M.A. Styugin
Siberian Federal University, Svobodnyy pr., 79, Krasnoyarsk, 660041, Russia, e-mail: styugin@gmail.com, ORCID 0000-0001-9746-2719

The Method of Generation Program Code with Indistinguishable Functionality

Keywords: obfuscation, indistinguishability, untrusted computations, key-based obfuscation, computational complexity, polynomial time algorithm.
The paper presents an obfuscation problem, in which indistinguishability is considered relatively to programs with distinguishable functionality. The functionality remains undisclosed owing to unknown parameters in the program's input. An attacker that can run only polynomial time algorithms will not be able to distinguish the program's functionality without having the input key. Thus, it was named a key obfuscator. The article sets a problem of creating key obfuscators and efficient algorithms for their operation. The obfuscator's general scheme of operation for logic schemes of any functionality is presented. Proof of a theorem is provided, which states that the obfuscator presented is a key obfuscator. The possibility of finding key obfuscator's efficient operation algorithms for specific applications is established. Key-based obfuscation enables finding solutions for tasks such as open storage and execution of a program code in an untrusted computational environment and makes it more difficult to carry out injection attacks and analysis of distributed computing. Authentication and confirmation of rights for using software can be another application of key obfuscation, because without the key program code is useless.

М.А. Стюгин
Сибирский федеральный университет, пр. Свободный, д. 75, Красноярск, 660041, Россия, e-mail: styugin@gmail.com, ORCID 0000-0001-9746-2719

СПОСОБ ПОСТРОЕНИЯ ПРОГРАММНОГО КОДА С НЕРАЗЛИЧИМОЙ ФУНКЦИОНАЛЬНОСТЬЮ

Введение

Формально обфускатор O представляет собой компилятор, основанный на эффективных (имеющих полиномиальную сложность) алгоритмах. Он получает на вход программу (circuit) P и продуцирует новую программу $O(P)$, которая имеет функциональность, аналогичную P, но при этом является в некотором роде неразборчивой (unintelligible) [1]. С этой точки зрения полученный код $O(P)$ представляет собой черный ящик, для которого нельзя получить никакой иной информации, кроме той, что можно получить на основе итеративного взаимодействия (oracleaccess) с P. В работе [2] впервые появилось понятие неразличимого обфускатора, где под неразличимостью принимается тот факт, что не существует полиномиального алгоритма, который может различить, что было на входе у обфускатора с точностью до пренебрежимо малой величины. Постановка такой задачи аналогична постановке задачи неразличимости шифров в классической криптографии [3]. В 2013 г. в работе [4] впервые было представлено решение задачи построения неразличимого обфускатора для любых алгоритмов и имеющего полиномиальную сложность от времени выполнения программы. Данное решение было названо важнейшим открытием в области криптографии за последнее десятилетие. Неразличимая обфускация решает одну из важнейших проблем крипто-

графии – хранение секрета внутри самой программы [5]. Это принципиально меняет многие криптографические подходы, как, например, ассиметричное шифрование, которое можно теперь делать с использованием симметричного ключа, поскольку шифратор может быть опубликован в обфусцированном виде с неизвлекаемым ключом [6–8].

На сегодняшний день неразличимая обфускация кода до сих пор остается больше теоретической задачей, редко реализуемой на практике.

Постановка задачи обфускации с неразличимой функциональностью. Результат, который мы планируем получить, не укладывается в формулировку задачи как классической [1], так и неразличимой обфускации [2]. Мы будем предполагать, что обфускатор $ikO(P, k)$ получает на вход помимо программы P еще и некоторый ключ – k. Далее мы будем называть его ключевой обфускатор.

Обфусцированное приложение $ikO(P, k)$ является неразличимым относительно исходного алгоритма P. Условие неразличимости можно сформулировать следующим образом.

Возьмем два эффективно вычислимых алгоритма P_0 и P_1 и атакующего (алгоритма различения) A. Введем определение вероятностного эксперимента $\mathrm{Priv}_{A,ikO}(n)$. Для любого случайно выбранного $i \in \{0, 1\}$ и случайного k побитовой длины n генерируем программу $ikO(P_i, k)$. Программу $ikO(P_i, k)$ подаем на вход A. Атакующий A является вероятностно-полиномиальным алгоритмом. На выходе он генерирует значение $i' \in \{0, 1\}$. Если $i = i'$, то эксперимент оценивается как удачный $\mathrm{Priv}_{A,ikO}(n) = 1$, в противном случае $\mathrm{Priv}_{A,ikO}(n) = 0$.

Тогда условие неразличимости для ключевого обфускатора можно сформулировать следующим образом.

Неразличимость (Indistinguishability). Для любых двух эффективных алгоритмов P_0 и P_1, принадлежащих некоторому классу $\mathbf{P_k}$, но *различной* функциональности и любого вероятностно полиномиального алгоритма A, выполняется

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

где $\varepsilon(n)$ – пренебрежимо малая величина (negligiblefunction).

Мы изменили постановку задачи таким образом, что после обфускации остается неясным, какой алгоритм выполняет программа. Допустим до обфускации была некоторая программа P. После применения ключевого обфускатора мы получили программу $P' \leftarrow ikO(P, k)$. При этом, если исходная программа P имела множество входных параметров **par**, т.е. $P(\mathbf{par})$, то программа на выходе P' будет иметь еще обязательным параметром ключ – k, т.е. $P'(k, \mathbf{par})$. При различных ключах k на входе мы будем получать различный алгоритм – $P'_k(\mathbf{par})$. Обозначим как $\mathbf{P_k}$ множество всех алгоритмов $P'_k(\mathbf{par})$ при всех возможных значениях ключа k. Если обфускатор эффективно использует ключ, то

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

где двойная скобка обозначает побитовую длину.

Данная исследовательская работа ориентирована на решение вопроса: *существует ли универсальный ключевой обфускатор для любых программных алгоритмов*?

Схема ключевого обфускатора. Для исследования возможности построения ключевого обфускатора воспользуемся базовыми криптографическими алгоритмами. Одним из таких базовых элементов криптографии являются однонаправленные функции. Если мы возьмем какую-либо константу, валидную длине выхода однонаправлен-

ной функции, то нам будет затруднительно ответить на вопрос о том, какое значение на входе однонаправленной функции может дать такое значение на выходе, и существует ли вообще подходящее значение входа.

Таким образом, мы можем заключить, что однонаправленные функции являются подходящим криптографическим примитивом для построения ключевого обфускатора. Если каждому из огромного числа значений выхода однонаправленной функции сопоставить алгоритм, то в результате функциональность программы с неизвестным значением входа остается также неизвестной (с точностью до пренебрежимо малых величин). Принцип функционирования такой программы показан на рис. 1.

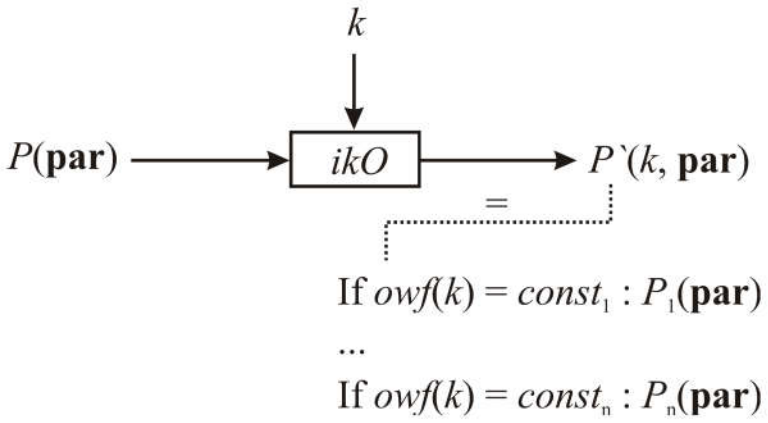

Рис. 1. Ключевой обфускатор с экспоненциально растущей длиной программы на выходе

Здесь и далее под аббревиатурой owf будем подразумевать однонаправленную функцию (one-wayfunction).

В результате, мы пришли к выводу что обфускатор, генерирующий на выходе множество алгоритмов (в количестве сравнимом с |**k**|) и осуществляющий их выбор на исполнение на основе однонаправленных функций, является *ключевым обфускатором*.

Однако очевидно, что объем программного кода на выходе такого обфускатора будет расти неполиномиально быстро от длины ключа *k*, поскольку в этом случае у нас $n = 2^{\|k\|}$. То есть количество условных операторов на выходе программы (как показано на рис. 1) будет расти экспоненциально к длине ключа *k*.

Рассмотрим другой пример обфускатора, где программный код на выходе будет генерироваться в процессе исполнения программы. Сделать это можно путем использования повторяющихся (рекурсивных функций). Однако результаты работы подобных функций будут всегда различны. Пример такого исполняемого кода приведен в листинге.

```
func_rb(k, par)
    if (owf(k) == x)
        result = par
    else
      func_rb(k||c(hash(k) mod t),f'(f(hash(k) mod t), par))
end;
```

Здесь функция f′ имеет вид:

```
f′(f, par)                    //    |par| = n
    I = ∅, i = 0
    While i≠ (n+1) do
        if (owf(k || i) mod 2) : I += {i}
        i += 1
    end while
    res = par + {f(par_i1, …, par_im)}
end;
```

Таким образом функция f′ нужна для того, чтобы выполнять передаваемые ей функции $f_{(hash(k) \bmod t)}$ с произвольным набором параметров. А сами функции $f_{(hash(k) \bmod t)}$, общее количество которых равно t, являются базовыми функциями, реализующими функциональность всей программы. То есть конечная программа должна быть собрана из базовых функций как из «кирпичиков». Совокупность таких базовых функций должна позволять разложить на нее функциональность исходной программы.

В результате мы можем прийти к следующей теореме.

Теорема 1. Обфускатор $ikO(P(\mathbf{par}), k) \rightarrow$func_rb($k$, **par**) является ключевым обфускатором для программ с любыми логическими схемами (circuits).

Доказательство данной теоремы достаточно длинное. Здесь мы приведем только общую логику его построения.

Первая часть доказательства рассматривает необходимый набор базовых функций. Как известно, любая логическая функция (схема) может быть построена с использованием ограниченного набора логических элементов, называющихся универсальным набором. В частности такими элементами являются операторы AND, OR и NOT [9]. В результате мы можем определить набор базовых функций обфускатора с использованием данных операторов.

Таким образом, мы получим три базовые функции:

$f_{AND}(par_1, \ldots, par_n) = par_1$ AND … AND par_n
$f_{OR}(par_1, \ldots, par_n) = par_1$ OR … OR par_n
$f_{NOT}(par_1, \ldots, par_n) =$ NOT par_1

С использованием данных операторов мы можем представить любую функциональную схему, подаваемую на вход.

Вторая часть доказательства определяет возможность получения нужной функциональной схемы с использованием ключа на входе. Очевидным является тот факт, что, используя тотальный перебор входных параметров ключа, мы будем получать различные логические схемы, длина которых ограниченна длинной ключа на входе. Таким образом, если искомая функциональная схема не была найдена на нужной длине ключа, то она будет найдена на ключе большей длины.

Несмотря на то, что мы нашли ключевой обфускатор, работающий на любых логических схемах, алгоритм его работы (приведенный в доказательстве) не является эффективным, поскольку использует тотальный перебор ключа на входе. Вторым очевидным недостатком его работы является большое количество программного кода на выходе, на исполнение которого потребуется многократно больше вычислительных ресурсов, чем для программы на входе. Однако на вопрос, поставленный в начале данной статьи, нам удалось получить однозначный ответ: *ключевой обфускатор существует для любых функциональных схем.*

Использование ключевого обфускатора. Возможность использования ключевого обфускатора основана на получении эффективных алгоритмов его работы. Данная задача не решена в общем виде, т.е. мы не знаем, существуют ли эффективные алгоритмы работы обфускатора для всех логических схем. Однако в работе [10] нами уже были рассмотрены эффективные алгоритмы работы ключевого обфускатора для частных случаев. Принципиальное различие между решением, приведенным в данной статье, и решением, рассмотренным в [10], заключается в том, что здесь на выходе ключевого обфускатора при переборе ключа мы будем получать все возможные логические схемы ограниченной длины. Для ключевого обфускатора, приведенного в [10], мы ограничиваемся только последовательными функциями (исключая функцию f′, позволяющую распараллеливать логику работы приложения).

Таким образом, говоря о практическом использовании ключевого обфускатора, стоит обратить внимание на определение неразличимости. Здесь мы говорим о функциональной неразличимости в рамках используемого множества $\mathbf{P_k}$. Есть ли необходимость в том, чтобы множество $\mathbf{P_k}$ содержало все возможные логические схемы равные по длине искомой программе? В подавляющем большинстве случаев такой необходимости нет.

Допустим, задача может заключаться в том, чтобы оставался неизвестным алгоритм шифрования, определенный конечным набором блочных шифров и принципов построения блоков замещения и перестановок. Алгоритм получения неразличимого алгоритма шифрования на выходе такого ключевого обфускатора будет эффективным, и для его построения нет необходимости спускаться до логических операторов. В данном случае комбинации построения блоков перестановок и замены будет достаточно, чтобы сгенерировать множество $\mathbf{P_k}$ необходимой мощности, например 2^{256}.

Результаты работы

В данной работе удалось сформулировать требования к ключевому обфускатору и доказать его существование для построения любых логических схем.

С прикладной точки зрения использование ключевых обфускаторов позволяет решать такие проблемы, как хранение кода приложения на недоверенных ресурсах и разделение логики программы с невозможностью восстановления, без знания ключевой информации. Поскольку непосредственный анализ кода обфусцированной программы не дает информации относительно тех алгоритмов, которые он может выполнять, мы можем не защищать его от анализа потенциальным злоумышленником. Сама по себе ключевая обфускация защищает программное обеспечение только от анализа на цифровых носителях, поэтому его можно безопасно хранить в недоверенной среде. Однако в совокупности с методами забывчивых вычислений [11], оно может так же запускаться на исполнения в недоверенной вычислительной среде без опасности анализа логики выполнения алгоритмов.

Еще одним направлением исследования ключевого обфускатора является подтверждение аутентичности программного обеспечения и прав на его использование. Без

знания ключа на входе, программа становится бесполезной. При этом процедуры дизассемблирования и иного реверс-инжиниринга неэффективны. Следовательно, отсутствует необходимость защиты программного обеспечения от копирования и распространения.

Технология ключевой обфускации, рассмотренная в данной статье, так же как и неразличимая обфускация в целом [5], на сегодняшний день являются больше теоретическими, а не прикладными проблемами. До повсеместного использования неразличимых обфускаторов необходимо решить множество прикладных вопросов, которые не позволяют использовать их на практике в связи с многократно растущей сложностью выполнения программного кода после обфускации. Однако, поскольку решения задач обфускации рассматриваются в эффективных алгоритмах (имеющих полиномиальную сложность от размеров исполняемых программ), то это свидетельствует о том, что все эти прикладные вопросы будут разрешены в ближайшем будущем. Как следствие, неразличимая обфускация существенно изменит принципы и стандарты в области криптографии, авторизации и аутентификации.

Работа выполнена при поддержке гранта Президента Российской Федерации МК-5025.2016.9.

СПИСОК ЛИТЕРАТУРЫ:

1. Hada S. Zero-knowledge and code obfuscation. Advances in Cryptology // ASIACRYPT '2000, 2000. P. 443–457.
2. Barak B., Goldreich O., Impagliazzo R. On the (Im)possibility of obfuscating programs // Lecture Notes in Computer Science. 2001.Vol. 2139 LNCS. P. 1–18.
3. Katz J., Lindell Y. Introduction to Modern Cryptography // CRC PRESS, 2007, 512 p.
4. Garg S., Raykova M., Gentry C. Candidate indistinguishability obfuscation and functional encryption for all circuits // Annual IEEE Symposium on Foundations of Computer Science (FOCS 2013). P. 13–34.
5. Garg, S., Gentry, C., Halevi, S., Raykova, M., Sahai, A., Waters, B. Hiding secrets in software: A cryptographic approach to program obfuscation // Communications of the ACM. 2016. Vol. 59. №. 5. P. 113–120.
6. Arriaga, A., Barbosa, M., Farshim, P. Private functional encryption: Indistinguishability-based definitions and constructions from obfuscation // Lecture Notes in Computer Science [17-th International Conference on Cryptology in India, INDOCRYPT 2016]. 2016. Vol. 10095 LNCS. P. 227–247.
7. Agrawal, S., Agrawal, S., Badrinarayanan, S., Kumarasubramanian, A., Prabhakaran, M., Sahai, A. On the practical security of inner product functional encryption (2015) PKC 2015 // LNCS, 9020. P. 777–798.
8. Ananth, P., Brakerski, Z., Segev, G., Vaikuntanathan, V. From selective to adaptive security in functional encryption, CRYPTO 2015 // LNCS, 9216, 2015. P. 657–677.
9. Büning H.K., Lettmann T. Propositional logic: deduction and algorithms // Cambridge University Press.ISBN 978-0-521-63017-7.
10. Styugin M. Indistinguishable Executable Code Generation Method // International Journal of Security and Its Applications. 2016. Vol. 10. №. 8. P. 315–324.
11. Maas M., Love E., Stefanov E., Tiwari M., Shi M., Asanovic M., Kubiatowicz J., Song D. PHANTOM: practical oblivious computation in a secure processor // Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. 2013. P. 311–324.

REFERENCES:

1. Hada S. Zero-knowledge and code obfuscation. Advances in Cryptology // ASIACRYPT '2000, 2000. P. 443–457.
2. Barak B., Goldreich O., Impagliazzo R. On the (Im)possibility of obfuscating programs // Lecture Notes in Computer Science. 2001.Vol. 2139 LNCS. P. 1–18.
3. Katz J., Lindell Y. Introduction to Modern Cryptography // CRC PRESS, 2007, 512 p.
4. Garg S., Raykova M., Gentry C. Candidate indistinguishability obfuscation and functional encryption for all circuits // Annual IEEE Symposium on Foundations of Computer Science (FOCS 2013). P. 13–34.
5. Garg, S., Gentry, C., Halevi, S., Raykova, M., Sahai, A., Waters, B. Hiding secrets in software: A cryptographic approach to program obfuscation // Communications of the ACM. 2016. Vol. 59. №. 5. P. 113–120.
6. Arriaga, A., Barbosa, M., Farshim, P. Private functional encryption: Indistinguishability-based definitions and constructions from obfuscation // Lecture Notes in Computer Science [17-th International Conference on Cryptology in India, INDOCRYPT 2016]. 2016. Vol. 10095 LNCS. P. 227–247.
7. Agrawal, S., Agrawal, S., Badrinarayanan, S., Kumarasubramanian, A., Prabhakaran, M., Sahai, A. On the practical security of inner product functional encryption (2015) PKC 2015 // LNCS, 9020. P. 777–798.

8. Ananth, P., Brakerski, Z., Segev, G., Vaikuntanathan, V. From selective to adaptive security in functional encryption, CRYPTO 2015 // LNCS, 9216, 2015. P. 657–677.
9. Büning H.K., Lettmann T. Propositional logic: deduction and algorithms // Cambridge University Press.ISBN 978-0-521-63017-7.
10. Styugin M. Indistinguishable Executable Code Generation Method // International Journal of Security and Its Applications. 2016. Vol. 10. №. 8. P. 315–324.
11. Maas M., Love E., Stefanov E., Tiwari M., Shi M., Asanovic M., Kubiatowicz J., Song D. PHANTOM: practical oblivious computation in a secure processor // Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. 2013. P. 311–324.