Estimating Software Obfuscation Potency with Artificial Neural Networks

Daniele Canavese(✉), Leonardo Regano, Cataldo Basile, and Alessio Viticchié

Politecnico di Torino, Torino, Italy
{daniele.canavese,leonardo.regano,cataldo.basile, alessio.viticchie}@polito.it

Abstract. This paper presents an approach to estimate the potency of obfuscation techniques. Our approach uses neural networks to accurately predict the value of complexity metrics – which are used to compute the potency – after an obfuscation transformation is applied to a code region. This work is the first step towards a decision support to optimally protect software applications.

Keywords: Software protection · Code obfuscation · Potency · Neural networks

1 Introduction

Obfuscation is one of the most effective and used solution to protect the software against reverse engineering and tampering. Several obfuscation techniques are available in literature [5] that can be driven by different protection/performance degradation parameters. While some results proved that it is impossible to create a perfect obfuscator [2], empirical results showed that obfuscation works well in practice [3,21]. Several tools are also prominent at industrial level[1].

Nevertheless, the strength of obfuscation in mitigating the attacks cannot be formally defined. Collberg et al. have proposed to compute an effectiveness index, named *potency*, by measuring the changes in complexity metrics induced by the obfuscation transformations [5]. This value can only be measured *a posteriori*, after the actual application of the transformation. As it is impractical to apply and measure all the possible ways to apply obfuscation, protection experts are asked to (1) use their intuition and past knowledge to select the parts to protect and the most promising techniques to apply on each of them, then to (2) apply the protections, and to (3) actually measure the effects. This practice conflicts with the need that several companies have to protect different versions of one or more applications in a very short time.

This paper proposes to estimate *a priori* the potency of obfuscation techniques, before they are applied on a specific code region. We used Artificial

[1] Two examples of commercial obfuscators are Stunnix (http://stunnix.com) and Proguard (https://www.guardsquare.com/en/proguard).

G. Livraga and C. Mitchell (Eds.): STM 2017, LNCS 10547, pp. 193–202, 2017.
DOI: 10.1007/978-3-319-68063-7_13

Neural Networks (ANNs) to predict the changes that obfuscation techniques cause on a set of metrics. Then, the predicted metrics have been used to estimate the *predicted potency* of the protection. Finally, the predicted potency has been used to make decisions about the best way to protect the application. Our assessment has proved that the estimated potency allows making nearly optimal decisions in a very limited time. Our ANNs predict very well the metric changes when a single protection is applied on each asset. However, the prediction ability decreases when ANNs are serially connected to estimate the changes created by the subsequent application of more protections, because of the error propagation. Nonetheless, this paper discusses how to improve the prediction abilities and extend this approach to be used to build a decision support system.

This paper is structured as follows. Section 2 contains the background of our approach, Sect. 3 introduces our methodology with a simple motivating example. Sections 4 and 5 discuss how we gathered our training/test data and detail the achieved results. Finally, Sects. 6 and 7 list the related works, draw our conclusions and sketch the future work.

2 Background

Software protection has been a crucial research topic since the last decades [7]. One of the most enduring technique is obfuscation [12,15], a set of transformations that can be applied both on data and code and at different levels: source, byte, or binary level. These transformations aim at making the software less intelligible thus hardening any attack that implies software understanding.

Obfuscation is not provably secure, as it has been demonstrated that it can be reverted, even automatically [19]. Moreover, a general obfuscator able to protect an application in untrusted environments cannot be created [2]. Although obfuscation is actually a kind of *security through obscurity*, it enhances the level of protections of the applications, as it delays attacks (rather than preventing them at all), as empirically assessed by Ceccato et al. for code obfuscation [3], and by Viticchié et al. for data obfuscation [21].

Collberg et al. have proposed several measures to evaluate the effectiveness of an obfuscation [5]. The *potency* aims at evaluating the complexity introduced by the transformation and gives an index of how hard would be to understand the obfuscated code. Given the transformation $\mathcal{T} : P \xrightarrow{\mathcal{T}} P'$ that transforms the original program P into the obfuscated program P', the potency of $\mathcal{T}$ with respect to the program P can be obtained with the following formula (from [5]):

$$\mathcal{T}_{POT}(P) = \frac{E(P')}{E(P) - 1}$$

where $E(\cdot)$ is a complexity metric of a program. Therefore, the potency can only be computed after the transformation has been applied, or, as proposed by this work, by predicting the values of the metrics in the transformed program.

A *code region* is a portion of well-formed code, that is, a slice of syntactically valid code if parsed in isolation. A code region is unequivocally determined by

the containing source file, its starting and ending line numbers. Code regions are hierarchical, i.e. one can contain another. Therefore, a program (or one of its portions) can be seen as a tree where each node is a code region containing all its descendant nodes.

A *protection* is a technique that applies a transformation on the application source or binary code. The transformation applied by a protection can be fixed or depending on a set of parameters. We will use the definition *Protection Instance (PI)* to identify a selection of the parameters of a protection. In other words, a PI is a precise way to use a protection. For example, the Diablo binary code obfuscator [20], allows the selection of various obfuscation types (e.g. opaque predicates or control flow flattening), and the (expected) level of obfuscation effectiveness (passed as an integer). Hence, a Diablo protection instance includes one obfuscation technique and an integer for the expected effectiveness. The PIs of a protection are determined by the combinations of the allowed values of the parameters. Since some parameters can be unbounded, the cardinality of the PIs set may be infinite. However, the bare combination of the parameters domain sets may produce PIs that are meaningless or do not significantly differ one from another. Carefully selecting a set of PIs may reduce the decision space.

A *metric* is a measurement of a software feature. In this work, we only use binary level complexity metrics. Indeed, metrics computed at source level can be altered, thus invalidated, by the compiler (e.g. for optimisation purposes). Software metrics can be defined depending on several aspects of the software structure [4,10,16]. We concentrated on the the seven metrics that Collberg et al. proposed to use to compute the potency [5], however, the tool that we choose to work with, Diablo, is only able to compute the cyclomatic complexity and Halstead length. The *cyclomatic complexity* measures the complexity as the number of linearly independent paths in the program's control flow graph [16], an index of the nesting level easily computed on binary code. The cyclomatic complexity v of a control flow graph G is $v(G) = e - n + p$, where e is the number of edges, n is the number of nodes and p is the number of connected components of the graph. The *Halstead length* considers a program implementation as a sequence of *operators* and their relative *operands* [10]. The length N of a program, according to Halstead, is calculated as $N = N_1 + N_2$, where N_1 is the total number of operators and N_2 is the total number of operands.

3 Motivating Example

In this section we show how our approach can positively impact the process of protecting, by means of binary code obfuscation, the intellectual property of the algorithms underlying a test application. Our test application is *Sumatra*[2], an open-source command line tool written in C that compares DNA sequences. Sumatra provides two functionalities: comparing among all the DNA sequences in a single dataset, or (pairwise) comparing DNA sequences from two datasets.

[2] https://git.metabarcoding.org/obitools/sumatra/wikis/home.

We will behave like it was a proprietary software that must be protected against reverse engineering, to preserve the IP of the comparison algorithms.

Sumatra performs the DNA comparison in four consecutive phases. The first phase evaluates the command line arguments and calls the proper comparison functions. During the second phase, Sumatra parses the DNA datasets and stores them in several internal data structures. The third phase is the core of the program as it performs the actual comparison of the DNA sequences, so it should be obfuscated in a thorough way. The fourth and last phase presents the results to the user. These algorithm should be also strongly protected since they access the internal data structures, thus revealing information about the core algorithms.

We have classified the phases with a sensitivity value on a two level scale (i.e. high, low), as high sensitive assets should be protected with highly effective obfuscation techniques. Namely, the first, second and forth phases have a low sensitivity (and contains respectively one, five and three assets), and the third phase has a high sensitivity (and contains ten assets).

Mitigation is performed by applying on every asset a tuple of PIs, as applying them in different orders may lead to different results. Ideally, we must aim at reaching the best level of protection, by exploring the whole space of the possible ways to protect each asset. Thus, the solution space of the best protection becomes the set of all the PI combinations. As some PIs can be applied to the same code region several times, the number of PI combinations is theoretically unbounded. Therefore, we fixed the maximum length of PI sequences to l, so that the number of the sequences to consider is $c \geq \text{COMB}(l, n_p)$, where $\text{COMB}(l, n_p)$ is the number of l-combinations with repetition of n_p PIs.

To measure the overall effectiveness of a sequence of PIs σ on the asset a we introduced the *combined potency* P:

$$P_{\sigma,a} = \sum_{m \in M} w_m \cdot \pi_{\sigma,m,a} \tag{1}$$

where $\pi_{\sigma,m,a}$ is the potency of σ on a for the metric m (which uses the value of m before and after the protection has been applied [5]), M is the set of all the metrics m where the values $\pi_{\sigma,m,a}$ are computed and w_m are arbitrary weights.

Deciding the best protection would need the computation of the potency of all the possible σ sequences on each asset. Since we can protect all the assets with the same sequence of PIs, the number of times the target program needs to be obfuscated equals the number of the possible sequences[3].

Instead of computing the metrics *a posteriori*, we use ANNs to predict, from the value of the metrics of the unprotected application, the value of the metrics after the application of a PI. A single ANN predicts the changes of the metric m induced by the application of a single PI. Hence, we have trained an ANN for each pair (PI, m). The changes in the metric m when a sequence σ of PIs is estimated by using ANNs serially, i.e. by using the output of an ANN as input

[3] We do not take into account the case of nested assets, i.e. when an asset contains other asset. With nested assets, the number of compilation needed increases, since all the compilations should be repeated separately for each nesting level.

of the next one. Evaluating the potency of all possible PI sequences on all the n_a assets requires $b = n_a \cdot n_m \cdot (\text{COMB}(l, n_p) - 1)$ ANN simulations, where n_m is the number of metrics needed to compute the combined potency.

We experimentally demonstrated that our approach is faster, by testing it on Sumatra (see Sect. 5). We need more ANN interrogations than the protection applications required by the current approach, however, a ANN interrogation is typically completed in a few milliseconds, while the time needed for obfuscating assets (and often also compiling the whole application) may be in the order of seconds. Moreover, n_m is a small value and, from our experience in the ASPIRE project[4], increasing the application's lines of code slightly increases the number of assets n_a, but greatly increases the obfuscation (and compilation) time.

4 Data Set Acquisition

We trained the ANNs with a set of pre- and post-transformation metrics' values on a sufficiently large set of code regions. The Diablo linker [20] has been used to compute the complexity metrics on a set of code regions and to apply branch functions [12], function flattening [22] and opaque predicates [6] obfuscations. Diablo takes as input the object files and a JSON file, which allow the selection of the code regions of interest, then it maps these regions to the corresponding assembler instructions via the debugging information.

We selected 21 open source packages from the Debian repository. They encompass different areas such as scientific computations (e.g. the `libstarlink-pal` astrophysical library), security (e.g. the `ccrypt` cryptographic tool), network management (e.g. the `qmail` mail server) and utilities (e.g. the `bc` calculator).

For each application, our work-flow consisted of the following steps:

1. automatically divide the application in code regions (i.e. potential assets);
2. for each optimization flag `-Os`, `-O2` and `-O0` do:
   (a) compile the application without any obfuscation and extract the metrics;
   (b) for each PI, compile the application, apply the current PI to all the code regions in Step 1, and extract the post-PI metrics.

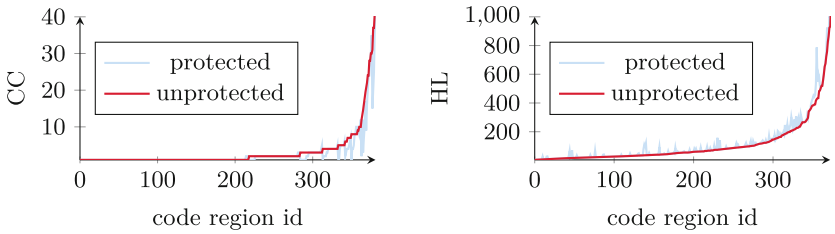
To implement the Step 1, we created a simple tool that parses the source files and automatically generates the JSON file with the code regions. The tool selects as valid code regions every function body and, recursively, each nested loop, if statement, or curly brackets block (see Fig. 1 for an example). With this tool, we identified 35510 code regions.

We have defined 2 PIs for each of the three supported Diablo obfuscations types. One PI has been defined (using an effectiveness parameter) to have a low protection/overhead obfuscation, for the other one, we required a high protection/overhead obfuscation. Figure 2 shows the graphs of the cyclomatic complexity (CC) and Halstead length (HL) before and after applying the PI

[4] https://aspire-fp7.eu/.

```
int function(int a, int b) {
    int c = 0, d = a;
    while (d > 0) {
        if (function2(b, d) % 2 == 0) {
            function3(TRUE);
            --c;
        }
        else {
            function3(FALSE);
            ++c;
        }
        --d;
    }
    return c;}
```

Fig. 1. Example of function splitting.



(a) Cyclomatic complexity. (b) Halstead length.

Fig. 2. Code region metrics for the "branch functions, high overhead" PI.

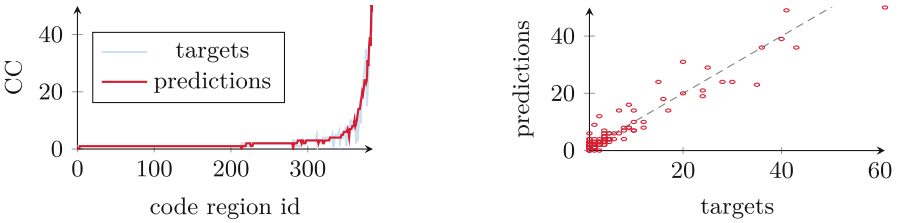
"high overhead, branch functions" to a 1% random selection of the data set[5]. The plots clearly show that the relationship between the pre- and post-PI metrics cannot be easily modelled since it is a complex non-linear multivariate transformation.

5 Experimental Results

We tested our ANNs to estimate the potency of protected versions of the Sumatra application (Sect. 3). We used Diablo 2.82, GCC 4.9.2 and MathWorks MATLAB R2017a on an Intel i7-4980HQ CPU @ 2.80 GHz with 16 GiB RAM under Debian GNU/Linux testing with kernel 4.9.0.

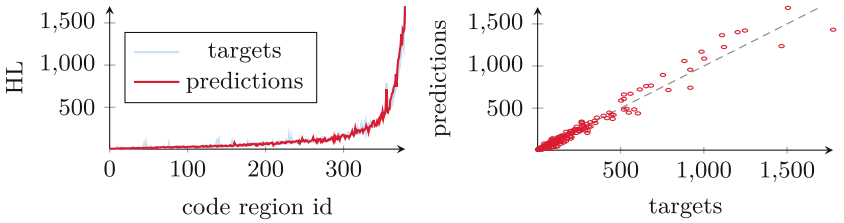
We randomly selected 10% of our data set as the test set (3551 samples) and the remaining observations formed the training set (31959 samples). Then, we trained 36 feed-forward neural networks (6 PIs × 6 metrics) by using a multi-loop approach to iteratively try different parameters (i.e. training and activation functions, number of neurons) and find the ideal architectural structure. We selected the parameters that minimized the Root Mean Square Error (RMSE), which was computed using a k-fold cross-validation approach with $k = 10$ where we averaged the errors of each validation fold [11].

[5] For the sake of readability, we limited the y-axis to about one quarter of the maximum metric value in Figs. 2, 3, and 4.



(a) Error plot. (b) Targets-predictions scatter plot.

Fig. 3. CC predictions for "branch functions, high overhead" PI.



(a) Error plot. (b) Targets-predictions scatter plot.

Fig. 4. HL predictions for "branch functions, high overhead" PI.

Table 1. Neural networks performance.

PI	CC			HL		
	RMSE	MAE	R^2 [%]	RMSE	MAE	R^2 [%]
Branch functions, low overhead	2.4	0.7	97.3	160.5	32.9	97.8
Branch functions, high overhead	3.2	0.9	96.9	142.6	33.9	97.8
Function flattening, low overhead	3.2	0.7	96.1	86.4	24.8	99.1
Function flattening, high overhead	4.5	0.9	92.8	74.2	26.4	98.7
Opaque predicates, low overhead	2.6	0.7	97.6	110.1	37.8	98.2
Opaque predicates, high overhead	3.3	0.8	96.4	94.9	34.5	98.8

Each network receives as inputs six metrics (i.e. cyclomatic complexity, Halstead length, the number of input/output operands, instructions and edges in the control flow graph). These values are normalized in the $[-1, +1]$ range and then processed by the ANNs, having one hidden layer with three neurons. We used the Levenberg-Marquardt back-propagation training with early stopping to avoid overfitting (the `trainlm` function). The hidden layer activation function is the hyperbolic tangent sigmoid, while the single neuron output layer function is a simple linear transfer (the `tansig` and `purelin` functions).

Table 1 reports the RMSE, the Mean Absolute Error (MAE) and the coefficient of determination (R^2) of the ANNs used to predict the cyclomatic complexity (CC) and the Halstead length (HL). The average cyclomatic complexity in our

Table 2. Test application results.

Priority	Length 1			Length ≤ 2			Length ≤ 3		
	High	Low	All	High	Low	All	High	Low	All
Compilation time [s]	156.1			949.6			5535.9		
Simulation time [s]	1.6			9.2			31.6		
Accuracy [%]	100.0	77.8	89.5	50.0	66.7	57.9	12.1	37.0	21.9
Proximity [%]	100.0	87.0	93.9	84.6	75.4	80.2	64.2	70.4	67.2

data set is 5.1 with a maximum value of 187 and all our CC networks show a MAE less than one. On the other hand, the average HL is 206.2 with a peak at 6887 and our HL networks have a MAE of about 30. The coefficient of determination R^2 is well beyond 90%, proving that our ANNs accurately predict metric changes.

Figures 3a and 4a plot the targets (i.e. actual) and predicted metrics (random selection of the 10% of the test set). The ANN predictions easily follows the real data, grasping the general trend of the metrics. Note that the Halstead length curves are very close together, so that the ANN is also able to model several spikes in the graph. Figures 3b and 4b sketch the scatter plots of the predicted-target ratios. Each point represents a sample, the closer the point to the dashed line the better. The cyclomatic complexity prediction shows more dispersion, but the overall trend is still close the ideal line.

Assessing how the estimated metrics can be used to select the best PI sequence for each asset is more complex, as this prediction requires to serially connect multiple ANNs, thus propagating the errors. For the 10 high priority Sumatra assets we use the high overhead PIs, and for the 9 low priority ones the low overhead PIs. We fixed the maximum length of the PI sequences to $l = 3$, giving us a grand total of 760 sequences. We computed the combined potency of each sequence (Eq. 1) using both the real and predicted metrics, as shown in Table 2, which also reports the compilation and the simulation times (the time to estimate all the metrics with our ANNs). As expected, the time exponentially increases as the length of PI sequences grows. However, actually applying the protections required 1.5 h, while the prediction was completed in 31.6 s.

To assess how good are our predictions, we introduced the *accuracy*, a percentage that reports how often the estimated potency allowed the selection of the same protections using the measured potency. For sequences of length 1, we yield nearly a perfect score, which decreases as the depth increases. We also introduced the notion of *proximity*, as accuracy does not report how close a solution is to the optimum. The function $\text{ORD}(\sigma_i, a_i)$ returns the position of σ_i in the list of PIs for the asset a_i sorted in descending order of measured potency. Let's take one PI sequence for each asset, $\Pi = (\sigma_1, \ldots, \sigma_{n_a})$, the proximity of Π with maximum length l and with n_p PIs is defined as:

$$\text{PROX}(\Pi) = \frac{1}{n_a} \sum_{i=1}^{n_a} \frac{\text{COMB}(l, n_p) - \text{ORD}(\sigma_i, a_i)}{\text{COMB}(l, n_p) - 1}$$

A proximity of 100% indicates that the optimum was reached and a 0% means that the worst solution was chosen. At length 1, the proximity of our solution is 93.9%, very close to the optimality. When the length increases to 2 and 3, it decreases to 80.2% and 67.2%, still being relatively close to the best value.

6 Related Works

To our knowledge, the only work that leverages ANNs for software security assessment, by identifying security flaws in software design, is a paper of Adebiyi et al. [1]. They manually converted 715 attack scenarios gathered from various vulnerability databases into regular expressed attack patterns [9], and used the latter to train a back-propagation neural network to classify the attacks.

On the other hand, several works used ANN to assess the security of computer networks. Liu et al. proposed a message security scheme to encrypt messages with a Real-time Recurrent Neural Network-based (RRNN) cipher [14]. Fu et al. leveraged a back-propagation ANN to assess the security of wireless networks against risk assessment models defined by the authors [8]. Turčaník designed a packet filter that uses ANNs to greatly reduce the time needed to perform the filtering [18]. Finally, several intrusion detection systems makes use of ANNs to classify access logs (e.g. firewall logs) in normal and anomalous [13,17].

7 Conclusions and Future Work

This paper presented an approach to estimate the potency of obfuscation techniques with neural networks that are able to accurately predict the value of several software metrics. To improve the precision of the technique and achieve a higher lever of accuracy, we will enlarge the data set used to train the ANNs, allowing them to be more accurate on a single protection. Moreover, we will train ANNs that predict changes created by sequences of protections.

In addition, we are also considering other obfuscators and protection techniques that alter software metrics. Finally, by also taking into account the dynamic metrics, we will exploit machine learning to predict protection overheads introduced on execution performance, memory allocation and network usage.

References

1. Adebiyi, A., Arreymbi, J., Imafidon, C.: Applicability of neural networks to software security. In: 14th International Conference on Computer Modelling and Simulation, pp. 19–24 (2012)
2. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K.: On the (im)possibility of obfuscating programs. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 1–18. Springer, Heidelberg (2001). doi:10.1007/3-540-44647-8_1

3. Ceccato, M., Penta, M.D., Nagra, J., Falcarin, P., Ricca, F., Torchiano, M., Tonella, P.: The effectiveness of source code obfuscation: an experimental assessment. In: IEEE 17th International Conference on Program Comprehension, pp. 178–187 (2009)
4. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Trans. Softw. Eng. **20**(6), 476–493 (1994)
5. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Technical report, University of Auckland, July 1997
6. Collberg, C., Thomborson, C., Low, D.: Manufacturing cheap, resilient, and stealthy opaque constructs. In: 25th ACM SIGPLAN Symposium on Principles of Programming Languages, pp. 184–196 (1998)
7. Collberg, C.S., Thomborson, C.: Watermarking, tamper-proofing, and obfuscation-tools for software protection. IEEE Trans. Softw. Eng. **28**(8), 735–746 (2002)
8. Fu, J., Huang, L., Yao, Y.: Application of BP neural network in wireless network security evaluation. In: 2010 IEEE International Conference on Wireless Communications, Networking and Information Security, pp. 592–596 (2010)
9. Gegick, M., Williams, L.: On the design of more secure software-intensive systems by use of attack patterns. Inf. Softw. Technol. **49**(4), 381–397 (2007)
10. Halstead, M.H.: Elements of Software Science. Operating and Programming Systems Series. Elsevier Science Inc., New York (1977)
11. Kohavi, R.: A study of cross-validation and bootstrap for accuracy estimation and model selection. In: 14th International Joint Conference on Artificial Intelligence, pp. 1137–1143 (1995)
12. Linn, C., Debray, S.: Obfuscation of executable code to improve resistance to static disassembly. In: 10th ACM conference on Computer and Communications Security, pp. 290–299 (2003)
13. Lippmann, R.P., Cunningham, R.K.: Improving intrusion detection performance using keyword selection and neural networks. Comput. Netw. **34**(4), 597–603 (2000)
14. Liu, C.Y., Woungang, I., Chao, H.C., Dhurandher, S.K., Chi, T.Y., Obaidat, M.S.: Message security in multi-path ad hoc networks using a neural network-based cipher. In: 2011 IEEE Global Telecommunications Conference, pp. 1–5 (2011)
15. Low, D.: Protecting Java code via code obfuscation. Crossroads - Spec. Issue Robot. **4**(3), 21–23 (1998)
16. McCabe, T.J.: A complexity measure. IEEE Trans. Softw. Eng. **2**(4), 308–320 (1976)
17. Mukkamala, S., Janoski, G., Sung, A.: Intrusion detection using neural networks and support vector machines. In: Proceedings of the 2002 International Joint Conference on Neural Networks, vol. 2, pp. 1702–1707 (2002)
18. Turčaník, M.: Packet filtering by artificial neural network. In: 2015 International Conference on Military Technologies, pp. 1–4 (2015)
19. Udupa, S.K., Debray, S.K., Madou, M.: Deobfuscation: reverse engineering obfuscated code. In: 12th Working Conference on Reverse Engineering, pp. 45–54 (2005)
20. Van Put, L., Chanet, D., De Bus, B., De Sutter, B., De Bosschere, K.: Diablo: a reliable, retargetable and extensible link-time rewriting framework. In: 5th IEEE International Symposium on Signal Processing and Information Technology, pp. 7–12 (2005)
21. Viticchié, A., Regano, L., Torchiano, M., Basile, C., Ceccato, M., Tonella, P., Tiella, R.: Assessment of source code obfuscation techniques. In: IEEE 16th International Working Conference on Source Code Analysis and Manipulation, pp. 11–20 (2016)
22. Wang, C., Davidson, J., Hill, J., Knight, J.: Protection of software-based survivability mechanisms. In: 2001 International Conference on Dependable Systems and Networks, pp. 193–202 (2001)