, DC (2001)
17. IdaPro: Ida pro - disassembler (2010), http://www.hex-rays.com/idapro/ (last accessed January 2010)
18. Poznyakoff, S.: Gnu cflow (2010), http://savannah.gnu.org/projects/cflow (last accessed January 2010)
19. Ciocoiu, I.B.: Hybrid feedforward neural networks for solving classification problems. Neural Processing Letters 16(1), 81–91 (2002)
20. Asadi, R., Mustapha, N., Sulaiman, N.: New supervisioned multi layer feed forward neural network model to accelerate classification with high accuracy. European Journal of Scientific Research 33(1), 163–178 (2009)
21. Haykin, S.: Neural Networks: A Comprehensive Foundation. Prentice Hall (1998)
22. Hertz, J.A., Krogh, A.S., Palmer, R.G.: Introduction to the Theory of Neural Computation. Addison-Wesley, Redwood City (1991)
23. Moler, C.B.: MATLAB — an interactive matrix laboratory. Technical Report 369, University of New Mexico. Dept. of Computer Science (1980)
24. Men, H., Wu, Y., Gao, Y., Kou, Z., Xu, Z., Yang, S.: Application of support vector machine to heterotrophic bacteria colony recognition. In: CSSE (1), pp. 830–833 (2008)
25. Angulo, C., Ruiz, F., González, L., Ortega, J.A.: Multi-classification by using tri-class svm. Neural Processing Letters 23(1), 89–101 (2006)
26. Burkard, J.: C software (2010), http://people.sc.fsu.edu/~burkardt/ (Last accessed January 2010)
27. Oliveira Cruz, A.J.: C software (2010), http://equipe.nce.ufrj.br/adriano/c/exemplos.htm (last accessed January 2010)