International Journal of Security and Its Applications
Vol. 10, No. 8 (2016), pp.315-324
http://dx.doi.org/10.14257/ijsia.2016.10.8.27

Indistinguishable Executable Code Generation Method

Mikhail Styugin

Siberian Federal University, Krasnoyarsk, Russia
stygin@gmail.com

Abstract

The classical objective of obfuscation considers indistinguishability of the obtained code in relation to original programs of equal functionality. The present paper reviews another objective of obfuscation, when indistinguishability is considered relatively to programs of different functionality. In this case, an obfuscator is provided with a key at the input. It is nearly impossible to discover the program's functionality without having the key. Hence, that obfuscator type is named a key obfuscator. The paper proves existence of a key obfuscator and existence of effective algorithms for its operation demonstrated by recursive functions. The practical relevance of the result obtained by the research is the possibility to store an executable code in an untrusted computational environment and to obstruct injection attacks and distributed computing analysis. RKB-Obfuscator (Recursive Key Blur), an application for obfuscating high-level applications is presented. The presented application matches recursive functions with commands of a high-level programming language.

Keywords*: indistinguishability; untrusted computations; key-based obfuscation*

1. Introduction

The classical problem of obtaining obfuscator *O* was based on the condition that a program provided as input for obfuscator P and the output program *O*(*P*) have equal functionality and the new generated program *O*(*P*) is unintelligible in some sense [1]. The output program is a black box and no information about it can be obtained except for what can be discovered from the iterative interaction with *P*. In 2001, a team of researchers demonstrated [2] that existence of an obfuscator that complies with those requirements is impossible. A weaker, but a possible definition of an obfuscator was presented in paper [2] in order to maintain the cryptographic problem of obfuscation. In that case, the objective of the obfuscator was that it should not be distinguishable, which one of the two algorithms had been obfuscated within the accuracy to a negligible function when the two algorithms are compared. This indistinguishability condition defines that for every probability polynomial-time algorithm *A* there is a negligible function α, that is such that for all programs C_1 and C_2, being of equal functionality and equal size *k*, the following condition holds:

$$\left|\Pr(A(O(C_1)) - \Pr(A(O(C_2))\right| \leq \alpha(k).$$

That is, no polynomial algorithm can distinguish what was the input of the undistinguishable obfuscator within the accuracy to a negligible value. That objective is similar to the objective of ciphertext indistinguishability in classical cryptography [3]. In 2013 paper [4] was the first to present the solution for creating an obfuscator indistinguishable for all polynomial complexity algorithms. It enabled solving the previously unsolvable problem of functional encryption [5]. Some of those solutions are presented in papers [6] and [7]. The idea of the obfuscator was created by implementing complex mathematical problems, such as the discrete logarithm problem in finite fields. The solution was called Multilinear Jigsaw Puzzles.

ISSN: 1738-9976 IJSIA

This paper considers the idea of obfuscation from another viewpoint and demonstrates that an indistinguishability requirement for it can be formulated and proved. Let us consider an obfuscator that not only generates output programs having indistinguishable code, but also, a code of unknown functionality. Assume that the program code consists of multiple separate executable modules. Depending on the input parameters, every module uses data in a particular way and transfers control to the next module. The system's output will always be different depending on the input parameters. Here is the system shown in Figure 1.

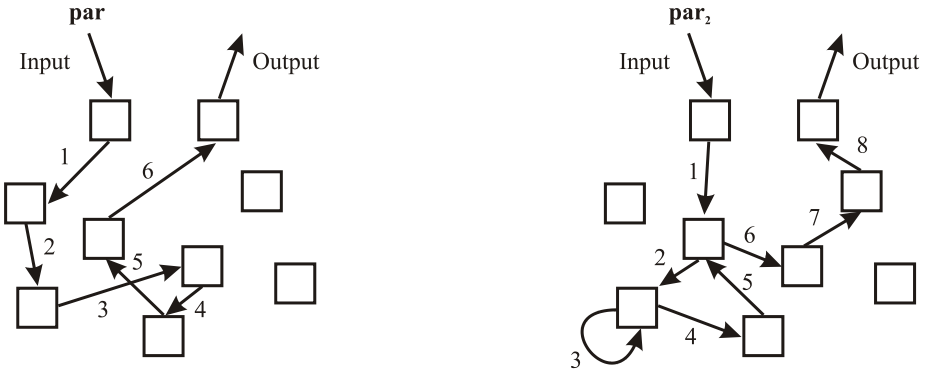

Figure 1. The Program with Different Functionalities Depending on Input Parameters

The program code, which consists of multiple executable modules, is performed differently depending on input parameters. For example, given some input parameters, AES encryption algorithm is executed, and given other input parameters, GOST algorithm is executed. A legal system user is provided with the required input parameters for correct operation of the application. However, the program remains "indistinguishable" in a sense that its functionality cannot be revealed by analyzing its code.

The present paper formulates the obfuscation problem, which enables obtaining a code of indistinguishable functionality. Possibility of creating such obfuscator and its practical implementation in efficient algorithms are presented.

2. Setting the Objective for Key-Based Obfuscation

The result that we aim to obtain does not conform neither to the objective of classical [1] nor indistinguishable obfuscation [2]. We make the assumption that obfuscator $ikO(P, k)$, in addition to program P, has a key k as input, and it will be referred to as *key obfuscator*. A key is a mandatory parameter of output. Having the key in our disposal, the task is either about a regular obfuscator $O(P)$ or indistinguishable obfuscator $iO(P)$.

Obfuscated program $ikO(P, k)$ is indistinguishable from the original algorithm of P. The condition for indistinguishability may be defined as follows:

Given two effectively computable algorithms P_0 and P_1 and attacker A (the distinguishing algorithm). The definition of a probabilistic experiment $\text{Priv}_{A,ikO}(n)$ is introduced. Program $ikO(P_i, k)$ is generated for every randomly selected $i \in \{0, 1\}$ and a random k of bit length n. Program $ikO(P_i, k)$ is provided to A as input. Attacker A is a probabilistic polynomial-time algorithm, and it generates $i' \in \{0,1\}$ as output. When $i = i'$ the experiment is considered to be successful $\text{Priv}_{A,ikO}(n) = 1$, otherwise, $\text{Priv}_{A,ikO}(n) = 0$.

Therefore, the condition of indistinguishability for a key-based obfuscator may be defined as follows:

Indistinguishability. For every two effective algorithms P_0 and P_1 that are of equal length but have *different* functionality and for any probabilistic polynomial-time algorithm A holds the below condition

$$\Pr\left(Priv_{A,ikO}(n)\right) \leq \frac{1}{2} + \varepsilon(n),$$

where, $\varepsilon(n)$ is a negligible function.

A little explanation should be provided. We changed the problem setting so that after obfuscation the algorithm executed by the program is unintelligible. Given some program *P* before obfuscation and after implementing a key-based obfuscator program $P' \leftarrow ikO(P, k)$ is generated. In that case, the initial program *P* had a set of input parameters **par,** that is, *P*(**par**). Hence, the output program *P'* will also have key *k* as a mandatory parameter, that is, *P'*(*k*, **par**). Different P'_k(**par**) algorithms will be generated in the output when different keys denoted by *k* are used. The set of all algorithms of P'_k(**par**) at all possible values of *k* shall be denoted as $\mathbf{P_k}$. Provided that the key is effectively used by the obfuscator, then

$$|\mathbf{P_k}| \approx |\mathbf{k}| = 2^{\|k\|},$$

where the double bars denote bit length.

It is easy to prove that when algorithm space $\mathbf{P_k}$ is significantly less than key space $2^{\|k\|}$, then the condition of indistinguishability is not fulfilled. Thus, when a key length is 128 bit then the output program generated by obfuscator *ikO*(*P, k*) will execute around 2^{128} algorithms given different keys in output.

Another important conclusion can be drawn from the conditions of indistinguishability:

Conclusion 1. For any probabilistic polynomial-time algorithm *A* (an attacker), the defined problem of the possibility to reveal whether program $P' \leftarrow ikO(\ldots,k)$ executes a specific algorithm P cannot be solved by implementing efficient algorithms. That means that complexity of such proving algorithms will grow nonpolynomially to $\|k\|$.

Further presentation of the research material will concern two important questions:

Is existence of a key-based obfuscator possible?

and

Are there effective algorithms for establishing a key-based obfuscator?

In case existence of a key-based obfuscator is possible and effective algorithms for establishing it can be found then many practical problems can be solved:

- Storage of program code on untrusted websites. As direct analysis of an obfuscated program code does not provide information about the algorithms which it can execute, hence, it does not need protection from analysis. That makes a fundamental difference between the key-based obfuscator from the classical obfuscator, as for the latter, obfuscator functionality of an obfuscated program is always known.
- The code does not need to be encrypted. Code encryption is used to protect it from injection attacks. However, in order to run the code, it has to be decrypted to get executable files, which may be compromised. The key-based obfuscator output program is already executable, and it does not require decryption. Analysis of its functionality can be performed when the program is running, but it is more difficult and will be described below.
- The program can be split into many parts and some of the parts will be redundant (they will not be used or will be used in other obfuscated algorithms). However, for the attacker that employs program code analysis there is no opportunity to find out which parts are used in execution of particular algorithms). Provided that the above

parts of applications can be physically separated, it will significantly complicate cracking for intruders.

The problem of key-based obfuscator's existence is reviewed below.

3. Inefficient Algorithms for Creating a Key-Based Obfuscator

In order to prove existence of a key-based obfuscator the below function shall be analyzed

```
Func(k, par)
    x = hash(k)
    if (x == (x0)): P0(par)
    if (x == (x1)): P1(par)
    …
    if (x == (x2^n-1)): P2^n-1(par)
end;
```

The above function *hash*(.) denotes a one-way function.

Theorem 1. Obfuscator *ikO*(*P*(**par**), *k*) →*Func*(*k*, **par**) is a key-based obfuscator.

Assume that one of the two programs $P_{t_0}, P_{t_1} \in \{P_0, P_1, \dots, P_{2^n-1}\}$ is fed to the obfuscator as input.

For defining which of the two functions P_t (where $t \in \{t_0, t_1\}$ was obfuscated, it is necessary to find and verify the existence of the corresponding key that holds $hash(k) = x_t$. As hash function is a one-way function, hence, according to the definition in [3] for every probabilistic polynomial inversion algorithm *A* there is a negligible function *negl*(*n*) being such that:

$$\Pr\left[A\left(f(x)\right) \in f^{-1}\left(f(x)\right)\right] \le negl(n).$$

As a consequence the success of the experiment for distinguishing the obfuscator's input programs is evaluated by the following function:

$$\Pr\left(Priv_{A,ikO}(n)\right) \le \frac{1}{2} + \Pr\left[A\left(f(x)\right) \in f^{-1}\left(f(x)\right)\right] \cdot q(n),$$

where, *q*(*n*) denotes the number of algorithm executions. In accordance with the properties of negligible functions $negl(n) \cdot poly(n) = negl(n)$. Therefore, the probability of a successful experiment is expressed as

$$\Pr\left(Priv_{A,ikO}(n)\right) \le \frac{1}{2} + negl(n),$$

which conforms to the indistinguishability condition. ■

The above example is a proof for existence of key-based obfuscators. However, program code at obfuscator's output grows nonpolinomially to the key length. Hence, the algorithm itself is inefficient.

With no claim for universality of conclusions, assume that the output program of the key-based obfuscator may not grow exponentially to the key length when the function is used recursively.

Consider the following example


```
Func_rb(k, par)
    if (hash(k) == x)
         result = par
    else
         func_rb(k||c(hash(k) mod t), f(hash(k) mod t)(par))
end;
```

A set of factors $\{c_0, c_1, \ldots, c_{t-1}\}$ and a set of functions $\{f_0, f_1, \ldots, f_{t-1}\}$ are used. Therefore, the size of the obfuscator's output program is defined with an optional value *t*. Operation "||" denotes concatenation of two lines.

For example, when factor $t = 4$, then in case 64 program cycles of the function are executed recursively, thus, 2^{128} different operation algorithms with different output values will be obtained.

Theorem 2. Obfuscator *ikO*(*P*(**par**), *k*) →Func_rb(*k*, **par**) is a key-based obfuscator.

In order to prove the above statement, we shall consider the fact that execution of a recursive function can be either endless or it can end with a condition (*hash*(*k*) == *x*). Therefore, we need to define an algorithm that can find such *k*, at which the program execution would end. For that purpose, solutions to the below equation need to be found:

$$k||c_{i_1}||c_{i_2}||\ldots||c_{i_n} = hash^{-1}(x).$$

$$\begin{cases} k||c_{i_1}||c_{i_2}||\ldots||c_{i_n} = hash^{-1}(x) \\ hash(k) \bmod t = i_1 \\ hash(k||c_{i_1}) \bmod t = i_2 \\ \ldots \\ hash(k||c_{i_1}||c_{i_2}||\ldots||c_{i_{n-1}}) \bmod t = i_n \end{cases}$$

Here $i_1, \ldots, i_n \in \{0, \ldots, t\text{-}1\}$. In case execution of a particular algorithm *P*(**par**) needs to be checked, then a system of equations with additional condition has to be solved:

$$P(\mathbf{par}) = f_{i_n}(f_{i_{n-1}}(\ldots f_{i_1}(\mathbf{par})\ldots)).$$

It is evident that a mandatory component of a solution to any of those problems is to calculate the value of $k||c_{i_1}||c_{i_2}||\ldots||c_{i_n} = hash^{-1}(x)$, which also implies finding a polynomial algorithm for inverse calculation of the hash function. As it was defined previously, for all probability polynomial algorithms the probability of its computation will be less than or equal to the negligible function $negl(n)$. Therefore, the probability of a successful experiment $\text{Priv}_{A,ikO}(n)$ is

$$\Pr\left(Priv_{A,ikO}(n)\right) \le \frac{1}{2} + negl(n),$$

which complies to the indistinguishability condition. ■

Recursive function *Func_rb*(*k*, **par**) satisfies several important conditions:

- efficient computability;
- linear size proportional to variable *t*;
- ability to contain any program algorithm represented as $P(\mathbf{par}) = f_{i_n}(f_{i_{n-1}}(\ldots f_{i_1}(\mathbf{par})\ldots))$;

- based on the analysis of the function's code it is impossible to establish whether the computations stop at some key value (there are no polynomial algorithms for that);
- based on the analysis of the function's code it is impossible to establish whether the program is able to execute algorithm $P(\mathbf{par})$ at some key value (there are no polynomial algorithms for that).

Those results appear to be interesting with concern to the obfuscation problems that were set forth in the beginning of the paper. Therefore, the answer to the question of whether it is possible to get an effective algorithm for obfuscator *ikO*(*P*(**par**), *k*) →*Func_rb*(*k*, **par**), that runs in polynomial time to key length *n*.

4. Efficient Algorithms for Creating a Key-Based Obfuscator

We considered the possible ways to create a key-based obfuscator *ikO*(*P*(**par**), *k*) →*Func_rb*(*k*, **par**). Given that several basic functions $\{f_0 \dots f_{t-1}\}$ are defined and those functions are used by the recursive function *func_rb* for repeated creation of the obfuscated algorithm $P(\mathbf{par}) = f_{i_n}(f_{i_{n-1}}(\dots f_{i_1}(\mathbf{par}) \dots))$. The number of nested functions (calls of recursive function for the function itself) in that case shall be such that the function space would not be less than the key space. The number of nested functions shall be denoted as *h*. Then the minimum value of $h = 2n/t$.

Theorem 3. Complexity of algorithm *ikO*(*P*(**par**), *k*) →*Func_rb*(*k*, **par**) with a fixed value of *t* and a specified expansion of function $P(\mathbf{par}) = f_{i_n}(f_{i_{n-1}}(\dots f_{i_1}(\mathbf{par}) \dots))$, where $i_1, \dots, i_n \in \{0, \dots, t\text{-}1\}$ increases nonpolinomially to key length *n*.

For proving the above statement the algorithm logics is reviewed. Given that in the expansion of $P(\mathbf{par}) = f_{i_n}(f_{i_{n-1}}(\dots f_{i_1}(\mathbf{par}) \dots))$ the first function to be executed is $f_{i_1}(\mathbf{par})$. Hence, we need to set such initial key *k* so that the below is fulfilled

$$hash(k) \bmod t = i_1.$$

As the hash function is a one-way function, enumeration of the function's parameters will be required for computation of an inverse value. For each attempt of enumeration

$$\Pr\left[A\left(hash(k)\right) \in hash^{-1}\left(hash(k)\right)\right] = \frac{1}{t},$$

thus, an average of *t* program cycles is required to perform the above operation.

Likewise, when after that $f_{i_2}(\mathbf{par})$ is to be executed the following should be computed

$$hash\left(k||c_{i_1}\right) \bmod t = i_2.$$

In the above case we also get an average number of cycles for completion (finding factor c_{i_1} by enumeration), which is *t*. As operations are executed consecutively the total complexity of the two operations is *O*(.) = 2*t*. Algorithm complexity increases greatly, if the same basic function $P(\mathbf{par}) = f_{i_n}(f_{i_{n-1}}(\dots f_{i_1}(\mathbf{par}) \dots))$ occurs twice and consequently the same factor $c_{i_e} = c_{i_l}$ in the original line for hash function. Then all the values obtained earlier have to be enumerated for every enumerating operation. The number of the duplicating will not be less than (*h*-*t*). Thus, complexity of the whole algorithm can be evaluated with the formula below:

$$O[ikO(P(\mathbf{par}), k) \rightarrow func\_rb(k, \mathbf{par})] = \underbrace{(t + t + \dots + t)}_{t} \cdot \underbrace{t \cdot t \cdot \dots \cdot t}_{h-t} = t^{2+(h-t)} = t^{2\frac{n}{t}-t+2}.$$

As a result it was established that complexity of the obfuscation algorithm for a fixed *t* will increase nonpolinomially to *n*. ■

When there are four basic functions and 128-bit key then the number of algorithm's iterations will not be less than 4^{62}, which is not practicable at the level of computational capabilities at present.

However, it absolutely does not mean that the obfuscator with such algorithm cannot be used. To reduce operations we can increase *t* and reduce the number of algorithm iterations for a fixed *n*. For example, given 16 basic operations for 128-bit key the number of algorithm runs will be only 256. That is to say that the algorithm will be quite simple, given the above parameters.

An important conclusion can be drawn from the above.

Conclusion 2. Algorithm *ikO*(*P*(**par**), *k*) →*Func_rb*(*k*, **par**) for the set function's expansion $P(\mathbf{par}) = f_{i_n}(f_{i_{n-1}}(\dots f_{i_1}(\mathbf{par})\dots))$, где$i_1$, …, $i_n \in \{0, \dots, t\text{-}1\}$ where i_1, …, $i_n \in \{0, \dots, t\text{-}1\}$ will be efficient, provided that factor *t* will be selected depending on the length of key *n*.

Thus, with consideration of selecting factor *t* we obtained an efficient algorithm for the key-based obfuscator.

However, increase in the number of *t*'s in basic functions of recursive function func_rb(*k*, **par**) is not always convenient. Creation of efficient algorithms for the key-based obfuscator with low *t*'s at large key lengths remains the problem of current concern. It may be possible if the initial problem setting is reformulated.

For that we do not define the form of algorithm's representation $P(\mathbf{par}) = f_{i_n}(f_{i_{n-1}}(\dots f_{i_1}(\mathbf{par})\dots))$. In the new problem setting we define a set of factors $\{c_0, c_1, \dots, c_{t\text{-}1}\}$ and an initial code *k*. Thus, computing hash values we set the rigorous execution sequence of basic functions $\{f_0 \dots f_{t-1}\}$. Therefore, the objective is to find the basic functions $\{f_0 \dots f_{t-1}\}$ to fulfil $P(\mathbf{par}) = f_{i_n}(f_{i_{n-1}}(\dots f_{i_1}(\mathbf{par})\dots))$ provided that the recursive execution sequence of basic functions is known.

For processor basic commands, there is always an algorithm for simple generation of sequence $f_{i_n}(f_{i_{n-1}}(\dots f_{i_1}(\mathbf{par})\dots))$. However, obfuscator's operation at the processor's basic command level will cause significant increase of the program code size and slower program code execution. For nontrivial functions (which are not processor basic commands) the above obfuscation is possible, but the problem of a universal algorithm for that kind of obfuscation is still to be solved.

Given that the original program executes function $P(x) = x^2$. The recursive chain of basic functions *Func_rb*(*k*, *x*) is represented as $f_1(f_2(f_2(f_1(x))))$. For $P(x) = f_1(f_2(f_2(f_1(x))))$ we can $f_1(x) = x^{1/2}$and $f_2(x) = x^{8^{1/2}}$. The above recursion of basic functions will function similarly to *P*(*x*).

For all processor's basic commands it can be determined which of them execute the initial command given a certain combination. It can be done with consideration to the fact that for every operation there is a reverse operation that could bring the required result in recursion. However, implementation of a recursive function for every basic command is too time-consuming and will cause significant expansion of the application's executable code.

We will have to determine whether the program needs to be indistinguishable when an adversary can analyze the program when it is running, that is all its keys at the program's input will be visible. Thus, the program can remain indistinguishable in the sense stated in the definition introduced in [2], or only remain unintelligible. In the latter case, the recursive function does not need to be implemented for each basic command, but it can be used for a collection of commands.

Finding basic functions is not such a simple task with a collection of commands or algorithms that are more complex. That problem was partially solved by the RKB-Obfuscator.

5. Practical Implementation of a Key-Based Obfuscator by the Example of the Rkb-Obfuscator

RKB-Obfuscator (Recursive Key Blur Obfuscator) employs a key-based obfuscator algorithm by matching commands of a high-level programming language and recursive functions of *func_rb* type.

Here is an example of a short program.

```
For (i=0; i<10; i++)
    Print("helloworld")
end;
```

The above program executes a cycle and displays the line "hello world" ten times.

A recursive function with the same function is as follows:

```
func_rb(k, i)
    if (hash(k) == x)
          result = i
    else
          print("hello world")
    func_rb(k||c, i = i + 1)
end;
```

Values *k*, *c* and *x* are selected for the recursive function to repeat exactly 10 times. Thus, functionality identical to the former program is provided.

Similar patterns for creating recursive functions can be defined for all high-level commands.

6. Conclusion

The paper formulated requirements to key-based obfuscators and proved its existence and presented efficient algorithms for it. Those algorithms were the foundation for development of the RKB-Obfuscator, which functions as a key-based obfuscator at the level of a high-level programming language.

As to practical implementation, key-based obfuscators enables solving such problems as storage storing an application's code at untrusted locations and splitting a program's logic with no possibility to recover it without the key data. Since direct analysis of the obfuscated program's code does not provide information about the algorithms it can execute, hence, it does not need protection from analysis by a potential adversary.

Acknowledgments

This work was supported by the Grant of President of Russian Federation MK-5025.2016.9 under Governmental Contract No. 14.Y30.16.5025-MK.

References

[1] S. Hada, "Zero-knowledge and code obfuscation", Advances in Cryptology - ASIACRYPT '2000, **(2000)**.
[2] B. Barak, O. Goldreich, R. Impagliazzo, S.Rudich, A.Sahai, S.Vadhan and K. Yang, "On the (Im)possibility of obfuscating programs", 21st Annual International Cryptology Conference, CRYPTO, Santa Barbara, CA; United States, **(2001)** August 19-23.
[3] J. Katz and Y. Lindell, "Introduction to Modern Cryptography", CRC PRESS, New-York, **(2007)**.
[4] S. Garg, M. Raykova, C. Gentry, A. Sahai, S. Haleviand and B. Waters, "Candidate indistinguishability obfuscation and functional encryption for all circuits", Annual IEEE Symposium on Foundations of Computer Science (FOCS), **(2013)**.
[5] J. Alwen, M. Barbosa, P. Farshim, R. Gennaro, S. Gordon, S. Tessaro and D. Wilson, "On the Relationship between Functional Encryption, Obfuscation, and Fully Homomorphic Encryption. Cryptography and Coding", Lecture Notes in Computer Science, vol. 8308, pp. 75-80.
[6] A. Sahai and B. Waters, "How to use indistinguishability obfuscation: deniable encryption, and more", Proceedings of the 46th Annual ACM Symposium on Theory of Computing, **(2014)**.
[7] S. Garg, C. Gentry and S. Halevi, "Candidate Multilinear Maps from Ideal Lattices", Lecture Notes in Computer Science (EUROCRYPT 2013), vol. 7881, **(2013)**, pp. 1-17.

Authors

Mikhail Styugin, He is a senior lecturer at Siberian Federal University and a scientist at Siberian State Aerospace University (Krasnoyarsk, Russia). PhD degree in computer science. Conducts research in the area of information security systems and technologies of information warfare. Owner of two companies that develop solutions in the area of information security systems on the Internet.