A New Code Obfuscation Scheme for Software Protection

Conference Paper · April 2014

DOI: 10.1109/SOSE.2014.57

CITATIONS
11

READS
456

2 authors:

Aniket Dilip Kulkarni
Tata Consultancy Services Limited
7 PUBLICATIONS **22** CITATIONS
SEE PROFILE

Ravindra Metta
Tata Consultancy Services Limited
16 PUBLICATIONS **51** CITATIONS
SEE PROFILE

A New Code Obfuscation Scheme for Software Protection

Aniket Kulkarni
TCS Research, Tata Consultancy Services,
Plot 54 B, Hadapsar Industrial Estate,
Pune - 411013, India
Email: aniket.kulkarni@tcs.com

Ravindra Metta
TCS Research, Tata Consultancy Services,
Plot 54 B, Hadapsar Industrial Estate,
Pune - 411013, India
Email: r.metta@tcs.com

***Abstract*—IT industry loses tens of billions of dollars annually from security attacks such as tampering and malicious reverse engineering. Code obfuscation techniques counter such attacks by transforming code into patterns that resist the attacks. None of the current code obfuscation techniques satisfy all the obfuscation effectiveness criteria such as resistance to reverse engineering attacks and state space increase. To address this, we introduce new code patterns that we call nontrivial code clones and propose a new obfuscation scheme that combines nontrivial clones with existing obfuscation techniques to satisfy all the effectiveness criteria. The nontrivial code clones need to be constructed manually, thus adding to the development cost. This cost can be limited by cloning only the code fragments that need protection and by reusing the clones across projects. This makes it worthwhile considering the security risks. In this paper, we present our scheme and illustrate it with a toy example.**

I. Introduction

The commercial value of software piracy was $58.8 billion in 2010 [17]. Use of popular languages such as Java increases an attacker's ability to steal intellectual property (IP), as the source program is translated to an intermediate format retaining most of the information (such as meaningful variable names) present in source code. An attacker can easily reconstruct source code from such intermediate formats to extract sensitive information such as proprietary algorithms present in the software. Hence, there is a need for development of techniques and schemes to obfuscate sensitive parts of software to protect it from reverse engineering attacks.

Code obfuscation refers to a class of techniques (described by Collberg et al. [1]) that transform a source program P into a target program P′ such that P and P′ have the same "observable behavior" and P′ is difficult to reverse engineer by an attacker. Formally, according to [1], *in order for an obfuscating transformation from P to P′ to be a legal obfuscating transformation, the following conditions must hold:*

- *If P fails to terminate or P terminates with an error condition, then P′ may or may not terminate*
- *Otherwise, P′ must terminate and P′ must produce same output as that of P*

This typically done using certain transformation patterns. A simple example of such a transformation to replace meaningful identifier names in the original code such as *creditCardNumber, computeInterest*, with meaningless random names such as *a, b*. For a taxonomy of obfuscation techniques and the patterns they employ, please refer to [1].

While it has been shown to be impossible to build an obfuscator that can obfuscate all programs [3], there are several specialized obfuscation techniques and tools that are useful in practical settings [4], [6], [20], [7], [8], [18], [19], [9], [10], [16]. The effectiveness of these techniques is measured using the criteria described in Section II viz. potency, resiliency, cost, resistance to reverse engineering attacks and increase in program state space. These criteria help us measure different aspects of any obfuscation technique. To the best of our knowledge, there is no obfuscation technique that satisfies all of these criteria. This means that all existing obfuscation techniques are weak in some aspect or the other. Further, in many software applications, most of the code does not need protection. Only a few sensitive code fragments, such as license checking code and data masking code, need to be protected.

The above observations motivated us to design an obfuscation scheme that protects sensitive code fragments while satisfying all the effectiveness criteria. Our obfuscation scheme is multi-layered. It combines some of the existing obfuscation techniques such that we achieve better strength than each of them. Further, our scheme relies on an idea of *nontrival code clones* to provide better resistance to attacks. These clones require to be constructed manually and thus require additional development effort. However, once created, these clones can be stored in a repository and be reused whenever needed (we already have a working prototype for this). Considering (a) the potential security risks, (b) the higher resistance offered by our scheme and (c) the additional development effort being a one time cost, the advantages of our scheme seem to substantially outweigh the additional cost.

In Section II, we describe all the measurement criteria and some of the obfuscation techniques relevant to our work. We present our scheme and illustrate it with a toy example in Section III. In Section IV, we reason about the effectiveness of our scheme and share the evaluation results on the toy example. Our concluding remarks and future work are in Section V.

II. EFFECTIVENESS CRITERIA

A. Based on Complexity Metrics

Collberg et al. [1] describe the following three measures to evaluate effectiveness of code obfuscation techniques:

- Potency: the degree to which a human reader is confused with obfuscated code.
- Resilience: the degree to which automated de-obfuscating attacks are resisted (section II-B)
- Cost: the performance overhead added to source application due to obfuscation.

The above criteria are related to software complexity metrics such as McCabe's cyclocmatic complexity [12]. Ideally, one would like high potency and high resiliency with low performance overhead.

B. Based on Resistance to Attacks

- Static Analysis Attack: In this, an attacker builds a CFG (Control Flow Graph, a higher level representation of code) of software that helps to comprehend functionality of software. It is obtained by statically analysing code [14].
- Dynamic Analysis Attack: In this, an attacker executes the software on various inputs and studies its output by observing the execution traces [14]. Dynamic analysis is harder than static analysis as the software needs to be run with different inputs.
- Code Clone Detection Attack: In this, an attacker detects and removes code clones present in program to reduce code size of program. Attacker can employ existing code clone detection techniques such as the matching Abstract Syntax Tree (AST) (Baxter et al. [15]) to detect and remove code clones.

An obfuscation technique that increases the analysis efforts of an attacker is said to be resistant to the attack. Higher the analysis effort needed, higher the resistance to reverse engineering attacks as described by Schrittwieser et al. [16].

C. Based on Program State Space

Chow et al. [10] describe obfuscation of control flow of a program by expanding the state space of the program. This is achieved by embedding an instance I of a hard combinatorial problem into the program by applying semantic preserving transformations. This technique expands the state space of the program in such a way that it is hard to compress the expanded state space. In order to detect a desired property of the program, an attacker first needs to find the solution to I by performing static analysis, thus increasing the reverse engineering efforts of an attacker.

D. Existing Obfuscation Techniques

- Identifier Renaming: In this, meaningful names present in the original code are replaced with random meaningless names. It is a one way transformation (as original meaningful names cannot be recovered by attacker) without any cost overhead. Ceccato et al. [5] experimentally show that identifier renaming increases an attacker's efforts needed to perform a successful attack.
- Control Flow Obfuscation Techniques: These techniques transform the control flow of a program to hide the algorithms used in the programs by introducing fake control flows [8]. For example, the technique in [7] introduces opaque predicates, boolean valued expressions whose value is known to an obfuscator, but it is computationally hard for an automatic de-obfuscator to deduce the outcome. The obfuscation technique described by Wang et al. in [9] flattens the control flow and introduces new variables into the program to decide which control flow path is to be taken during an execution. The obfuscation scheme described by Schrittwieser et al. in [16] transforms the control flow to depend upon program's input data, thus shifting the attacker's efforts from static analysis to dynamic analysis.

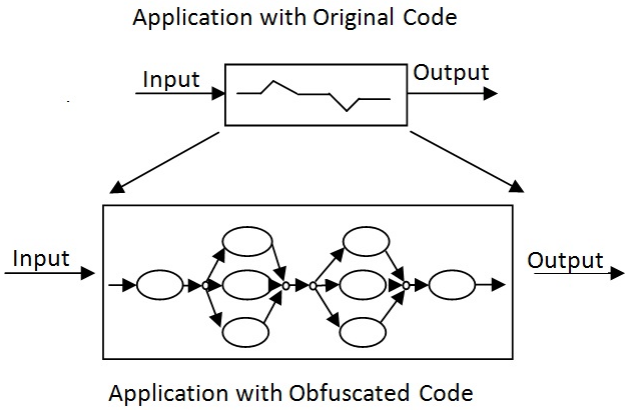

Fig. 1. Our Code Obfuscation Scheme

III. OUR OBFUSCATION SCHEME

In this paper, we focus on the specific problem of protecting sensitive parts of a software such as license checking and data masking mechanisms. In many software applications, such sensitive parts usually consist of a small portion of the entire software. In our internal projects, the size of such components is about a few hundred lines of code.

Given such sensitive parts of a software, we propose a four step approach to obfuscate them. First, divide the sensitive parts into logical code fragments. Second, construct *non-trivial* code clones for each of the fragments. Third, link the clones to the corresponding original fragments using dynamic predicate variables to create valid control flow paths such that one of them is randomly selected at run time. This introduces an exponential number of semantically equivalent paths corresponding to single path present in original code and an attacker then needs to execute and analyze many execution traces to understand the functionality of the sensitive code. Fourth, apply identifier renaming and any other user-specified obfuscation technique(s) to further increase the attacker's efforts to understand obfuscated code. This process is depicted in Figure 1, in which ellipses represents logical code clones constructed for obfuscation. These four steps are explained below.

A. Step 1: Logical Code Fragments

As there is no automatic method to recognise a code fragment as sensitive, we expect the developer to specify code

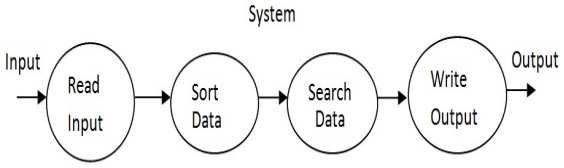

Fig. 2. Data processing application

snippets that are sensitive. Given this specification, we divide the code implementing a sensitive functionality into logical code fragments. We loosely define a logical fragment to be a code fragment that implements specific functionality and is contained within a method. This is because, as we have often seen, each sensitive part of a software is, usually, either a basic block (a block of code with single entry and single exit) or a method.

For an example, let us consider a data processing application. The application reads some inputs, sorts them, searches for some data values and writes output. As depicted in Figure 2, this application can be divided into four logical code fragments "Read Input","Sort Data", "Search Data" and "Write Output". Once such logical code fragments have been identified, we apply the steps below on each of them.

B. Step 2: Nontrivial Code Clones

In this step, we add resistance to static analysis attacks. A clone of a code snippet C is another code snippet C' that is semantically equivalent to C. That is, replacing C with C' in a program P will produce a program P' that produces the same output as P for all the inputs (if P terminates for that input). In paper [1], a code cloning method is described as a technique to increase reverse engineering efforts.

```
Swap1:    int x, y, t;    t = x; x = y; y = t;

Swap2:    int a, b, u;    u = a; a = b; b = u;
```

Fig. 3. Example of trivial code clones

Due to introduction of code clones for obfuscation, complexity of obfuscated code is increased. Baxter et al. [15] describe code clone detection techniques such as matching of AST. These techniques can be used to detect and remove code clones, thus decreasing the reverse engineering efforts. While existing techniques can detect structurally similar clones such as Swap1 and Swap2 (obtained by renaming variables) in Figure 3 above, they cannot detect structurally dissimilar clones such as 'Memory swap' and 'XOR swap' in the code below:

```
1 /* Memory swap */
2 int
3 Swap_1(int x, int y)
4 {
5     int t;
6     t = x;
7     x = y;
8     y = t;
9 }
```

```
1 /* XOR swap */
2 int
3 Swap_2(int x, int y)
4 {
5     int u;
6     u = x ^ y;
7     x = u ^ x;
8     y = u ^ y;
9 }
```

We define such semantically equivalent code clones with different AST as nontrivial code clones. It is very hard to detect these using any of the existing static analysis techniques.

In this step, a developer has to construct the code clones manually. Once the clones are constructed, we propose to store them in a repository together with the information of their semantic equivalence. Next time onwards, given a sensitive code fragment, one can first check if any structurally similar clone exists in the repository. If such a clone is found, its corresponding nontrivial clones in the repository can be used to obfuscate the code fragment. Only when such a clone is not found, one has to manually construct the nontrivial clones.

For example, consider the proprietary license checking software used within companies for licensing. Once one manually builds nontrivial code clones for the license checking software, these clones can be reused for obfuscating license checking code, possibly with minor customization related modifications, across various products. Usually sensitive pieces of such software are small in size and the one time additional development cost of the nontrivial clones is worth it.

C. Step 3: Linking Code Clone Fragments

In this step, we add resistance to dynamic analysis attacks. Once the nontrivial code clones are constructed, we replace the original code fragments with these clones in such a way that one of clones gets randomly selected during each execution of the corresponding original code fragment. If we order the code clones naively for execution, an attacker may figure out the 'logic' of the code-fragment by using dynamic analysis. Predicate variables are boolean valued variables that are proposed in the literature (e.g. Collberg et al. [1]) to help overcome this problem. There are two kinds of predicate variables:

1) **Static opaque predicate variables**: Static opaque predicate variables were introduced by Collberg et al. [1]. An example of static opaque predicate variable is statements "x:=5; b := (x $>$ 10)" in which variable "b" always evaluates to false. Such predicate variables introduce fake control paths in code, as described by Low [8]. Static predicates cannot resist dynamic analysis attacks as fake control paths are never selected during execution of software.
2) **Dynamic opaque predicate variables:** Palsberg et al [11] describe dynamic predicate variables as a set of correlated boolean variables. These variables evaluate to the same value in a given run of the software, but they evaluate to different values in different runs of the software.

Our obfuscation scheme employs a variant of dynamic predicate variables. We use a set of conditions which are evaluated at run time such that the number of combinations of values of the predicate variables is exponential. These variables enable selection of a particular combination of code clone fragments for a given run of obfuscated software. They introduce valid control flow paths by keeping dynamic structures (such as linked list) as shown in Figure 4. This increases static and dynamic analysis efforts as an attacker has to understand the exponential number of code clone combinations with no perceivable pattern.

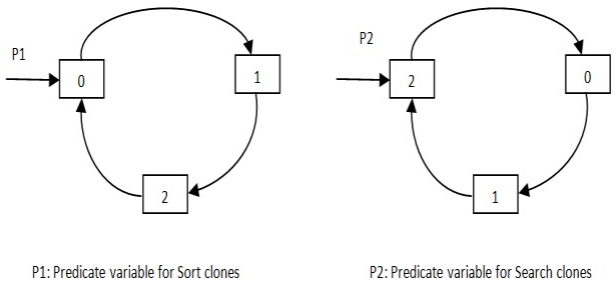

Fig. 4. Sort and search: dynamic predicates

For example, suppose we need to obfuscate the $sort()$ and $search()$ functions of our data processing example. Suppose both the functions have three clones each. Let our dynamic predicate variables be P1 and P2 for $sort()$ and $search()$ respectively. During each run of the software, P1 and P2 take the values $\{0, 1, 2\}$. If P1 is zero during a run, then the first clone of $sort()$ is selected for execution. Similarly if P1 is 1 then the second clone of $sort()$is selected and if P1 is 2 then the third clone of $sort()$ is selected. Similarly for P2.

The way we achieve different values for P1 and P2 is by maintaining two linked lists - one for $sort()$ and one for $search()$ - such that both lists contain the numbers from 0 to 2 in a random order as shown in Figure 4. Variables P1 and P2 move randomly through the lists such that they point to different nodes (containing integer data values) at different times. During each run of software, clones for $sort()$ and $search()$ are selected by matching the data value pointed to by P1 and P2 respectively.

Suppose we have (a) k fragments F_1 to F_k of a sensitive software fragment and (b) each F_i has C_i clones. Then we may create k linked lists L_1 to L_k such each L_i contains all the number between 0 and C_i in a random order. We then simply iterate through the lists a random number of times during each run to choose a random list node. And we select the clone corresponding to the number contained in the random list node. Instead of this elaborate arrangement to pick a random number between 0 and $C_i - 1$, if we directly generate a random number between 0 and $C_i - 1$, then it is easier for an attacker to figure out that the choice is being made randomly and guess that the choices encode the same functionality. Out scheme increases the attacker's efforts by completely randomizing the selection of the clones during each run.

Figure 5 shows the data processing system after linking code clone fragments using these dynamic predicate variables. Notice that any sequence of logical code clone fragments is possible for a given run of the software. Therefore, an attacker needs to generate many execution traces for understanding functionality of the software.

D. Step 4: Your Obfuscation on top

After linking the clones using the dynamic predicate variables as in Step 3, we now propose the use of any or all of the other standard obfuscation schemes that one would like. For example, the identifier renaming technique (see Section II-D) can be applied to logical code clone fragments as it has little or no cost overhead. Figure 6 shows obfuscated data

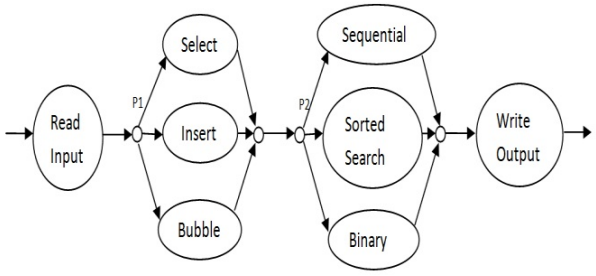

Fig. 5. Linked Data Processing Application

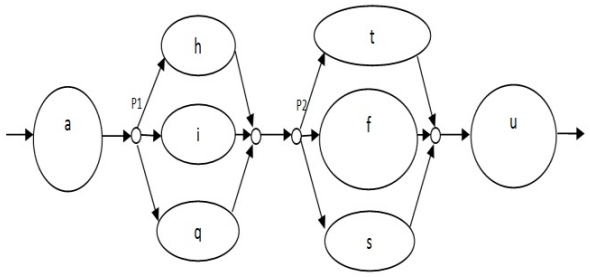

Fig. 6. Renamed Data Processing Application

processing system after renaming the identifiers. Of course, on top of identifier renaming, one may choose to apply state of the art code obfuscation techniques such as control flow transformations [1], [10]. Thus our scheme is complementary to the existing obfuscation techniques.

IV. EVALUATION OF OUR SCHEME

In this section, we first reason about the effectiveness of our scheme. We have also implemented a working prototype for automating our scheme and experimented on the Data Processing Application example of Figure 2. We share the results of this experimentation in Section IV-D

A. Potency, Resiliency, Cost

Anckaert et al. [13] evaluate code obfuscation transformations on software complexity metrics for code, control flow, data and data flow. If the values of these metrics are higher, then the complexity of the program is higher too, thus making it more difficult to understand and reverse engineer the program.

Potency: the degree to which a human reader gets confused is high as (a) the clones are nontrivial, and (b) many valid control flow paths that are confusing to understand due to their invocation through the dynamic predicates and (c) the identifier names are also obfuscated.

Resiliency: As the nontrivial code clones are structurally dissimilar to the original code, none of the existing existing reverse engineering techniques can identify the clones. Further, both the nontrivial code clones and the dynamic predicate variables increase the software complexity metrics of the obfuscated code. Thus, our scheme is more resilient than the existing techniques.

Cost: the performance overhead of our obfuscated code depends on the performance overhead of the nontrivial clones.

We are yet to assess the development and performance costs of the nontrivial clones on real world examples. In general, more the number of code clones introduced for obfuscation, higher the software complexity of obfuscated code, but higher the development efforts to construct the clones. This development effort of nontrivial code clones can be controlled by developers. When the clones get reused across projects, this cost would be lower as it would be amortized across the projects.

Also, we expect our obfuscation scheme to lead to negligible loss of performance because:

- identifier renaming technique has no cost over head [1], and
- code clones are linked using dynamic predicate variables. Our construction of dynamic predicated variables is similar to the construction described by Collberg et al. [7], which was shown to be computationally inexpensive.

B. Resistance to Attacks

To the best of our knowledge, our technique is the only one that resists static, dynamic and clone detection attacks. The structural dissimilarity of the clones together with dynamic predicate variables increases the complexity of the obfuscated code's control and data flow. Thus static analysis attacks are resisted. The reader may refer to [13] for more details on this.

Our obfuscation scheme introduces exponential number of valid control flow paths in obfuscated code. If there are N logical code fragments and K clones per fragment, then the obfuscated code would have K^N paths corresponding to a single path in original code. Thus, an attacker needs to execute software many times to obtain traces to comprehend the functionality, thus resisting dynamic analysis attacks. Lastly, the structural dissimilarity of the clones resists static clone detection techniques and the dynamic predicate variables resist the dynamic clone detection techniques. To the best of our knowledge, ours is the first technique to achieve this.

C. Program State Space

State space of a program obfuscated with our technique is higher due to the insertion of code clones and the predicate variables. As nontrivial code clones have different AST, it is not possible to identify them using existing code clone detection techniques, which are based on AST matching (Baxter et al. [15]). Thus, it is difficult for an attacker to minimize state space by removing nontrivial code clones used for obfuscation.

D. Evaluation of the Toy Example

We have implemented our obfuscation scheme in Java for Java programs. We then explained our scheme to two of our developers and asked them to develop both trivial and nontrivial code clones for the $sortData()$ and $searchData()$ routines of a Java implementation of the Data Processing Application example of Figure 2. They have developed 3 non-trivial clones and 25 trivial clones within four hours. Using these clones, our tool automatically obfuscated the $sortData()$ and $searchData()$ routines. Then, with the help of certain open source tools, we analyzed the effectiveness of our obfuscation. We present the details of our analysis in the rest of this section.

Metric	sortData()			searchData()		
	Ori	TOb	NTOb	Ori	TOb	NTOb
LOC	23	432	67	9	181	50
10KB Data	7 s	7 s	8 s	1 s	1 s	1 s
25KB Data	110 s	110 s	115 s	2 s	2 s	2 s
50KB Data	540 s	542	553 s	7 s	7 s	7s

TABLE I. COMPARISON OF ORIGINAL AND OBFUSCATED CODE

In Table I, sortData() and searchData() are the sort and search functions of the data processing application of Figure 2. The row 'LOC' stands for Lines of Code and the last 3 rows show the execution times with the input for the Data Processing Application ranging from 10KB to 50KB of size. Columns Ori, TOb and NTOb correspond respectively to the original program, the program obfuscated with trivial clones and the program obfuscated with nontrivial code clones. This helps us measure the third criteria of Section II-A viz. performance overhead due to obfuscation. The results show that there is no significant performance loss for the data processing application indicating that our scheme satisfies the *Cost criterion*. The execution times in Table I are measured on a 32-bit Windows Vista machine with 2 GB RAM and a 2.1 GHz dual core processor.

In addition to the performance, we have also measured three other parameters as shown in Table II. In this table, *Cyclomoatic* shows the Cyclomatic complexity of the original program and the program obfuscated with the nontrivial code clones. Cyclomatic complexity [12] measures the number of linearly independent paths. We measured this using the PMD tool [23]. Notice that the Cyclomatic complexity of the obfuscated program is significantly higher than the Cyclomatic complexity of the original program. This indicates that the obfuscated program has high resilience and hence that our scheme satisfies the *Resilience* criterion. Further, the increased Cyclomatic complexity and LOC and renamed identifiers pose significant challenge for even a human reader of the obfuscated code. Thus our scheme satisfies the *Potency* criterion.

In Table II, the row Coverage denotes the basic block coverage. We measured the coverage using the EMMA tool [22]. The results indicate that all the basic blocks in the obfuscated code got executed, which means that all the clones were executed during some iteration of the other. This means that the obfuscated program contains all valid control flow paths. As the number of valid control flow paths are exponential in terms of code clones and logical code fragments, our obfuscation scheme resists dynamic analysis efforts.

We have also measured the memory usage of the original and obfuscated programs. As indicated by the 'Memory' row in Table II, our scheme incurs no significant memory overhead.

Lastly, in order to find if the clones may be automatically detected, we ran the tools in [14], [21], [23] on the code obfuscated with trivial clones and the code obfuscated with the nontrivial clones. All these tools successfully detected the trivial code clones. And, they did not detect any of the nontrivial clones. This shows that it is indeed more difficult for an attacker to detect the non trivial code clones.

The above results are only on a toy example and hence may

Metric	sortData()		searchData()	
	Ori	NTOb	Ori	NTOb
Cyclomatic	5	19	3	16
Coverage	100%	100%	100%	100%
Memory	4MB	4MB	4MB	4MB%

TABLE II. MEASUREMENTS OF OTHER PARAMETERS

not instill much confidence. However, they are inline with the expected results discussed in Sections IV-A, IV-B and IV-C. To be more confident, we are planning to try our scheme on two widely used in-house tools: a license checker and a TCS data masking tool for large data bases of financial firms.

V. CONCLUSIONS AND FUTURE WORK

In this paper, we presented an obfuscation scheme to protect sensitive code fragments of software. We informally showed that code obfuscated by our scheme satisfies the effectiveness criteria described in the literature, modulo the quality of the manual code clone patterns. Although the scheme requires additional construction cost for the clones, it seems useful to obfuscate sensitive code fragments such as data masking and license checking. Lastly, while we have illustrated the scheme for obfuscation of Java programs, but the scheme is applicable to software written in any imperative language such as *C*.

We are planning to experiment our scheme on large industry code. We are currently developing a framework for automating our scheme viz. implement the repository for code clones, automate the linking of clones using the dynamic predicate variables, and provide plugin support for applying other obfuscation techniques on top of ours. While a working prototype is ready, it needs more implementation support. Once implemented, this would enable us to do experimentation to understand the practical issues involved in applying our scheme, which forms our immediate future work.

REFERENCES

[1] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations", Technical report 148, Department of computer science, the University of Auckland, New Zealand, 1997.

[2] P.C. van Oorschot, "Revisiting Software Protection", Proc. 6th Int'l Conf. Information Security (ISC 03),LNCS 2851, Springer-Verlag, 2003, pp. 1-13;

[3] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. "On the (im)possibility of obfuscating programs." In J. Kilian, editor, Advances in Cryptology: CRYPTO 2001, 2001. LNCS 2139.

[4] B. Lynn, M. Prabhakaran, and A. Sahai. "Positive results and techniques for obfuscation", In Eurocrypt, 2004. Springer Verlag.

[5] M. Ceccato, M. DiPenta, J. Nagra, P. Falcarin, F.Ricca, M. Torchiano, and P. Tonella. "The effectiveness of source code obfuscation: an experimental assessment", In IEEE International Conference on Program Comprehension (ICPC 2009). IEEE CS Press, 2009

[6] Larry D-Anna, Brian Matt, Andrew Reisse, Tom Van Vleck, Steve Schwab, and Patrick LeBlanc. "Self-protecting mobile agents obfuscation report Final report", Technical Report 03-015, Network Associates Laboratories, June 2003.

[7] C. Collberg, C. Thomborson, and D. Low. "Manufacturing cheap, resilient, and stealthy opaque constructs", In ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL98, San Diego, January 1998.

[8] Low, D. (1998). "Java Control Flow Obfuscation", Master's thesis. University of Auckland.

[9] Wang, C., Hill, J., Knight, J.C., and Davidson, J.W.: "Protection of software-based survivability mechanisms", In Proceedings of the 2001 conference on Dependable Systems and Networks. IEEE Computer Society. Pages 193-202. 2001.

[10] Chow, S., Gu, Y., Johnson, H., and Zakharov, V.A.: "An Approach to the Obfuscation of Control-Flow of Sequential Computer Programs", In the proceedings of 4th International Conference on Information Security, LNCS Volume 2200. Pages 144-155. Springer-Verlag. Malaga, Spain. 2001.

[11] Palsberg, J., Krishnaswamy, S., Kwon, M., Ma, D., Shao, Q., and Zhang, Y.: "Experience with software watermarking", In Proceedings of 16th IEEE Annual Computer Security Applications Conference. IEEE Press. p308. New Orleans, LA, USA. 2000.

[12] T. J. McCabe. "A complexity measure", IEEE Transactions on Software Engineering, 2(4):308-320, December 1976.

[13] B. Anckaert, M. Madou, B. D. Sutter, B. D. Bus, K. D. Bosschere, and B. Preneel. "Program obfuscation: a quantitative approach", In QoP '07: Proc. of the 2007 ACM Workshop on Quality of protection, pages 15-20, New York, NY, USA,2007. ACM.

[14] M. Madou, B. Anckaert, B. D. Sutter, and D. B. Koen. "Hybrid static-dynamic attacks against software protection mechanisms", In Proceedings of the 5th ACM Workshop on Digital Rights Management. ACM, 2005.

[15] I. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In Proceedings of ICSM. IEEE, 1998.

[16] S. Schrittwieser and S. Katzenbeisser, "Code Obfuscation against Static and Dynamic Reverse Engineering", Vienna University of Technology, Austria, Darmstadt University of Technology, Germany

[17] Business Software Alliance (May 2011), Eighth Annual BSA and IDC Global Software Piracy Study.

[18] A. Balakrishnan and C. Schulze,"Code Obfuscation: Literature Survey", Technical report, Computer Science Department, University of Wisconsin, Madison, USA, 2005.

[19] DashO - PreEmptive Solutions. http://www.preemptive.com/products/dasho

[20] ProGuard: Java obfuscator. http://proguard.sourceforge.net

[21] JCCD - Java Code Clone Detection API. http://jccd.sourceforge.net

[22] EMMA: a free Java code coverage tool. http://emma.sourceforge.net

[23] PMD: Java source code analyzer. http://pmd.sourceforge.net