1 Введение

Запутыватель кода (обфускатор) – это средство, которое – во многом, как и оптимизатор кода – неоднократно выполняет преобразования кода программы с сохранением неизменной ее семантики. Однако тогда как оптимизатор пытается сделать программу как можно более быстрой или малой по размеру, обфускатор пытается сделать ее как можно более непонятной. Обфускация обычно применяется к программам для защиты их от обратной инженерии или для защиты секрета, сохраненного в программе, от обнаружения.

В этой статье мы опишем модель *управляющей программы обфускации* (Obfuscation Executive, OE), реализованную в исследовательском средстве защиты ПО SandMark. OE – это общий цикл, который применяет запутывающие алгоритмы к частям защищаемой программы. Во многих случаях функции OE подобны оптимизирующим проходам компилятора: он считывает и анализирует приложение и многократно применяет преобразования кода с сохранением семантики, пока не достигнуто какое-то завершающее условие. В идеале, управляющая программа должна быть способна выбирать оптимальный набор преобразований и оптимальный набор участков программы, подлежащей запутыванию. Единственное необходимое взаимодействие с пользователем – это указать средству, что значит «оптимальный», т.е. как много накладных расходов при выполнении программы может допустить пользователь, как много запутываний он хочет добавить и какие части приложения являются критическими по безопасности или по скорости исполнения. Две главные проблемы в этом процессе: порядок, в котором преобразования будут применены («проблема последовательности стадий»), и как определить, что процесс должен завершиться. В случае оптимизатора кода порядок преобразований, как правило, фиксирован. Оптимизатор обычно завершает работу, когда больше нет внесения изменений в приложение или когда опробовано применение каждого преобразования хотя бы один раз. Как мы далее увидим, в случае с OE определение последовательности стадий и момента завершения не является таким простым.

Оставшаяся часть статьи организована, как изложено ниже. В секции 2 мы описываем общую модель OE. В секции 3 показано, как моделируются зависимости между преобразованиями. В секции 4 рассматриваются детали цикла запутывания. В секции 5 мы обсуждаем другие работы по данной теме, а в секции 6 – подводим итоги.

1.1 Применения обфускации

Начнем с рассмотрения некоторых областей применения запутывания кода. В частности, мы собираемся исследовать ситуации, когда ПО или секрет, скрытый в этом ПО, защищается с помощью обфускации. Также мы увидим, что запутывающие преобразования могут использоваться для облегчения атаки программных продуктов. Рисунок 1 иллюстрирует эти ситуации.

Злонамеренная обратная инженерия: чем больше кода погружено в легко декомпилируемый промежуточный формат, такой как байткод Java или MSIL, тем более озабоченными становятся разработчики в вопросе предотвращения возможности декомпилирования их кода и изучения общей структуры ПО или отдельно взятого ценного алгоритма со стороны конкурентов. Сейчас полагают, что никакой объем запутывающих преобразований не предоставит полной защиты от определенного врага. Скорее, цель – это сделать код Алисы столь запутанным, что Боб найдет повторное использование ее кода не менее легким, чем написание своего с нуля.

Управление цифровыми правами (DRM): DRM – это техника для защиты от медиа-пиратства. Идея состоит в «оборачивании» цифрового содержимого в виртуальный контейнер (*криптопакет*) с бизнес-правами, описывающими, как медиа-данные должны воспроизводиться, продаваться, обмениваться, сдаваться в аренду, и связанными с этими правами ценами. Содержимое может быть использовано только с помощью специального плеера, у которого есть криптографические ключи для разблокировки медиа-данных.

Злоумышленник, способный декомпилировать и проанализировать код плеера, может выявить скрытые ключи, что даст ему возможность наслаждаться медиа-данными бесплатно. По этой причине программные плееры должны быть серьезно заобфускированы и устойчивы к внешним воздействиям.

Злонамеренные мобильные агенты: мобильный торговый агент блуждает по онлайн-магазинам в поисках лучшей сделки для данного товара. Нечестный магазин может манипулировать агентом для создания видимости, что их сделка является самой предпочтительной. Хохл наводит на мысль, что обфускация агента может быть использована для сокращения числа таких атак.

Искусственное многообразие: техника запутывания кода была применена к операционным системам для защиты их от класса атак вредоносного ПО (вирусов и червей). Идея состоит в придании случайного характера коду, чтобы вредоносный агент не мог определить или использовать известную уязвимость. Конечно, вирусы сами с впечатляющим успехом используют технику обфускации для избежания детектирования вирусными сканерами.

Искажающие атаки: программные «отпечатки пальцев» - это уникальные идентификаторы покупателя, встроенные в ПО. Смысл в них в возможности для продавца проследить истоки появления пиратских версий его программы вплоть до самого «пирата», купившего ее. Пират, однако, может запутать программу перед ее продажей с надеждой уничтожения «отпечатков пальцев».

Защита от атак с коллизиями: атакующий может также использовать атаку с коллизиями против программы, защищенной «отпечатками пальцев», путем покупки двух по-разному помеченных копий и сравнения их на предмет поиска местоположения «отпечатка». Для предотвращения подобной атаки продавец должен применить разный набор запутывающих преобразований к каждой продаваемой копии, гарантируя, что сравнение двух копий программы принесет мало информации.

1.2 Алгоритмы обфускации

Большое число *метрик сложности программ* было определено для измерения читабельности, понятности и удобства эксплуатации программных продуктов. Обфускация кода предназначена для максимизации этих метрик сложности при минимизации штрафа потери производительности. Алгоритмы обычно состоят из комбинации следующих операций: **свертка/развертка (fold/flatten)** превращают d-мерную конструкцию в d + 1- или d – 1-мерную; **слияние/разделение (split/merge)** превращают составную конструкцию X в две конструкции {a; b} или две конструкции a и b в составную конструкцию X; **упаковка/распаковка (box/unbox)** добавляют или удаляют уровень абстракции; **добавление ссылок/разыменовывание (ref/deref)** добавляют или удаляют уровень косвенности; **переупорядочивание (reorder)** меняет местами две соседние конструкции; **переименование (rename)** присваивает новое имя уже помеченной конструкции. Здесь *конструкция* указывает на любой объект, с которым может манипулировать обфускатор, для какого-либо языка программирования. Например, если A – это вектор, то **fold**(A) превращает A в двумерный массиав. Если P – это статический метод, то **ref**(P) превращает его в виртуальный. Если B – это базовый блок, то **split**(B) разбивает его на две части путем вставки фальшивой ветви (т.н. *непрозрачный предикат*). Много других подобных преобразований было описано в литературе.

Запутывающие преобразования характеризуются своей *действенностью* (величина путаницы, которую они вносят), *устойчивостью* (размер, до которого они могут быть распутаны *деобфускатором*) и *стоимостью* (штраф потери производительности, вносимой в запутываемую программу).

2 Цикл запутывания

В сердце любого ОЕ лежит цикл, который выбирает часть приложения для запутывания и нужное преобразование из пула алгоритмов-кандидатов, а затем применяет это преобразование. После завершения преобразования, цикл рассчитывает, как велика изменившаяся часть кода, и на основании этого решает, продолжать ли процесс.

К сожалению, есть некоторые осложнения. В первую очередь, ни уровень защищенности, который мы хотели бы достичь, ни размер допустимых накладных расходов не являются неизменными по всему приложению. Некоторые подпрограммы могут быть критическими по производительности, другие – нет. Некоторые подпрограммы могут быть критическими по защите, другие – нет. Во-вторых, для любого данного нетривиального набора запутывающих преобразований будут ограничения на порядок, в котором они должны быть применены. Причина в том, что запутывающее преобразование *уничтожает* структуры в приложении. Это делает обфускированное приложение более сложным для анализа, что в итоге может привести к невозможности дальнейших преобразований. Наконец, не все запутывания могут применяться ко всем объектам приложения. Например, запутывание, переименовывающее классы, не может быть применено к классам, загружаемым динамически по имени. Также, если мы используем запутывания для сокрытия конкретной структуры в приложении (такой как «водяной знак» или набор криптографических ключей), то мы не можем применить преобразования, которые разрушат данные структуры.

2.1 Зависимости преобразований

ОЕ программы SandMark понимает шесть типов зависимостей между запутываниями и использующими их средствами, такими, как программы внедрения «водяных знаков». Это пре-/пост-предложения, пре-/пост-требования и пре-/пост-запрещения:

пост-предложение: предположим, что преобразование по разделению методов *MethodSplit* разделяет метод *computePrimes* на два метода с именами *computePrimes_FirstHalf* и *computePrimes_SecondHalf*. Выбор таких предложенных имен упрощает отладку преобразования. Для обеспечения того, что эти имена не сохранятся в финальной версии обфускированного приложения, *MethodSplit* использует *пост-предложение* для указания необходимости запуска запутывателя имен после разделения.

пре-предложение: предположим, что алгоритм смешивания сигнатур методов *PackFormals* на языке Java преобразует все параметры метода в массив объектов. Например, это будет преобразование метода *foo(Integer a,String b,char c){}* в *foo(Object[] args,char c){}*. Этот алгоритм будет только влиять на параметры-объекты, но не на параметры примитивных типов. Поэтому для того, чтобы сделать преобразование полезным, необходимо до этого алгоритма запустить трансформацию примитивных типов в объекты. Такое преобразование сделает из *foo(Integer a,String b,char c){}* метод *foo(Integer a,String b,Character c){}*, после чего *PackFormals* создаст *foo(Object[] args){}*. Зависимость такого типа названа *пре-предложением.*

пост-требование: предположим, что есть алгоритм внедрения «водяных знаков» (такой как CT), который встраивает «водяной знак» в виде структуры данных в приложение. Для упрощения реализации и отладки программа встраивания «водяных знаков» создает класс *Watermark*, содержащий код для построения «знака»: *Watermark { Watermark left, right; void createGraph() {...} }* Вызов метода *createGraph()* встроен в приложение. Очевидно, что это не слишком скрытно. Поэтому алгоритм встраивания «водяных знаков» требует после себя запуска преобразования по вставке функций в место вызова и запутывания имен. Это зависимость *пост-требования*.

пре-требование: предположим наличие двух запутываний *PublicizeFields* (который делает все поля класса публичными) и *InlineMethod*, производящего внутриклассовую вставку

функций. Поскольку вставленному телу метода не разрешено ссылаться ни на какие приватные поля, то мы сделаем для *InlineMethod пре-требование* метода *PublicizeFields*.
пост-запрещение: предположим, что алгоритм внедрения «водяных знаков» вставляет секретный «водяной знак» в приложение путем изменения частоты появления разных шаблонов инструкций. Стерн представляет такой алгоритм. Любые изменения в теле методов кода с «водяным знаком» могут разрушить этот «знак». Поэтому любой запутывающий алгоритм, который модифицирует тела методов, не следует запускать после внедрения «знака». Этот тип зависимости называется *пост-запрещение*.
пре-запрещение: предположим, что есть запутывающее преобразование *MergeArrays*, осуществляющее анализ альтернативных имен для определения расположения двух массивов для слияния. Этот алгоритм не следует запускать после любого алгоритма, затрудняющего анализ альтернативных имен. Колльберг представляет такие алгоритмы. Такой тип зависимости называется *пре-запрещение*.
пост-предложение/пре-предложение: предложения подобны требованиям в результирующем языке преобразований, которые они допускают. Однако тогда как нарушение требования испортит программу или сделает явным «водяной знак», предложение является лишь подсказкой от автора запутывания для ОЕ, что конкретные преобразования работают хорошо вместе.

В текущей реализации SandMark каждый запутывающий алгоритм и алгоритм внедрения «водяных знаков» точно определяет влияние, оказываемое на код. Он также определяет свойства других алгоритмов, которые пост-требуются, пре-предлагаются и т.д. Например, преобразование по разделению методов *MethodSplit* может регистрировать *OBFUSCATE_METHOD_NAMES* как пост-предложение, указывая ОЕ, что некий алгоритм, который будет выполняться и имеет это свойство, следует запускать после *MethodSplit*. В общем случае,

- для выполнения зависимости-требования один алгоритм с указанным свойством должен быть запущен;
- для выполнения зависимости-предложения один любой алгоритм с указанным свойством может быть запущен; и
- для выполнения зависимости-запрещения никакой алгоритм с указанным свойством не следует разрешать запускать.

В общем случае, простым обфускаторам нет нужды указывать много отношений-зависимостей с другими обфускаторами. Однако, когда обфускация используется с внедрением «водяных знаков», зависимости необходимы для обеспечения того, что запутывания успешно маскируют «водяной знак» без разрушения структур, в которые этот «знак» внедрен. С точки зрения разработки ПО, система зависимостей также позволяет создавать более сложные обфускаторы модульно и без заботы о преобразованиях, совершаемых вокруг них.