Запутывание (обфускация) программ
Обзор

Юрий Лифшиц

15 декабря 2004 г.

Аннотация

В этой работе собраны основные результаты и подходы новой области теоретической информатики - запутывания (обфускации) программ. Дан обзор методов запутывания, способов оценить качество обфускации, различных типов атак на запутанный код. Также приведен ряд конкретных постановок, для которых требуется разработка алгоритмов запутывания. Главная задача обзора – разобраться, какие результаты представляют наибольший интерес и решение каких открытых задач требуется в первую очередь. Удалось сформулировать три наиболее горячих точки: нужна математическая модель запутываения программ, требуются методы запутывания с доказанной секретностью и нужны новые подходы к оценке эффективности запутывающих преобразований.

1 Введение

Традиционная криптография занимается защитой доступа к секретной информации. Достаточно недавно (первые работы - 1997 год) начались исследования в области защиты от несанкционированного использования исходных кодов программ. Как ни странно, оказалось что четко сформулировать, что же такое запутывание программ, довольно сложно. Обфускатором называется преобразователь исходного кода программы в запутанный код. Полученная программа должна остаться исполняемой, причем время ее работы и длина описания не должны существенно вырасти, а семантика (зависимость выходных данных от входных) в точности совпадать с исходной. При этом должно выполняться свойство *секретности*: код новой программы не должен раскрывать информацию об алгоритмах, параметрах и внутренней структуре исходной программы.

В этом определении неясно, что мы понимаем под программой и что, строго говоря, означает секретность. Различные формализации этого определения дают различные подходы к запутыванию программ.

Запутывание программ (обфускация) начиналось, по всей видимости, с разработки практических приложений, выполняющих эту задачу. Развитие новых технологий программирования (таких как виртуальная машина Java и распространение программного

обеспечения в виде байт-кода) дали новую мотивацию для развития методов обфускации. Проводится международное соревнование по написанию коротких запутанных программ – International Obfuscation C Code Contest. В настоящее время существуют десятки коммерческих обфускаторов, однако серьезные фундаментальные исследования еще только начинаются.

Обзор написан с целью получить представление о теме и достичь такого уровня понимания, который позволил бы ставить открытые вопросы. В качестве выводов мы представляем первоочередные задачи для рассматриваемого исследовательского направления. Из предыдущих попыток рассмотреть результаты нескольких работ важно упомянуть обзор [2].

Изложим в нескольких предложениях основные выводы и наблюдения. Последние работы сформировали тендецию разработки защиты против конкретных атак, вместо разработок общего метода. Поэтому в ближайшее время наиболее актуальным станет построение адекватных математических моделей. Эта задача содержит в себе построение нескольких классификаций: представлений программ, мотивов атак на запутанный код, инструментов по распутыванию и уровней предварительных знаний об атакуемой программе и запутывающем преобразовании. Также в каждой модели необходимо будет четко сформулировать, что является успехом атакующего (распутывающего) участника.

Дальше мы выделяем тему доказанной секретности запутывающих преобразований. В статье [10] получен первый результат в этом направлении и мы верим, что оно несет в себе немало ярких и непохожих на текущие запутывающих методов.

Третьим направлением является тестирование запутывающих методов, оценка их надежности и построение системы оценки качества запутывающих преобразований.

2 Приложения (Use cases)

Прежде чем переходить к формальным определениям и моделям, необходимо ясно представлять практические применения запутывания программ.

Исходно, самым первым приложением для запутывания программы была защита важных модулей от их повторного использования в программных продуктах конкурентов. Это приложение мы называем защитой от извлечения модуля.

Следующим важным направлением является защита от внесения несанкционированных изменений в программу. Это направление очень близко к tamper-proofing. Задача у этих направлений общая (защита от изменений), но методы разные: запутывание делает трудным внесение таких изменений, которые дали бы зарание поставленный эффект, а tamper-proofing вводит в программу большое количество самопроверок на неизменность, удалить которые непросто. Скорее всего будущее останется за гибридными техниками. В качестве конкретных примеров можно упомянуть клиентские программы интернет-магазинов, программы, обслуживающие кредитные карты и банковские счета.

Частным случаем предыдущего направления является написание программ с ограниченной функциональностью, запароленным входом или ограниченным сроком действия. Сюда же можно отнести и программы, результаты которых разрешается использовать лишь ограниченным рядом способов (например, нельзя редактировать). Запутывание программ может стать (но пока не стало) серьезным препятствием для несанкционированного снятия встроенных ограничений.

Есть и совершенно другое приложение техник запутывания. Сейчас идут исследования по обфускации вирусов, которая должна помешать антивирусам распознать опасные программы.

Следующим направлением является защита алгоритма. Это важно, скажем, для новых техник стеганографии, где знание метода позволяет быстро построить успешную атаку.

Еще одним ярким примером служит защита от извлечения данных. Одним из интересных примеров является обфускация блочных шифров (block cypher). Суть блочного шифрования заключается в разбиении сообщения на блоки, после чего каждых блок шифруется с помощью ключа специальным (трудно обратимым) алгоритмом. Задача для обфускации заключается в том, чтобы реализовать этот алгоритм с фиксированным ключом таким образом, что анализируя код программы нельзя было бы узнать сам ключ. В статье [4] был предложен метод для решения этой задачи, однако позже в работе [8] был представлен полиномиальный алоритм извлечения ключа из запутанного этим методом кода.

3 Классификация атак

Для того чтобы создать качественные методы запутывания, необходимо ясно представлять, против каких именно действий противника мы хотим защитить код программы. Так как на данный момент нет определения "идеального шифрования", которое могло бы быть реализовано на практике, то идет борьба за "хорошее" шифрование. Таким образом, измерение "качества" обфускации становится очень важным. Чтобы дать оценку запутанности, необходимо иметь классификацию целей, методов и инструментов противника.

В этом разделе (написанным по мотивам [5]) будут изложены игровая модель защиты программ, различные неправомерные действия противника, и общие идеи по защите от этих атак. Защиту программного кода можно рассматривать как игру с двумя участниками - Алисой (A) и Бобом (B). A написала программу P и собирается ее распространять. Программа попадает к B, который, анализируя полученный код, пытается совершить одно из следующих неправомерных действий: 1) внести в программу определенные изменения (tampering) и пользоваться модифицированной программой; 2) растиражировать и перепродать программу другим пользователям (piracy); 3) выдернуть из программы некоторые модули и вставить их в свой конкурирующий продукт (malicious reverse engneering).

Для каждой атаки существует устоявшаяся концепция защитных методов. Техники защиты от изменений называются tamper-proofing и заключаются во внесении в программу постоянных проверок на отсутствие изменений. Более хитрая идея заключается в зашифровке кода и затем (в случае отсутствия изменений) постепенной динамической расшифровке исполняемых модулей. В случае пиратства используется механизм вкрапления водных знаков (watermarking). Суть метода в следующем: легальный продавец A программного продукта модифицирует код, "зашивая" в него свою подпись. Если подпись невозможно заменить на свою, то третье лицо C, которому нечестный покупатель B попытается перепродать программу всегда сможет убедиться в том, что права на продажу принадлежат A, а не B.

Защитой от повторного использования является предмет нашего обзора – запутывание программы (obfuscation). Естественная идея заключается в затруднении анализа кода, так чтобы невозможно было установить смысл отдельно взятого фрагмента кода.

Фактически классификацию методов и инструментов атак еще только предстоит разработать.

4 Модели

Обсудим в деталях различные моменты нашего неформального определения.

Рост потребляемых ресурсов. В теоретической модели допускается полиномиальный рост потребления всех ресурсов (память, время, размер кода). На практике обфускатор выстраивает запутывание в рамках наперед заданных ограничений по ресурсам. Как правило, эти ограничения позволяют программе вырасти лишь в константное число раз. Выполняется следующая закономерность – чем проще метод, тем меньше его цена (требуемое увеличение ресурсов) и тем менее эффективен метод. Самые примитивные методы имеют нулевую цену.

Отдельным вопросом является эффективность самого обфускатора. На практике нет жестких ограничений – ведь запутывать нужно всего один раз. Однако никаких интересных методов, которые требовали бы больших усилий по реализации пока не было предложено. Все запутывающие преобразования довольно легко и быстро реализуемы.

Случайные биты обфускатора. Во время своих запутывающих преобразований обфускатор имеет возможность выбирать параметры этих преобразований и сами преобразования из своего арсенала. Как и в классической криптографии, мы предполагаем, что сам обфускатор известен нашему противнику. Поэтому задача распутывания сводится к угадыванию строки случайных выборов обфускатора. Угадав строку можно убедиться, что эти выборы действительно привели именно к той запутанной версии, которая попала к пользователю. Таким образом (результат [1]), задача восстановления кода лежит в NP. В то же время более общая задача по нахождению минимальной программы эквивалентной данной является неразрешимой.

Разница между программой и функцией. В теоретической модели требуется реали-

зовать вычисление функции так, чтобы какие-то ее скрытые параметры нельзя было восстановить по этой реализации. На практике мы имеем дело с программым кодом, и объектом защиты могут быть не только параметры функции, но и параметры кода. Кроме того ведется запутывание не только автономных программ, но и программ получающих входные данные из нескольких источников.

Запутываемость. Не все программы могут быть запутаны одинаково хорошо. Важный пример – программа, воспроизводящая свой код. При сохранении семантики код исходной программы невозможно скрыть. Таким образом, ввиду отсутствия классификации моделей и атак, встает пока как следует нерешенная задача: определить и измерить степень потенциальной запутываемости. В работе [3] дана верхняя оценка на степень запутываемости и показано, что эта планка достижима не во всех случаях.

Предлагаемый подход. Наиболее продуктивным на данном этапе исследований представляется следующий план: построить как можно больше конкретных троек (программа, защищаемый секрет, возможности противника) близких к реальным приложениям и начать изобретение методов для их запутывания. Зафиксировав базовые примеры мы сможем сравнивать разные методы на одних и тех же задачах. Первый такой пример (запутывание многопарольной проверки) дает работа [10].

5 Основные подходы

Центральная задача теории обфускации заключается в следующем. Нужно придумать алгоритм (написать приложение), которой, получая на вход некоторую программу P, трансформировал бы ее в программу $O(P)$. При этом должны быть выполнены три условия: 1) сохранение семантики – те же выходы на тех же входах; 2) ограничение на рост потребляемых ресурсов – размер кода, память и время исполнения, требуемые запутанной программой по сравнению с исходной; 3) секретность – увеличение сложности программы, скрытие внутренних параметров. При этом на входе и на выходе обфускатора программа может быть записана в разных *представлениях*. Сам процесс преобразования называется обфускацией (запутыванием). Это определение необходимо завершить ответом на вопросы: (1) что мы понимаем под программой и (2) что значит секретность.

В информатике и программировании приняты самые разные модели вычислений и представления программ: машины Тьюринга, булевы схемы, объектно-ориентированные языки, ассемблер, машинный код. На первый вопрос можно дать два ответа. Мы можем считать что P – программа на определенном языке (в определенном представлении), и $O(P)$ записана в том же представлении (языке). В этом случае мы можем разрабатывать методы запутывания конкретного *представления* программы.

Второй вариант ведет к так называемому *архитектурному подходу* – мы разрабатываем специальное представление(архитектуру) программ, которое легко исполняется, но трудно для анализа. В этом случае обфускацией будет просто трансляция программы в это представление (на эту архитектуру).

Третий подход концентрируется на запутывании алгоритма, лежащего в основе программы, не фиксируя конкретного представления. Этот подход во многих случаях неприменим, но, в случае успеха, он дает большую защищенность параметров алгоритма. В этом случае программу мы рассматриваем просто как функцию, преобразующую входные данные в выходные.

Хорошего определения секретности еще не получено. Скорее всего, для каждой модели будут разработаны свои варианты определения.

6 Методы обфускации

6.1 Запутывание кода

На данный момент наиболее распространен подход, заключающийся в применении большого количества простых трюков. Каждый метод в отдельности легко распутывается, но в совокупности они делают задачу анализа кода (особенно статического анализа) достаточно сложной. Такая методология дает два важных преимущества: она универсальна (применима ко все программам) и очень легко программируется. Такие методы направлены на затруднение статического анализа. Основным недостатком является полное отсутствие оценок на качество запутанности. Более того, вряд ли эти методы вообще могут дать какую-то доказуемую секретность. Перечислим наиболее распространенные преобразования этой методологии [7]:

Разделение одной переменной на несколько, хранение в одной переменной нескольких переменных, изменение размерностей массивов (двумерные в одномерные и наоборот), замена одного цикла на два вложенных и наоборот, использование дополнительных указателей и разыменование их, переименование переменных и функций, клонирование, переупорядочивание независимых команд, вставка неиспользуемого кода.

Также разработано некоторое количество менее тривиальных методов. В качестве примера можно привести темные предикаты (opaque predicats). Суть его в следующем: в поток управления добавляются условные операторы, содержащие трудно предсказуемые (темные) предикаты. Одна из веток выхода таких операторов может содержать фальшивый код. Так же возможно использование в двух ветках разных представлений одного и того же кода.

Стоит упомянуть методы запутывания на уровне ассемблера [13, 9] и метод построения плоского потока управления [4].

6.2 Архитектурный подход

Параллельно запутыванию программ изучается и смежный вопрос: как организовать такую среду, в которой можно было бы исполнять программы без опасения, что их внутренняя структура будет изучена и подвергнута изменению или извлечению данных. Мы называем такой подход архитектурным. В настоящее время ведутся активные исследования и уже получено несколько патентов.

6.3 Алгоритмические подходы

Все результаты, рассматривающие защиту программ как защиту внутренних параметров функции, которую вычисляет эта программа, мы относим к алгоритмическому подходу. Явными плюсами является строгость модели и возможность доказывать оценки на полученную секретность. Минусом является тот факт, что приведенные в нашем обзоре практические приложения, часто не укладываются в функциональную модель. Скорее всего алгоритмический подход в ближайшее время будет расширен на другие модели, более адекватно отражающие реальные приложения.

Самой естественной постановкой для алгоритмического подхода является скрытие параметров. Требуется вычислить f_s так, что по коду программы нельзя вычислить параметр s. Сразу заметим, что это возможно только для семейства, в котором по семантике представителя невозможно эффективно вычислить параметр. Но есть и еще одно уточнение: скорее всего злоумышленнику нужен не сам параметр s, а хочется произвести какое-то действие, включающее этот параметр. Таким образом, запутывание программы должно рассматриваться как способ затруднить именно это действие. Как раз к защите параметров относится результат работы [10]. Доказуемая секретность также обсуждается в работе [12].

7 Как измерить запутанность программы?

Есть три подхода к этому вопросу. Первый заключается в том, что мы строго описываем модель, все возможности противника и указываем какие именно действия считаются взломом программы. Вычислительная сложность взлома и будет мерой запутанности.

Второй подход заключается в том, чтобы просто измерить сложность запутанной программы по одной из известных метрик (вроде количества предикатов или уровня вложенности циклов). Можно еще упомянуть подход "тестирование на студентах".

Третий подход (имеет дело только с алгоритмическим подходом), изложенный в работе [3], дает наиболее строгое определение "идеальной" запутанности. Результат $O(P)$ запутывания программы P обладает black-box security, если никакие действия с кодом $O(P)$ не принесут результатов больше, чем простое изучение семантики (input-output поведения) программы P. К сожалению, как доказано в работе [3], обфускация, реализующая black-box security, возможна не для каждой программы.

Запутываемость и изучаемость. Когда зафиксирована конкретная атака, необходимо разобраться, какие именно программы могут быть от нее защищены. В некоторых случаях сохранение семантики ведет к невозможности запутывания. Еще важно уметь выяснять, в каких случаях тестирование запутанной программы позволяет написать эквивалентную, не распутывая запутанный вариант программы. Собственно, цель написать эквивалентную может ставиться только в случае, когда нет доступа ко всей запутанной программе целиком, а, скажем, имеется только оракульный доступ.

Можно предложить следующий (оригинальный) метод. Взять два фиксированных семейства функций, эффективно не отличимых друг от друга (например, две реализа-

ции псевдослучайных генераторов). Метод запутывания будет считаться прошедшим *различительный тест*, если по запутанной реализации $O(P)$ функции одного из двух семейств невозможно эффективно дать правильный ответ с вероятностью не меньше $2/3$, какое семейство представляет $O(P)$.

Важным открытым вопросом является определение по программе: каков, вообще говоря, потенциал данной программы быть запутанной. Более того, в свете последних тенденций (различная защита для разных атак) нужно находить свой потенциал запутываемости против каждой атаки.

8 Смежные задачи и области

Стеганография занимается скрытием секретных данных в большом потоке незначительных данных [11]. Похожие техники могли бы быть полезны и для скрытия внутренних данных программы. Существенной проблемой тут остается динамический анализ, который позволит распутывающему участнику найти все данные, которые будут когда-либо вычислены во время исполнения программы.

Компиляторы занимаются разбором программного кода с целью оптимизации исполняемого кода. Таким образом они производят работу прямо противоположную запутыванию – упрощают программу. Большое количество результатов по анализу кода, терминология и целый ряд представлений программы будут очень полезны для развития запутывания программ.

Для получения доказанной секретности необходимо использовать арсенал, накопленный классической криптографией. Ярким примером могут быть **псевдослучайные генераторы**, про которые доказана их неотличимость от действительно случайных потоков информации. Еще одно важно свойство псевдослучайных генераторов: они являются идеальной моделью программы, которую нельзя изучить (и восстановить), изучая лишь input-output поведение. Еще одним важным понятием классической криптографии является **гомоморфное кодирование**, которое имеет непосредственное отношение к задаче скрытия параметров.

Для более устойчивых запутывающих преобразований, а также для того, чтобы не тратить время на слабые методы, необходимо в полной мере представлять всю мощь **средств и техник**, находящихся на вооружении у **хакеров**.

Для разработки запутывающих преобразований на различных уровнях представления программ и для развития архитектурного подхода необходимо иметь общее представление о вычислительных **архитектурах** и некоторых языках программирования. Особое внимание, в силу легкости декомпиляции, следует обратить на **java byte-code** и **.NET**.

Параллельно с запутыванием развиваются и другие направления защиты программ (software protection). Мы можем упомянуть **защиту от изменений** (tamper-proofing), технику **водяных знаков** (software watermarking) и защиту хостов от загружаемых на них программ.

Нельзя не упомянуть и юридический аспект защиты программ. Центральным понятием этой области является **DRM** - закон по цифровым объектам авторского права.

Благодарности

Благодарю Юрия Владимировича Матиясевича познакомившего меня с этой областью.

Список литературы

[1] A. W. Appel. Deobfuscation is in NP. Unpublished paper, August 2002.

[2] L. D'Anna and B. Matt and A. Reisse and T. Van Vleck and S. Schwab and P. LeBlanc. Self-protecting mobile agents obfuscation report. *Technical Report #03-015, Network Associates Labs*, June 2003.

[3] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan and K. Yang. On the (im)possibility of obfuscating programms, *Advances in Criptology – Crypto 2001*, LNCS 2139, pp. 1-18, Springer-Verlag, 2001.

[4] S. Chow and P. Eisen and H. Johnson and P. C. van Oorschot. White-box cryptography and an aes implementation. *In Proc. 9th International Workshop on Selected Areas in Cryptography (SAC 2002)*, pages 250-270. Springer LNCS 2595, 2003.

[5] C. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection. *IEEE Trans. Software Engineering*, 28(6), June 2002.

[6] C. Collberg and C. Thomborson and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. *In Proc. Symp. Principles of Programming Languages (POPL'98)*, Jan. 1998.

[7] C. Collberg, C. Thomborson and G. M. Townsend. Dynamic graph-based software watermarking. *Technical Report TR04-08*, April 2004.

[8] M. Jacob and D. Boneh and E. Felton. Attacking an obfuscated cipher by injecting faults. *In Proc. 2nd ACM Workshop on Digital Rights Management (DRM 2002)*, pages 16-31. Springer LNCS 2696, 2003.

[9] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. *In Proc. 10th ACM Conference on Computer and Communications Security (ACM CCS 2003)*, pages 290-299.

[10] B. Lynn and M. Prabhakaran and A. Sahai. Positive Results and Techniques for Obfuscation. *In Eurocrypt*, 2004.

[11] F. Petitcolas and R. J. Anderson and M. G. Kuhn. Information hiding – a survey. *Proc. of the IEEE (Special Issue on Protection of Multimedia Content)*, 87(7):1062-1078, July 1999.

[12] N. P. Varnovsky and V. A. Zakharov. On the Possibility of Provably Secure Obfuscating Programs. *In Andrei Ershov Fifth International Conference*, July, 2003, pages 91-102. Springer LNCS 2890, 2003.

[13] G. Wroblewski. General MEthod of PRogram Code Obfuscation. PhD thesis, *Wroclaw University of Technology*, Poland, 2002.