УДК 681.3:004.056

А. В. Анисимов, И. Ю. Иванов

ПОДХОДЫ К ЗАЩИТЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ОТ АТАК ЗЛОНАМЕРЕННОГО ХОСТА

Рассматриваются существующие методы защиты ПО от действий злонамеренного хоста, для каждого метода дается оценка его стоимости и обеспечиваемой защиты. Приводится комбинированный метод защиты ПО, представляющий собой усовершенствование двух существующих методов: затемнения, основанного на непроницаемых предикатах, и защиты от внесения изменений в код на основе «забывчивого хэширования». Предложенный метод обеспечивает более высокий уровень защиты по сравнению с базовыми методами, а также является применимым для более широкого класса программ.

Введение

В настоящее время достаточно актуальной является проблема защиты программного обеспечения (ПО). Под *защитой ПО* понимается противодействие *атакам* – злонамеренным действиям, направленным на то, чтобы заставить ПО работать некорректно (возможно, с выгодой для атакующего), либо на раскрытие конфиденциальной информации. Атаки на ПО можно разделить на два класса: атаки *злонамеренного клиента* и атаки *злонамеренного хоста*. В первом случае ПО выполняется на гарантированно доверенном хосте и подвергается атакам множества клиентов, пытающихся воздействовать на него извне через предоставляемый им интерфейс. Угроза атаки злонамеренного хоста имеет место, когда окружение, в котором будет выполняться ПО, на этапе его разработки неизвестно, а значит, последнее потенциально может подвергаться любому воздействию со стороны хоста, а также других программ, выполняющихся на нем. В дальнейшем изложении наряду с использованием термина «ПО» будем использовать термины «программа», «программный продукт», «приложение».

Следует отметить, что в данном случае термин «хост» понимается в более широком смысле, чем «узел в Интернете», и подразумевает некоторую управляющую сущность, в рамках которой выполняется программный код. Хостом может выступать и ядро операционной системы, исполняющее программу, и некая виртуальная машина, интерпретирующая байт-код. Хост инкапсулирует выполнение ПО и, следовательно, имеет полный доступ к его коду и данным.

Следует отметить, что противостоять атакам злонамеренного хоста гораздо сложнее, чем атакам злонамеренного клиента. Некоторую защиту от злонамеренных клиентов может обеспечить ограничение свободы клиента (sandboxing) – разрешение или запрещение клиенту исполнения конкретных команд. Кроме того, злонамеренный клиент не имеет доступа к коду и данным приложения. В случае же выполнения программы на злонамеренном хосте последний может делать с ней все, что угодно (точнее, все, что вычислительно позволимо) – просматривать код, анализировать зависимость выходных данных от входных, трассировать выполнение команд в памяти по шагам. Таким образом, если злонамеренному клиенту приходится иметь дело с «черным ящиком», то злонамеренному хосту программа доступна «на блюдечке».

Так как злонамеренный хост может выполнять любые действия с кодом клиентской программы, то он имеет все возможности для ее анализа. Поэтому говорить об абсолютной защите автономного клиентского приложения в этом контексте нельзя – любая защита, которую инкапсулирует приложение, доступна атакующему и может быть в течение определенного времени проанализирована и снята. Тем не менее, несмотря на то что полной защищенности клиентского приложения достичь невозможно, некоторого уровня защиты все же можно добиться. Приведенный Б. Шнайером [1] принцип гласит,

что защита себя оправдывает, если ее взлом обойдется атакующему дороже, чем создание с нуля продукта, аналогичного защищенному. На сегодня существуют методы обеспечения достаточно сложной для взлома защиты от злонамеренных хостов. Они будут рассмотрены ниже.

Рассмотрим следующие цели:

- проанализировать и классифицировать атаки на ПО со стороны злонамеренного хоста;
- проанализировать существующие методы защиты исходя из стоимости внедрения и обеспечиваемого уровня защиты;
- предложить метод, обеспечивающий защиту от обратного проектирования и модификации кода и данных программы, основываясь на методах, предложенных в последних работах в этой области.

Виды атак на клиентское ПО

Можно выделить три наиболее распространенных вида атак на ПО со стороны злонамеренного хоста: *несанкционированное использование либо распространение ПО* (пиратство); *кража интеллектуальной собственности или конфиденциальных данных*, содержащихся в программе (в том числе восстановление логики работы программы); *незаконная модификация* кода программы.

Пиратство – вид деятельности, связанный с неправомерным распространением или использованием ПО. Существует множество способов проведения таких атак, среди которых (но не ограничиваясь ими) можно выделить следующие:

- неправомерное копирование – перенос программы на другой компьютер и ее выполнение на нем в случае, если это не разрешено лицензией. Следует отметить, что для противодействия этой атаке важно сделать невозможным не столько *копирование* программы, сколько ее *выполнение* на другом компьютере. Этот вид атаки мало распространен в связи с несовместимостью с современной бизнес-моделью распространения ПО;
- неправомерное использование – выполнение программы (либо использование ее результатов) пользователем, которому автор или владелец не предоставил разрешения на выполнение;
- нарушение требований лицензии на ПО;
- перепродажа программного продукта от своего имени.

Незаконное копирование и перепродажа ПО, а также несанкционированное его использование лицами, которые не имеют на это права, ежегодно обходится производителям, по разным оценкам, потерями от 10 до 12 млрд. долларов. По данным [2], 36% всего используемого в мире ПО является пиратским. Можно утверждать, что пиратство –основная проблема для разработчиков и распространителей коммерческого ПО. Это подтверждается и обилием решений для противостояния пиратству. За всю историю коммерческого ПО были придуманы тысячи способов (как программных, так и аппаратных) защиты ПО от нелегального использования и распространения.

Кража интеллектуальной собственности или конфиденциальных данных – это целенаправленный процесс анализа кода ПО с целью извлечения из него определенных функциональных возможностей, а также раскрытия алгоритмов или данных, используемых в программе, которые могут представлять интерес для атакующего. Анализируя извлеченные из программы данные, атакующий (называемый также обратным проектировщиком, или *реверс-инженером*) может получить доступ к алгоритмам программы. Возможно, некоторые из этих алгоритмов защищены патентами или просто должны сохраняться в секрете (например, по причине манипулирования секретными ключами).

Основными инструментами реверс-инженера являются дизассемблер и отладчик (большинство современных реализаций объединяют эти инструменты в одном продукте).

Дизассемблер позволяет по выполняемому коду восстановить исходный код программы в виде инструкций на языке ассемблера, а в некоторых случаях – и в виде программы на языке более высокого уровня (например, C). Отладчик позволяет загрузить программу «внутрь себя» и контролировать ход ее выполнения (выполнять инструкции программы «по шагам», предоставлять доступ к ее адресному пространству, отслеживать обращения к разным участкам памяти). Следует отметить, что реверс-инженер может обойтись и без этих средств, просто рассматривая программу как «черный ящик», подавая ей на вход специальным образом сформированные данные и анализируя выходные данные. Однако анализ по методу «черного ящика» крайне неэффективен ввиду его малой производительности. Поскольку этот метод практически не используется при взломе программ, выполняющихся на злонамеренном хосте, вопрос противодействия ему в данной статье не рассматривается.

Стоить отметить, что обратный анализ программ, написанных на языках, которые компилируются в промежуточный интерпретируемый код (например, Java, C# и другие CLR-языки), на порядок проще программ, написанных на языках, компилирующихся в машинный код, поскольку в исполняемый код таких программ записывается информация про их семантическую структуру (об их классах, полях, методах и т.д.). В связи с возросшей популярностью таких языков программирования (и, в частности, .NET Framework) задача защиты программ, написанных с их использованием, становится все более актуальной.

Третьим видом атак на клиентские приложения со стороны злонамеренного хоста является **модификация кода программы** – преднамеренное или непреднамеренное изменение выполняемого кода программы, приводящее к отклонениям программы от нормального хода выполнения. Например, атакующий может изменить процедуру проверки лицензионного ключа так, чтобы для любого переданного значения она возвращала TRUE, и, таким образом, программа бы считала любое значение лицензионного ключа корректным. Кроме того, атакующий может добавить в программу-жертву код, отсылающий конфиденциальную информацию об окружении, в котором она выполняется, на его компьютер. Способность программы определять, что она была изменена, очень важна, так как изменения, внесенные в программу, могут привести к самым печальным последствиям (например, в случае, если программа обслуживает больницу, электростанцию или другую критическую службу). Отдельно стоит отметить, что подобные атаки могут быть выполнены компьютерными вирусами, заражающими исполняемый код. Если программа сумеет противостоять заражению (или определить, что она была заражена), это сможет существенно снизить темпы распространения вирусной эпидемии.

Следует отметить, что вышеперечисленные виды атак не являются независимыми друг от друга. Скорее, даже наоборот – они очень тесно переплетены между собой. Так, проблема создания генератора ключей (пиратство) опирается на исследование, каким образом в программе выполняется проверка лицензионного ключа (реверс-инженерию); еще один подход софтверных пиратов – удаление из программы кода проверки лицензионного ключа – связан с модификацией кода. Поэтому рассматривать атаки независимо друг от друга не имеет смысла, нужно сконцентрироваться на методах, обеспечивающих максимальное противодействие всем видам атак одновременно.

Методы противодействия атакам на клиентское ПО

Среди множества существующих решений для противодействия вышеописанным атакам можно выделить следующие наиболее распространенные методы защиты ПО:

- затемнение (obfuscation);
- вынесение критического программного кода в отдельный защищенный модуль;
- использование лицензионной метки;
- проверка целостности программного кода;

- создание защищенной среды выполнения.

Основной целью первых двух методов есть противодействие обратному проектированию. Очевидно, чтобы усложнить работу реверс-инженеру, нужно либо «запутать» исходные данные так, чтобы в них было сложно отследить какие-либо зависимости (а следовательно, и разобраться в принципе их работы), либо поместить их на такой носитель, где анализировать их будет невозможно или очень сложно.

Метод на основе лицензионной метки частично решает проблему пиратства. Он основан на присваивании каждому «легальному» пользователю некоего кода, необходимого для корректной работы программы. Наличие этого кода является необходимым условием для запуска программы либо ее полнофункциональной работы.

Одним из необходимых условий безопасности является гарантия целостности программного кода. Допускать выполнение программного кода, в который были внесены изменения, ни в коем случае нельзя ввиду невозможности гарантирования его корректности. Поэтому важнейшей задачей обеспечения как безопасности выполняемого кода, так и его корректности является проверка целостности программы.

Под защищенной средой выполнения понимается некоторое окружение, обладающее свойствами обеспечения защиты исполняемого программного кода от исследования или изменения извне. В качестве примера защищенной среды выполнения можно привести операционную систему, не предоставляющую доступа к адресному пространству памяти программы.

Рассмотрим каждый из методов более подробно.

Затемнение

Затемнение (другой термин – *запутывание*) программного кода является естественным следствием желания обеспечить сохранность чувствительного кода путем максимального сокрытия логики его работы. Очевидно, что с помощью одного запутывания невозможно достичь уровня защиты хотя бы сравнимого с уровнем, который обеспечивают криптографические методы, такие как, например, шифрование публичным ключем. Однако защита программного обеспечения от злонамеренного хоста – область, в которой вся мощь шифрования оказывается полностью беззащитной перед реверс-инженерией. Единственным способом, который мог бы затруднить обратный анализ, является запутывание семантики, сокрытие логики работы программы с тем, чтобы помешать атакующему понять особенности ее работы.

Верхним пределом времени, необходимого для выполнения реверс-инженерии программы P, является время, необходимое для изучения P как черного ящика (т.е. основываясь только на анализе зависимости выходных данных программы от входных), плюс время, необходимое для реализации обнаруженных зависимостей в новой программе. Таким образом, идеальным затемнением будет преобразование, которое по программе P строит такую программу P', что стоимость ее анализа эквивалентна анализу P по методу черного ящика.

Коллберг [3] выделил основные принципы затемнения программного кода. *Затемнить программу* P означает, имея множество *затемняющих преобразований* $T = \{T_1, \ldots, T_N\}$ и программу P, состоящую из объектов исходного кода (классы, методы, функции, операторы) $\{S_1, \ldots, S_K\}$, построить новую программу $P' = \{S_1'=T_{i1}(S_1), \ldots, S_j' = T_{ij}(S_j), \ldots\}$ такую, что выполняются следующие условия:

- поведение P' полностью совпадает с поведением P (преобразования являются семантически замкнутыми);
- P' обеспечивает сокрытие семантики, т.е. анализ и реверс-инженерия P' будет занимать не меньшее время, чем анализ и реверс-инженерия программы P;
- устойчивость каждого преобразования $T_{ik}(S_j)$ является максимальной, т.е. вычислительно сложно построить про-

```
enum CoffeeType {
  Espresso,
  Capuccino,
  Glasse
};

class CoffeeMaker {
  void  PrepareCoffee(CoffeeType  type)
{
  BoilWater();
  if (type == Espresso) {
    PrepareEspresso();
  } else if (type == Capuccino) {
    PrepareCapuccino();
  } else if (type == Glasse) {
    PrepareGlasse();
  }
}
}
```

```
enum Xk83_cC_394bn {
  OZ0ec,
  j0_390_SdntSXs,
  a
};

class _LdlJS38_38ncJ {
  void Sdc03_299(Xk83_cC_394bn xd93)
{
  X304afel3004dl();
  if (xd93 == OZ0ec) {
    XrKdJXld();
  } else if (xd93 == j0_390_SdntSXs) {
    os9Scxmd();
  } else if (xd93 == a) {
    dKcmx_eek291__x_c99();
  }
}
}
```

Рис. 1. Пример лексического затемнения

грамму, которая будет выполнять обратные преобразования, или же выполнение такой программы будет чрезвычайно ресурсоемким;

- незаметность преобразований T_{ik} является максимальной, т.е. по статическим свойствам S'_j будут достаточно близки к S_j;
- стоимость (дополнительное время выполнения или занимаемое пространство, вызываемое преобразованиями) P' является минимальной.

Выделяется несколько видов затемняющих преобразований исходя из способа их действия [3].

Лексические преобразования. Этот вид затемняющих преобразований является самым простым и в то же время обеспечивающим наименьшую защиту. Действие лексических преобразований распространяется только на лексическую структуру программы, т.е. фактически сводится к переименованию идентификаторов (переменных, классов, методов), затрудняя, таким образом, анализ логики программы на основании имен этих идентификаторов.

Основной сферой применения лексических преобразований являются интерпретируемые языки программирования, исходные программы на которых компилируются не в машинный код, а в *байт-код* – машинно-независимый псевдоязык, понятный интерпретатору языка. Обычно байт-код содержит информацию об исходном коде (имена идентификаторов, названия классов и т.п.), поэтому вопрос скрытия такой информации является существенным для интерпретируемых языков.

Пример лексического преобразования показан на рис. 1. Слева приведен оригинальный код программы, а справа – код той же программы после лексического затемнения.

Лексические преобразования предотвращают некоторое количество попыток воровства интеллектуальной собственности, однако не создают серьезного препятствия для опытного реверс-инженера.

Преобразования хода выполнения. Второй тип затемняющих преобразований оперирует с процессом выполнения программы, т.е. субъектом преобразований являются последовательности вызовов функций (операторов).

Коллберг в [4] предложил несколько затемняющих преобразований процесса выполнения программы, основанных на *непроницаемых предикатах* (opaque predicates). Предикат P является *непроницаемым* (opaque), если его значение известно до за-

темнения, но его сложно получить при анализе затемненной программы. Будем использовать обозначение P^F (P^T), если выходом P всегда есть ложь (истина), и $P^?$, если P иногда может возвращать истину, а иногда ложь. На базе множества непроницаемых предикатов можно построить затемняющие преобразования, которые вносят изменения в ход выполнения некоторой процедуры. На рис. 2,а мы разделяем блок последовательно выполняющихся команд A и B , вставляя простой истинный предикат P^T, который позволяет предположить, что B выполняется только иногда. На рис. 2,б команда B разделяется на две различные затемненные команды B и B'. Предикат $P^?$ случайным образом выбирает одну из них во время выполнения. На рис. 2,в P^T всегда выбирает B из пары команд B и B_{bug}, намеренно созданной неправильной версией B.

Существует достаточно большое количество преобразований процесса выполнения, похожих на показанные на рис. 3; некоторые из них обсуждаются в [4]. Устойчивость таких преобразований напрямую зависит от устойчивости непроницаемых предикатов, на которые они опираются. Таким образом, очень важно уметь конструировать устойчивые непроницаемые предикаты, важными критериями выбора которых является их незаметность и низкая стоимость.

При выборе непроницаемых предикатов логично исходить из проблем, с которыми сталкиваются автоматизированные системы, проводящие анализ затемненного кода. Так, большинство таких систем используют методы, основанные на статическом анализе. Принимая во внимание факт, что точный статический анализ структур, основанных на указателях и параллельных регионах, является достаточно затруднительным, эти структуры можно использовать в качестве непроницаемых предикатов.

В качестве примера сильных непроницаемых предикатов можно привести предикаты на основе псевдонимов. Основной идеей этого метода является добавление в программу кода, который строит множество сложных динамических структур. Множество глобальных указателей ссылаются на вершины внутри этих структур. Вышеуказанные структуры в ходе выполнения программы будут случайным образом обновляться (будут изменяться внутренние указатели, добавляться и удаляться вершины и т.п.), но некоторые условия будут поддерживаться в неизменном состоянии. Примерами таких условий могут быть «указатели p и q никогда не будут указывать на один и тот же элемент в куче», «существует путь от p до q» и т.п. Впоследствии эти условия будут использоваться для генерации непроницаемых предикатов.

В [5] в качестве преобразования хода выполнения предлагается вырождение графа логики выполнения. Определение графа логики программы извне – достаточно простая операция, линейно зависящая от количества простых блоков в программе. По вырожденному же графу достаточно сложно определить ход работы программы.

Вырождение графа логики проходит в

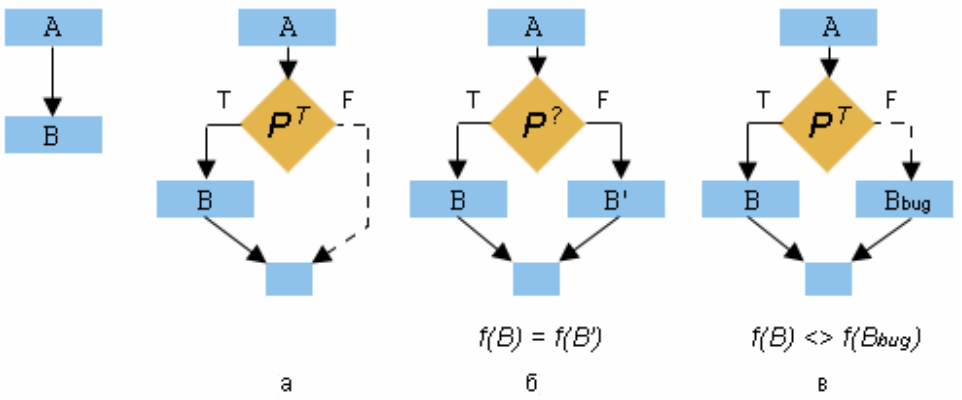

Рис. 2

два этапа. На первом этапе высокоуровневые структуры (например, циклы) преобразуются в эквивалентные `if-then-goto`-структуры. Пример такого преобразования показан на рис. 3.

На втором этапе операторы `goto`, появившиеся в результате первого этапа, модифицируются таким образом, что цели этих операторов `goto` определяются динамически. Пример второго этапа приведен на рис. 4.

Преобразования данных. Третьим типом затемняющих преобразований является затемнение данных. Наиболее распространенным преобразованием является *разделение переменной* – представление атомарного значения (например, целого или булевого) в виде некоторой сложной структуры данных. В [3] представлено несколько таких представлений. Как пример, рассмотрим разделение, показанное на рис. 5. Булева переменная *V* разделяется на две целые переменные *p* и *q*. Используя это представление, можно создать новые реализации для булевых операций. На рис. 6 показана реализация операции логического И.

На рис. 6 показано разделение трех булевых переменных `A`, `B` и `C` на короткие целые переменные `a1`, `a2`, `b1`, `b2`, `c1`, `c2` соответственно. Интересным аспектом выбранной интерпретации является возможность вычислить одно значение несколькими способами. Например, операторы (2') и (3') отличаются друг от друга, однако оба присваивают переменной значение `False`.

В [5] показано преобразование данных, основанное на замене прямых обращений к данным на косвенные. Так, вместо использования локальных переменных-указателей все указатели хранятся в некотором глобальном массиве. При необходимости получить нужный указатель вычисляется некоторая функция, зависящая от состояния программы. Возвращаемое значение этой функции содержит индекс в глобальном массиве, где находится нужный указатель.

Шифрование кода. Отдельным случаем затемнения являются методы, реализующие шифрование программного кода (как всего кода программы, так и отдельных его участков). Эти методы используются в основном для затемнения выполняемого машинного (не интерпретируемого) кода.

```
int a, b;
a = 1;
b = 2;
while (a < 10) {
  b = a + b;
  if (b > 10) {
    b--;
  }
  a++;
}
doSomething(b);
```

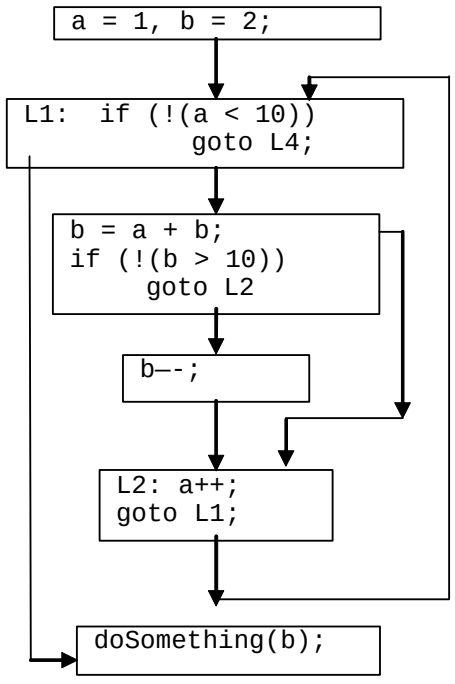

Рис. 3

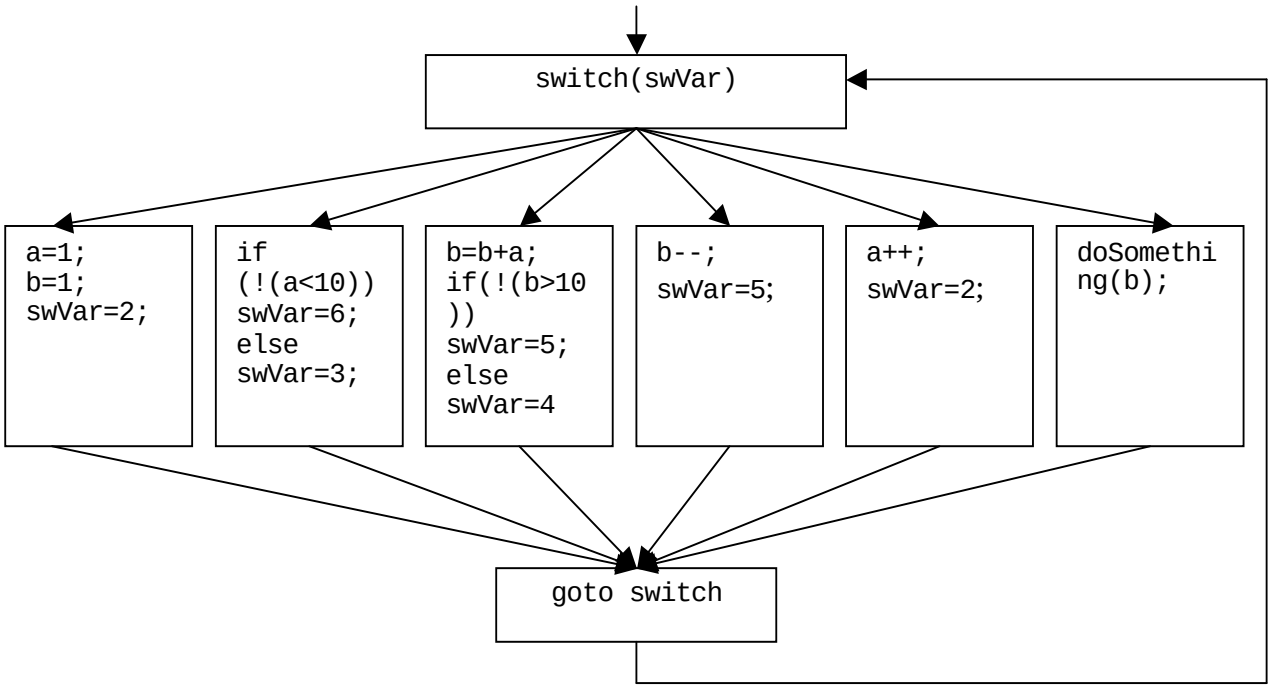

Рис. 4

Ключ для шифрования (и соответственно расшифрования) кода должен быть надежно защищен. В идеале, он должен отличаться для каждого пользователя программы. Однако такой подход является неприемлемым для продуктов с большим количеством пользователей, поскольку предполагает, что каждому пользователю будет выдаваться отдельная копия продукта, зашифрованная ключом этого пользователя. Поэтому более удобным вариантом является использование общего ключа и его хранение в недоступном месте (например, на смарт-карте или сервере приложений). Еще одним подходом к хранению ключа является его генерация в ходе выполнения программы, однако такой подход более уязвим к реверс-инженерии.

Можно выделить два подхода к расшифровке выполняемого кода: расшифровка всего кода при запуске программы и поэтапная расшифровка необходимых участков кода непосредственно перед их выполнением с последующей обратной зашифровкой. Слабостью первого подхода является возможность снять копию участка памяти, содержащего расшифрованную программу, поэтому этот подход крайне ненадежен. Для того чтобы получить программу в незашифрованном виде, атакующему нужно выполнить все зашифрованные фрагменты, для каждого фрагмента снимая дамп памяти. При большом количестве зашифрованных фрагментов эта задача становится практически невыполнимой.

Шифрование программного кода также может применяться и для защиты кода от модификации. При шифровании некоторого

g(V)		F(p,q)	
P	Q	V	2p + q
0	0	False	0
0	1	True	1
1	0	True	2
1	1	False	3

И	0	1	2	3
0	3	0	0	0
1	3	1	2	3
2	0	2	1	3
3	3	0	0	3

Рис. 5

(1) bool A, B, C;	(1′) short a1, a2, b1, b2, c1, c2;
(2) B = False;	(2′) b1 = 0; b2 = 0;
(3) C = False;	(3′) c1 = 1; c2 = 1;
(4) C = A & B;	(4′) x = AND[2*a1 + a2, 2*b1 + b2]; c1 = x/2; x2 = x%2;
(5) C = A & B;	(5′) c1 = (a1 ^ a2) & (b1 ^ b2) ; c2 = 0;
(6) if (A) …;	(6′) x = 2 * a1 + a2; if ((x==1) || (x==2)) …;
(7) if (B) …;	(7′) if (b1 ^ b2)…;

Рис. 6

участка кода совместно с данными этого участка кода можно поместить контрольную сумму его байт. Таким образом, если злоумышленник внесет изменения в шифртекст (например, с экспериментальной целью), контрольная сумма при расшифровке не совпадет с оригинальной и эти изменения можно будет отследить.

Определенный интерес представляет метод на основании *саморасшифровывающегося кода*, описанный в [6]. Данный метод не предполагает наличия некоторого фиксированного ключа. Программный код разбивается на участки, которые шифруются независимо друг от друга. В процессе выполнения программы каждый следующий участок расшифровывается ключом, полученным применением некоторой функции к байтам предыдущего. Очевидно, что этот метод не предоставляет защиты от неавторизованного выполнения кода, а только от реверс-инженерии и модификации выполняемого кода.

В [7] предлагается использовать шифрование кода в связке с интерпретатором команд. Инструкции последовательно расшифровываются и передаются на выполнение интерпретатору.

Другие подходы. В [6] представлен практический метод затемнения, направленный на сокрытие информации о следе выполнения (execution path) и доступе к данным. Идея заключается в том, что программа делится на сегменты, постоянно перемещаемые в памяти. Каждый раз после выполнения конкретный участок кода перемещается в другую область памяти, делая, таким образом, невозможным анализ кода, исследуя последовательность адресов доступа при выполнении программы.

Лоурейро и Молва [8] представили метод сокрытия функций, выполняющихся в потенциально опасном окружении, на основе кодов, исправляющих ошибки. Целями метода являются обеспечение секретности исполняемого алгоритма (т.е. сокрытие внутреннего поведения программы) и его целостности (атакующий не может изменить алгоритм).

В [9] предложен метод сокрытия функций, который основывается на шифровании функций таким образом, что и в зашифрованном виде они могут выполняться. Существенным его ограничением является неуниверсальность (метод позволяет скрывать только полиномиальные функции).

Эффективность защиты. Затемнение путем преобразований хода выполнения или разделения данных обеспечивает достаточно серьезную защиту от обратного проектирования кода. Для взлома затемненной таким образом программы атакующему необходимо воссоздать граф выполнения, для чего он должен определить, какие из непроницаемых предикатов являются «реальными», а какие – «заглушками». Автоматизировать подобную задачу достаточно сложно.

Поскольку большинство затемняющих преобразований детерминированы, то задача затемнения легко автоматизируется, ее можно свести к задаче создания лексического/синтаксического анализатора и преобразования полученных от них структур. По-

скольку все без исключения компиляторы включают такие анализаторы, то на уровне компилятора затемнение реализуется просто.

Существенным плюсом затемнения, делающим его наиболее предпочтительным из всех рассматриваемых методов, является его полная автономность. Затемнение оперирует только с исполняемым кодом и не требует для своей работы никаких дополнительных модулей (как программных, так и аппаратных).

Однако затемнение не является универсальным методом. В [10] доказывается, что невозможно построить затемняющее преобразование, которое будет эффективно затемнять любую входную программу, и приводится семейство функций, для которых невозможно построить эквивалентные затемненные функции. Тем не менее на практике для большинства программ можно построить затемняющие преобразования, которые будут обеспечивать достаточно высокий уровень защиты.

Вынесение критического программного кода в отдельный защищенный модуль

Эта группа методов основывается на переносе критических участков программного кода (например, конфиденциальных данных или данных, представляющих собой интеллектуальную собственность) на отдельный носитель, имеющий существенно бо́льшую степень защиты, чем исходная программа. Под такой степенью защиты понимается невозможность доступа к коду и к данным, которые хранятся на носителе, невозможность трассировки выполнения кода и т.п. Когда программе необходимо получить защищенные данные или выполнить секретный код, она обращается к такому носителю по некоторому протоколу и получает эти данные. Таким образом, атакующему, вместо исходной программы, приходится противодействовать укрепленному носителю информации.

Кроме обеспечения сокрытия данных, метод с применением защищенного модуля решает и другие задачи защиты программного обеспечения. Во-первых, защищенный носитель может проверять, имеет ли пользователь право на запуск программного продукта (аутентифицировать пользователя). Во-вторых, он обеспечивает целостность программного кода, по крайней мере тех участков, которые хранятся и выполняются на нем.

Защищенные модули можно разделить на два типа: пассивные (модули, которые только сохраняют данные, но не имеют возможности выполнять программный код) и активные (модули, поддерживающие выполнение кода). Очевидно, что вторые представляют больший интерес, так как, в отличие от первых обеспечивают защиту алгоритмов.

Можно привести несколько разновидностей таких модулей.

Защитные заглушки были одним из первых средств противодействия атакам на программное обеспечение. Заглушка представляет собой небольшое устройство, которое подсоединяется к некоторому (например, LPT, COM или USB) порту компьютера.

Наиболее простые устройства могут содержать только идентификационные данные пользователя. Программа при своем запуске проверяет наличие такого устройства в соответствующем месте и при его отсутствии аварийно завершается. Такой подход обеспечивает защиту только от пиратства и не обладает большой стойкостью, так как злоумышленник легко может отключить команды проверки наличия заглушки в коде программы. Более сложные заглушки могут содержать микроконтроллер, который обладает способностью обрабатывать запросы программы, возвращая соответствующие им ответы. В ходе выполнения программа посылает устройству запросы и проверяет правильность ответов. Такой подход обеспечивает несколько большую защиту, однако и она может быть снята использованием таблиц вопросов-ответов. Наиболее сложные устройства обладают небольшим количеством памяти и поддерживают возможность выполнения программного кода. В этом случае наиболее критические части программного обеспечения могут располагаться и выполняться на контроллере. В этом случае стойкость за-

щиты полностью зависит от стойкости защиты контроллера.

Смарт-карты являются сравнительно молодой разновидностью защитных заглушек. Ранее они уже успешно применялись в качестве средства обеспечения защиты в других областях – например, предоплаченные телефонные карточки, sim-карты. Концептуально смарт-карты имеют лишь два отличия от заглушек:

- являются стандартизированными, устройства их чтения и записи доступны для большинства платформ;
- смарт-карты могут содержать более чем одну систему защиты, так как данные на них могут сохраняться и изменяться после выпуска карты.

В настоящее время широко используются так называемые криптокарты (crypto token). Криптокарта представляет собой небольшое устройство, которое «умеет» хранить секретные ключи и выполнять криптографические вычисления (например, вычисление модульной экспоненты). Методы защиты ПО, основанные на использовании устойчивых аппаратных средств, достаточно широко описаны в литературе (напр., [11]).

Подход на базе **использования доверенного сервера** обеспечивает достаточно высокий уровень защиты. Он заключается в следующем. Разработчик ПО размещает в Интернете сервер приложений и на этот сервер перекладывается функциональность наиболее критичных участков ПО. Выполняясь на пользовательской машине, программа посылает серверу запрос на выполнение определенной функции, после чего он возвращает ей результат:

Очевидно, что метод с использованием доверенного сервера полностью решает проблему противостояния обратному проектированию и, в частности, проблему раскрытия секретного алгоритма. Кроме того, этот метод может использоваться для защиты от пиратства путем добавления в протокол вызова серверной функции аутентификации клиента. Однако за обеспечиваемую защиту приходится платить некоторыми недостатками. Во-первых, поскольку обмен данными между клиентскими компьютерами и сервером выполняется через сеть, доступ к которой может получить кто угодно, то необходима строгая аутентификация доверенного сервера и шифрование потоков данных между клиентскими компьютерами и доверенным сервером. Во-вторых, на разработчика ложится двойная задача – необходимо обеспечить еще и защиту сервера от внешних атак (подходы к организации такой защиты выходят за рамки данной статьи). Слабостью подхода является и достаточно низкая производительность (например, при необходимости многократного выполнения на сервере некоторой функции), связанная с пропускной способностью каналов связи и ограниченными возможностями сервера. Кроме того, недостатком является зависимость конечного пользователя от наличия подключения к Интернету и состояния сервера (при этом пользователь не может быть уверен, что разработчик будет на рынке через 5-10 лет).

Существует множество вариантов ([9], [12]), рассматривающих разные способы применения подхода на базе использования доверенного сервера; большинство из них посвящено протоколам обмена данными между сервером и компьютерами пользователей.

Эффективность защиты. Описанная схема защиты имеет несколько узких мест, соответственно, существует несколько возможных атак, связанных с их эксплуатацией.

Первым узким местом, общим и для доверенного сервера, и для защитных заглушек, является канал между клиентским ком-

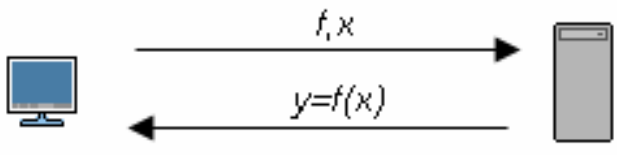

Рис. 7

пьютером и защищенным устройством. Так как кто угодно может вклиниться в этот канал, то возникает необходимость обеспечения аутентификации обоих участников и шифрования протокола, по которому стороны обмениваются информацией.

Вторым узким местом является защищенное устройство. В этом аспекте доверенный сервер является гораздо более предпочтительным вариантом, нежели заглушки, так как злоумышленник не имеет к нему физического доступа. Обеспечить защиту сервера от внешних атак гораздо дешевле, чем создать полностью защищенную от вторжения заглушку.

Стоимость подхода на основе защитных заглушек или смарт-карт достаточно высока, поскольку каждого пользователя продукта нужно обеспечить таким устройством. Поэтому данный подход подойдет продуктам, имеющим не слишком большое количество пользователей. Существенно более дешевым является подход на базе доверенного сервера. При сравнительно невысокой стоимости он обеспечивает очень высокий уровень защиты.

Защита от модификации кода программы

Целостность программного кода может быть нарушена по разным причинам. Во-первых, внести изменения в код может вирус, внедрившийся в программу. Во-вторых, изменения могут быть преднамеренно внесены недоброжелателем (например, таким способом он может отключить систему проверки лицензионного кода либо попытаться удалить из программы водяной знак). В любом случае выполнение измененного программного кода нежелательно по причине невозможности гарантировать правильность его работы. Для предотвращения выполнения программы в случае, если ее код был модифицирован, в программу можно добавить специальный проверочный код, который, в случае обнаружения изменений в коде программы, будет ее аварийно завершать. На этот код будут возложены такие задачи:

а) обнаружить, что программа была модифицирована,

б) если это возможно, исправить обнаруженные изменения, в противном случае обеспечить аварийное завершение программы,

В идеальном случае обнаружение изменения в коде и соответствующее этому обнаружению аварийное завершение программы должны быть широко распределены во времени и коде программы, чтобы запутать потенциального атакующего. Очевидно, что код `if (tampered()) i = 1 / 0` не обеспечивает достаточной защиты в связи с возможностью тривиального обнаружения и устранения.

Следующие два способа являются наиболее используемыми для проверки программы на наличие внесенных изменений:

1) исследовать исполняемый код программы и проверить, совпадает ли он с оригинальным. Для этого можно использовать некоторую одностороннюю хэш-функцию (напр., SHA1). Проверять можно не весь код, а только части, для которых неизменность является критичной. Один из вариантов таких проверок приводился в разделе, посвященном шифрованию программного кода;
2) исследовать правильность промежуточных результатов, получаемых в программе. Этот подход известен как *проверка программы* (program verification) [13], [14]. Кроме рассматриваемых целей, этот метод также может использоваться для верификации/ тестирования программы на этапе разработки и является самым применяемым для решения задачи обнаружения изменений в программе.

В [15] для проверки приложения на неизменность предлагается использовать множество объектов – троек <*тестер, интервал, корректор*>. *Тестером* называется подпрограмма, вычисляющая хэш-функцию от некоторого *интервала* (участка кода) программы и сравнивающая его с оригиналом. В процессе выполнения тестеры запускаются и проверяют правильность интервала, за который они отвечают, путем сравнения оригинального значения хэш-функции и действительного. В случае, если значения не совпа-

дают, программа через некоторое время аварийно завершается. *Корректоры* необходимы для безопасного хранения оригинальных значений хэш-функции. Корректоры могут быть как константами (т.е. просто массивом байт), так и подпрограммами, конструирующими оригинальное значение. Очевидно, второй подход обеспечивает бо́льшую защиту.

Использование каскадного подхода и создание зависимостей между тестерами поможет усложнить защиту, обеспечиваемую тестерами. В этом случае злоумышленнику для снятия защиты даже только с одного интервала потребуется обнаружение и уничтожение всех тестеров, находящихся в коде программы.

Подход, похожий на рассмотренный выше, описывается в [16]. Ответственными за целостность приложения являются *стражники* (guards) – подпрограммы, выполняющие проверку целостности данных. Отличием их от тестеров является возможность восстановления измененных участков кода. Для исправления изменений используются коды, исправляющие ошибки.

Несмотря на то, что на основе подхода, использующего проверки участков выполняемого кода, можно построить достаточно сложную систему защиты программы от модификации, этот подход имеет один серьезный концептуальный недостаток. Поскольку проверка целостности выполняется на уровне программного кода, то после внедрения защиты код программы не может быть изменен. Это приводит к достаточно большим неудобствам – например, к невозможности оптимизации (сжатия) программного кода с сохранением семантики, невозможности добавить к программе цифровую подпись. Кроме того, поскольку тестеры следят за целостностью *лексики*, в то время как в первую очередь стоит задача обеспечить целостность *семантики*, возникает проблема анализа и оценки уровня обеспечиваемой защиты.

У этого подхода имеются и другие недостатки. Например, операция чтения программой своего собственного кода (характерная для приложений, проверяющих свою целостность вышеописанным образом) является редко выполняемой, а потому легко обнаружимой. Поэтому более предпочтительны методы, проверяющие целостность семантической структуры программы.

Один из таких методов, названный *забывчивым хэшированием* (oblivious hashing), рассматривается [17]. В качестве объекта проверки предлагается использовать не бинарный код, а *след выполнения* программы (функции, участка программы и т.п.). Программа (функция, участок) представляется в виде последовательности абстрактных машинных инструкций $I=\{i_1, i_2, \ldots, i_N\}$, которые обращаются для чтения и записи к участкам памяти $M=\{m_1, m_2, \ldots, m_K\}$, начального состояния памяти M_0, счетчика инструкций C и его начального значения C_0. Следом некоторого участка программы называется пятерка (I, M, C_0, M_0, P), где P – некоторый входной параметр, который влияет на выполнение участка программы. В процессе выполнения участка программы вычисляется хэш-функция от его следа. Поскольку след отражает семантику участка программы, полученное значение хэш-функции по сути является цифровой подписью, выполненной над поведением этого участка. Изменение как кода, так и данных, с которыми оперирует участок, приведет к несовпадению полученного при проверке значения хэш-функции с оригинальным значением, что позволит определить наличие изменений в программе.

Недостатком данного подхода является его неуниверсальность. Так, проверку методом забывчивого хэширования невозможно реализовать для участков программы, оперирующих случайными данными (такими, как текущее время, положение указателя мыши и т.п.), или данными, множество значений которых очень велико. Это существенно ограничивает множество подпрограмм, для которых можно применить этот метод.

Эффективность защиты. Защита от модификации кода программы является неотъемлемой частью общей задачи защиты клиентского программного обеспечения от злонамеренного хоста. Кроме того, программа, обладающая возможностью обнаружить изменения в своем коде, является более

безопасной и в случае выполнения на доброжелательном хосте. Программа, не отслеживающая изменений в своем коде, потенциально может представлять угрозу как для себя, так и для других программ, выполняющимся с ней в одном окружении.

Основными атаками на алгоритмы защиты от модификации, оперирующие бинарным кодом программы, являются удаление проверочного кода и подмена хранимых внутри программы оригинальных значений хэш-функции. Из этого следует, что и проверочный код, и оригинальные значения хэш-функций должны быть хорошо защищены от воздействия. Для этих целей можно использовать затемнение, помещение проверочного кода/данных на защищенный носитель и т.п. Атаки на алгоритмы, проверяющие целостность семантической структуры программы, являются гораздо более сложными, в первую очередь из-за сложности их обнаружения.

Создание защищенной среды выполнения

Методом, обеспечивающим достаточно высокий уровень безопасности клиентского ПО, является перенос функций защиты в среду выполнения. Примерами таких защищенных сред может быть ядро операционной системы, виртуальная выполняющая машина или же оборудование, поддерживающее защиту ПО. Под защищенностью среды понимается сложность или невозможность доступа к устройству и содержимому среды извне, а также наличие определенных функциональных возможностей по обеспечению безопасности выполняемой программы.

Можно выделить основные задачи, которые будут возложены на защищенную среду выполнения:

- проверка целостности программного кода;
- запрещение доступа к адресному пространству программы;
- контроль за выполнением программы, отслеживание необычных и ошибочных ситуаций;
- обеспечение шифрования программного кода;
- проверка наличия у пользователя права на выполнение конкретной копии программы.

Очевидно, что реализация данного метода не сводится к реализации только защищенной среды (например, операционной системы). Понадобится целая инфраструктура различных объектов (среда выполнения, компиляторы, отладчики и т.п.) для того, чтобы обеспечить разработчикам возможность легко разрабатывать программы, не волнуясь про обеспечение их защищенности. Однако, несмотря на сложность реализации подхода, защита, которую будет обеспечивать такая инфраструктура, будет более чем высокой.

Схематично данный подход можно изобразить следующим образом:

В подходе используются два основных

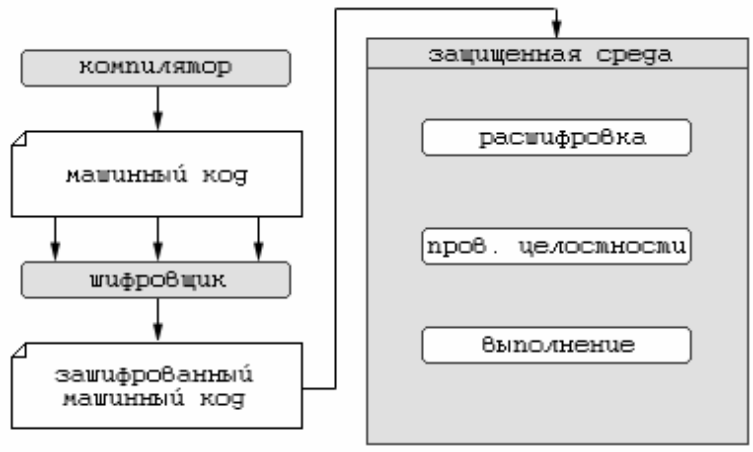

Рис. 8

инструмента – шифровщик и защищенная выполняющая среда. Шифровщик содержит в себе два ключа – открытый (для шифрования) и закрытый (для подписывания). Зашифрованная среда также содержит в себе два ключа, соответствующие ключам, принадлежащим шифровщику.

Выполняемый код, созданный компилятором, передается на вход шифровщику, который подписывает его секретным ключом, после чего шифрует открытым. При загрузке защищенной программы, среда расшифровывает ее с помощью закрытого ключа и проверяет целостность с помощью открытого. Если целостность программы подтверждена, защищенная среда выполняет ее.

Таким образом, программа является защищенной криптографически стойкими методами как от анализа кода, так и от изменения.

Кроме встроенных в ядро функций защиты, защищенная среда может предоставлять программе некоторый API (программный интерфейс) для работы с защитой. Например, программа сама может «спросить» у ОС (вызывая ее функции), не была ли она изменена. Кроме того, программа может передавать введенный пользователем лицензионный ключ на проверку операционной системе, а операционная система (для проверки лицензионного ключа) будет соединяться с сервером разработчика программы. Среда также может предоставлять программе защищенное хранилище данных. Таким образом, вместо того чтобы хранить секретные данные в открытом виде, программа может передавать их на хранение среде для обеспечения их безопасности.

Эффективность защиты. Защищенная среда выполнения обеспечивает очень высокий уровень защиты от злонамеренного хоста. Во-первых, гарантируется полноценная защита от модификации программного кода, поскольку его целостность проверяется на уровне среды и защищается надежными криптографическими методами. Кроме того, вследствие ограничения доступа к данным программы во время ее выполнения задача обратного проектирования существенно усложняется. Фактически реверс-инженеру остаются только две возможности: анализировать выполняемый код программы (причем и эта возможность отпадает при использовании шифрования программного кода) либо ограничиться для анализа программы методом «черного ящика». Следует также отметить, что программы, выполняющиеся в такой среде, абсолютно устойчивы к заражению компьютерными вирусами.

Есть и другое достоинство – перенос задачи защиты в защищенную среду выполнения позволит разработчику решать поставленные перед ним задачи, не тратя времени на построение системы защиты для своей программы.

Данный подход имеет два узких места, которые может эксплуатировать атакующий. Первой возможной атакой является подмена среды выполнения. Поэтому необходима обязательная аутентификация программой защищенной среды (эта возможность может встраиваться в программу шифровщиком). Для аутентификации среды могут использоваться криптографические алгоритмы с открытым ключом, такие, как RSA или алгоритм Эль-Гамаля.

Вторым узким местом является необходимость обеспечения целостности самой защищенной среды. Поскольку такая среда является полностью доступной для анализа, обеспечение ее целостности – достаточно сложная задача. Создатели среды должны также обеспечить безопасное хранение секретных ключей, необходимых для подписывания и расшифровки кода. Аналогичные требования накладываются и на программу-шифровщик.

Лицензионные метки

Под термином *лицензионной метки* будем понимать некоторую сущность, позволяющую определить, имеет ли программа право на выполнение определенной функции на определенной вычислительной машине. Основной и практически единственной задачей лицензионных меток является защита ПО от пиратства. Этот метод очень распространен среди современных производителей ПО.

В качестве метки могу выступать: уникальный лицензионный номер (ключ), код активации, файл с лицензией, менеджер лицензий, онлайновая система активации ПО и т.п. Лицензия может привязываться к имени пользователя, организации или существующему оборудованию компьютера.

Безопасность, обеспечиваемая данным методом, зависит исключительно от алгоритмов формирования и проверки лицензионных меток. Для взлома простейших лицензионных меток достаточно приобрести одну лицензию на программу, после чего раздать полученную лицензионную метку всем желающим. Более надежной является лицензионная метка с необходимостью онлайновой активизации и привязкой к оборудованию, но она приводит к некоторым неудобствам для пользователей.

Основными атаками на такие схемы защиты, кроме непосредственного опубликования лицензионной метки, являются: а) создание генератора меток и б) обнаружение и удаление кода, отвечающего за проверки лицензионной метки из ПО. Можно отметить, что генератор ключей для Windows XP был создан за несколько месяцев, несмотря на то что эта операционная система имела достаточно хорошо проработанную защиту с системой онлайновой активации и привязкой к аппаратуре.

Для избежания атаки (а) необходимо, чтобы алгоритм, используемый для построения лицензионных меток, основывался на стойком криптографическом алгоритме. В частности, в [18] описана система лицензирования, основанная на криптографических алгоритмах с открытым ключом. Что касается противостояния атаке (б), то сама по себе лицензионная метка ее не обеспечивает. Поэтому при использовании этого метода важно обеспечить также защиту программы от обратного проектирования.

Принимая во внимание то, что недостатком систем онлайновой активизации является требование подключения к Интернету, такая схема может быть удобна для программных продуктов, деятельность которых связана с этим (браузеры, почтовые клиенты, серверы и т.п.)

Следует отметить, что лицензионной меткой не обязательно должен являться числовой регистрационный ключ. Так, в этом качестве можно использовать часть функциональности продукта. Например, меткой может быть динамически подключаемая библиотека (DLL), содержащая в себе, кроме лицензионных данных, некоторую функциональность, необходимую для корректной работы продукта. Такой подход обладает двумя положительными качествами – во-первых, его применение сделает невозможным создание генератора ключей, так как атакующий не обладает исходным кодом для сборки библиотек, во-вторых, атакующий не может применить атаку на обнаружение проверочного кода ввиду ее бессмысленности.

Эффективность защиты. Как было сказано выше, уровень защиты, обеспечиваемый лицензионной меткой, зависит только от алгоритмов, используемых в ней самой. Процесс обработки меток программой сам по себе достаточно неустойчив к обратному проектированию, поэтому использование меток имеет смысл только при наличии дополнительной защиты программы в виде затемнения или защищенного модуля. Когда такая защита присутствует, лицензионные метки являются эффективным и недорогим методом защиты от пиратства.

Альтернативные подходы к защите программного обеспечения

Кроме описанных выше методов защиты клиентского ПО, можно отметить несколько альтернативных подходов к его безопасности. Так, в [19] предлагается метод, который можно назвать *эволюционированием продукта* (software aging). Этот метод основывается на периодических обновлениях ПО. Несмотря на кажущуюся бессмысленность, он оказался достаточно хорошей защитой от нелицензионного использования ПО. Суть его заключается в том, что производитель ПО через некоторые, сравнительно небольшие интервалы времени (один-два месяца), обновляет продукт, немного изменив метод защиты. Пользователям, обладающим легальной версией, необходимо просто ее обновить.

Пользователи же, использующие нелегальную версию, должны ждать, пока взломщик вскроет обновленную версию. Таким образом, нелегальные пользователи становятся зависимыми от взломщика, у которого появляется несравнимо больше обязанностей – теперь ему нужно взламывать новую версию продукта каждый месяц. Задача усложняется, если от версии к версии производитель серьезно изменяет защиту продукта. Важность взлома очередной версии для нелегального пользователя тем больше, чем больше очередная версия продукта несовместима с предыдущей.

Еще одним альтернативным подходом к защите является *кастомизация* – независимая сборка программного продукта для каждого конкретного клиента. В таких случаях для каждого зарегистрированного пользователя собирается отдельная версия продукта с помещенной в код информацией про особу, на которую регистрируется продукт. Таким образом предотвращается получение копии продукта случайным лицом, а в случае, если продукт будет незаконно опубликован, по имеющейся в его коде информации можно будет определить совершившего кражу. Однако метод создания персональных сборок имеет смысл только при достаточно небольшом количестве пользователей программы, так как при большом количестве пользователей он будет слишком накладным.

Кроме математических подходов к защите ПО существует еще множество технических подходов, усложняющих вломщику задачу. Так, широко распространенным является применение защиты от отладки – комплекса мер, направленных на обнаружение факта запуска программы «внутри» отладчика, и аварийное завершение ее работы в случае, если отладчик будет обнаружен.

Затемнение + проверка целостности – комбинированный метод защиты ПО

Основываясь на [4], [17], [20] и [21], можно предложить комбинированный метод защиты ПО от обратного проектирования и модификации кода. Метод состоит из двух независимых составляющих: затемнения и добавления проверочного кода.

По аналогии с определением непроницаемого предиката, приведенного выше, приведем понятие непроницаемой функции.

Непроницаемой назовем функцию, результат которой известен на этапе затемнения, но его сложно определить при исследовании затемненной программы.

Затемняющая составляющая метода состоит из двух этапов, суть каждого из которых состоит в выполнении преобразований над исходным кодом программы. На первом этапе каждой переменной (константе), используемой в программе, поставим в соответствие некоторый объект, который будет хранить ее значение. В качестве такого объекта можно рассматривать как объект в понимании ООП, так и просто указатель на структуру, в которой хранятся данные переменной. Создадим глобальный массив `obj` всех объектов, используемых в программе. Таким образом, каждая переменная/константа, используемая в программе, будет уникально идентифицироваться индексом в этом глобальном массиве.

Далее, определим в программе функцию f такую, что

- f является непроницаемой;
- входом f является целое число – уникальный идентификатор некоторой точки в программе, в которой происходит обращение к переменной, выходом – индекс в массиве объектов.

В исходной программе заменим все обращения к переменным/константам на обращения к соответствующим объектам таким образом:

- каждый объект адресуется как некоторый элемент глобального массива объектов;
- номер объекта в массиве определяется динамически с помощью вызова функции f.

Таким образом, после первого этапа все прямые обращения к переменным в коде программы заменены на косвенные, что существенно усложняет обратный анализ про-

граммы. При достаточной непроницаемости функции f задача определения, какой именно объект является параметром при вызове той или иной функции, является практически неразрешимой.

На втором этапе создадим глобальный массив `func` всех функций, вызываемых в программе. Каждой функции в соответствие поставим уникальный индекс в этом массиве. Аналогично функции f введем непроницаемую функцию g, которая, принимая некоторый уникальный целочисленный аргумент, будет возвращать номер функции в массиве функций. Аналогично тому, как это делалось на первом этапе, заменим все прямые вызовы функций на косвенные.

Замена всех прямых вызовов функций на косвенные, определяемые на этапе выполнения, делает недоступной для статического анализа информацию о последовательности вызовов функций. Фактически единственная информация, доступная атакующему после второго этапа, – это последовательность выполнения условных операторов и операторов цикла (if, while, switch и т.п.), однако этой информации недостаточно для восстановления логики работы программы, так как аргументы этих операторов защищены непроницаемой функцией.

Составляющая метода, ответственная за проверку целостности, основывается на принципе «забывчивого хэширования», предложенном в [17]. Однако в отличие от «забывчивого хэширования», применимость которого имеет существенные ограничения, предложенный метод может использоваться для проверки целостности любых подпрограмм.

Объектом проверки являются непрерывные безусловные участки кода, целостность каждого из которых проверяется независимо. По ходу выполнения участка, команда за командой, на вход хэш-функции *h* подаются индексы вызываемой функции и ее параметров в глобальных массивах.

После завершения выполнения участка программы полученное значение хэш-функции сравнивается с корректным значением для этого участка. Таким образом обеспечивается целостность хода выполнения программы – последовательности вызываемых функций и передаваемых им объектов-параметров. Однако следует отметить, что вышеуказанная процедура проверяет только целостность *ссылок* на параметры функцій, т.е. что в функцию передаются правильные переменные. Процедура не гарантирует целостности

```
s = 1;
t = x;
u = y;
while (u) {
  if (u & 1)
    s = (s * t) % n;
  u >>= 1;
  t = (t * t) % n;
}
Console.WriteLine(s);
```

```
obj[f(1)] = obj[f(2)];
obj[f(3)] = obj[f(4)];
obj[f(5)] = obj[f(6)];
while (obj[f(7)]) {
  if (obj[f(8)] & obj[f(9)])
    obj[f(10)] = (obj[f(11)]*obj[f(12)])%obj[f(13)];
  obj[f(14)] >>= obj[f(15)];
  obj[f(16)] = (obj[f(17)]*obj[f(18)])%obj[f(19)];
}
obj[f(20)].WriteLine(obj[f(21)]);
```

Рис. 9. Результат первого этапа

0	1	2	3	4	5	6	7
s	1	t	x	u	y	N	Console

Рис. 10. Глобальный массив объектов

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
0	1	2	3	4	5	4	4	1	0	0	2	6	4	1	2	2	2	6	7	0

Рис. 11. Возвращаемые значения функции f

```
func[g(1)](obj[f(1)], obj[f(2)]);
func[g(2)](obj[f(3)], obj[f(4)]);
func[g(3)](obj[f(5)], obj[f(6)]);
while (func[g(4)](obj[f(7)])) {
  if (func[g(5)](obj[f(8)], obj[f(9)]))
    func[g(8)](obj[f(10)], func[g(7)](func[g(6)](obj[f(11)], obj[f(12)]),
      obj[f(13)]));
  func[g(9)](obj[f(14)], obj[f(15)]);
  func[g(12)](obj[f(16)], func[g(11)](func[g(10)](obj[f(17)], obj[f(18)]),
    obj[f(19)]));
}
func[g(13)](obj[f(20)], obj[f(21)]);
```

Рис. 12. Результат второго этапа

0	1	2	3	4	5	6
=	!=0	&	*	%	>>=	WriteLine

Рис. 13. Глобальный массив функций

данных, содержащихся в используемых объектах. Для проверки целостности данных можно использовать следующий подход. При изменении значения, хранящегося в какомлибо объекте, вычисляется хэш-функция от значений всех объектов в глобальном массиве. При каждом последующем обращении к значению объекта, значение хэш-функции подчитывается заново и сверяется с сохраненным. Различие этих двух значений свидетельствует о том, что в данные программы были внесены изменения.

Стойкость предложенного метода напрямую зависит от стойкости используемых непроницаемых функций. Проблема построения таких функций выходит за рамки данной статьи, с некоторыми примерами можно ознакомиться в [4]. Предложенный метод имеет следующие достоинства по сравнению с методами, описанными в литературе ранее.

- Использование непроницаемых функций вместо непроницаемых предикатов, а также косвенных вызовов функций позволяет достичь более высокого уровня сокрытия данных.
- Процедура проверки целостности «работает» на уровне логики программы, а не на уровне выполняемого кода, а следовательно, является независимой от особенностей среды выполнения.
- Процедура проверки целостности является универсальной и не зависит от защищаемой программы.

Выводы

Проблема защиты ПО от действий злонамеренного хоста является сравнительно молодой (первые работы в этой области появились в 1996 году). В то же время за последние несколько лет было опубликовано множество работ, посвященных этой проблеме. Во многом это обусловлено ее актуальностью в связи с лавинообразным ростом отрасли

```
func[0](obj[0], obj[1]);
func[0](obj[2], obj[3]);
func[0](obj[4], obj[5]);
while (func[1](obj[4])) {
  if (func[2](obj[4], obj[1]))
    func[0](obj[0], func[4](func[3](obj[0],
      obj[2]), obj[6]));
  func[5](obj[4], obj[1]);
  func[0](obj[2], func[4](func[3](obj[2],
    obj[2]), obj[6]));
}
func[6](obj[7], obj[0]);
```

```
h(0, 0, 1);
h(0, 2, 3);
h(0, 4, 5);
h(while, 1, 4);
  h(if, 2, 4, 1);
    h(0, 0, 4, 3,
      0, 2, 6);
  h(5, 4, 1);
  h(0, 2, 4, 3, 2,
    2, 6);

h(6, 7, 0);
```

Рис. 14. Хэширование данных в ходе выполнения программы

информационных технологий.

Несмотря на большое количество опубликованных работ, проблема защиты ПО от злонамеренного хоста находится в зародышевом состоянии. Практически отсутствуют работы, по формализации проблемы и постановке общих задач, связанных с защитой; большинство посвящено конкретным методам, защите от конкретных атак, а не общему подходу к защите. Отсутствуют математические модели угроз и методы оценки способов защиты.

Тем не менее методы, приведенные в статье, несмотря на их техничность, являются достаточно общими и обеспечивают защиту от широкого спектра атак. В то же время ни один из них не обеспечивает универсальной защиты, а потому для достижения максимального уровня защиты рекомендуется использование всех доступных методов.

Из рассмотренных наиболее привлекательным представляется метод, основанный на использовании защищенной среды выполнения. Однако его внедрение достаточно дорого, поскольку связано с разработкой такой защищенной среды. Тем не менее уровень защиты, обеспечиваемый защищенной средой, является адекватным стоимости ее разработки.

Метод с использованием доверенного сервера также обеспечивает высокий уровень защиты, однако доставляет определенные неудобства для пользователей ПО. Затемнение кода – удачный компромисс между обеспечиваемой им защитой, стоимостью внедрения и удобством использования. В отличие от метода на основе защищенного модуля оно не требует никаких дополнительных средств для своей работы (как программных, так и аппаратных). Однако стойкость защиты, обеспечиваемой затемнением, зависит от конкретного его вида, и, кроме того, затемнение может применяться не для всего множества программ.

Отдельно следует отметить важность процедуры проверки целостности программы. Если программа будет в состоянии определить присутствие изменений в ее коде, это сможет предотвратить не только попытки незаконного завладения программой, а и множество других отрицательных воздействий, таких, как заражение компьютера вирусом или кража конфиденциальных данных. Поэтому проверка на присутствие изменений в коде программы существенно повышает общую защищенность программы и вычислительных устройств, на которых она будет выполняться.

Комбинированный метод защиты, приведенный в статье, развивает идеи, предложенные в [17] и [20]. Предложенный метод обладает рядом достоинств, в частности универсальностью и независимостью от конкретной платформы. Кроме того, использование непроницаемых функций вместо непроницаемых предикатов обеспечивает более высокий уровень сокрытия программного кода.

Проблема защиты программного обеспечения от злонамеренного хоста является одной из наиболее сложных проблем современного этапа развития информационных технологий. Несмотря на обилие предлагаемых решений, задача защиты еще даже не формализована. Можно выделить следующие шаги, которые должны быть предприняты в этом направлении в первую очередь:

- формализовать понятия «защитить программу», «взломать программу»;
- определить модель угроз, дать математические оценки сложности конкретных атак и методов противостояния им.

Только после того, как два эти шага будут сделаны, можно будет оценивать стойкость алгоритмов в математическом смысле.

1. *Шнайер Б.* Прикладная криптография. Протоколы, алгоритмы, исходные тексты на языке Си. - М.: Изд. ТРИУМФ, 2002. - 816 с.
2. *Business* Software Alliance. - http://www.bsa.org.
3. *Collberg C., Thomborson C., Low D.* Breaking Abstractions and Unstructuring Data Structures // Proc. of the 1998 Intern. Conf. on Computer Languages, 1998. - P. 28.
4. *Collberg C., Thomborson C., Low D.* Manufacturing Cheap, Resilient and Stealthy Opaque Constructs // Proc. of 25th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, 1999. - P. 184 – 196.
5. *Wang C., Davidson J., Hill J.* Protection of Software-based Survivability Mechanisms // Proc. of the 2001 In-

tern. Conf. on Dependable Systems and Networks (formerly: FTCS), 2001. - P. 193 – 202.
6. *Aucsmith D.* Tamper Resistant Software: An Implementation // Proc. of the First Intern. Workshop on Information Hiding, 1996. - P. 317 – 333.
7. *Zeljko Vrba.* User-mode program tracing and an application to encrypted execution engine. - http://www.coredump.com.hr/documents/cryptexec.pdf.
8. *Loureiro S., Molva R.* Function Hiding Based on Error Correcting Codes // Proc. of Cryptec99 - Intern. Workshop on Cryptographic Techniques and Electronic Commerce, 1999. - P. 92 – 98.
9. *Sander T., Tschudin C.* On Software Protection via Function Hiding // Proc. of the Second Intern. Workshop on Information Hiding, 1998. — P. 111 – 123.
10. *On the (Im)possibility* of Obfuscating Programs / B. Barak, O. Goldreich, R. Impagliazzo et al // Proc. of the 21st Annual International Cryptology Conf. on Advances in Cryptology, 2001. - Vol. 2139. - P. 1 – 18.
11. *License* protection with a tamper-resistant token / C. Chong, B. Ren, J. Doumen et al // Proc. of 5th WISA, 2004. — P. 223 – 237.
12. *Zhang X., Gupta R.* Hiding Program Slices for Software Security // Proc. of the Intern. Symp. on Code generation and optimization: feedback-directed and runtime optimization, 2003. - Vol. 37. - P. 325 – 336.
13. *Blum M.* Program Result Checking: A New Approach to Making Programs More Reliable // Proc. of the 20th Intern. Colloquium on Automata, Languages and Programming, 1993. - P. 1–14.
14. *Spot-Checkers* / F. Ergün, S. Kannan, S. Kumar et al // Proc. of the 30th Annual ACM Symp. on Theory of Computing, 1998. - P. 259 – 268.
15. *Horne B., Matheson L., Sheehan C.* Dynamic Self-Checking Techniques for Improved Tamper Resistance // Proc. of the ACM CCS-8 Workshop on Security and Privacy in Digital Rights Management, 2001. - Vol. 2320. - P. 141 – 159.
16. *Segurson A.* Summary of Protecting Software Code by Guards. — www.cs.arizona.edu/~collberg/Teaching/620/2002/Papers/Segurson1.ps.
17. *Oblivious* Hashing: A Stealthy Software Integrity Verification Primitive / Y. Chen, R. Venkatesan, M. Cary et al // Proc. of the 5th Intern. Workshop on Information Hiding, 2002. - Vol. 2578. - P. 400 – 414.
18. *Lee B., Kim K.* Software Protection Using Public Key Infrastructure // Proc. of Symp. on Cryptography and Information Security, 1999. - P. 433 – 437.
19. *Jakobsson M., Reiter M.* Discouraging Software Piracy Using Software Aging // Proc. of the ACM CCS-8 Workshop on Security and Privacy in Digital Rights Management, 2002. - Vol. 2320. - P. 1 – 12.
20. *Collberg C., Thomborson C.* Watermarking, Tamper-proofing, and Obfuscation — Tools for Software Protection // IEEE Transactions on Software Engineering, 2002. - Vol. 28. - N. 8. - P. 735 – 746.
21. *Collberg C., Thomborson C.* Software Watermarking: Models and Dynamic Embeddings // Proc. of 26th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, 1998. - P. 311 – 324.

Получено 23.09.05

Об авторах

Анисимов Анатолий Васильевич
доктор физ.-мат. наук, профессор

Место работы автора
Киевский национальный университет
им. Т. Шевченко, декан факультета кибернетики
03680, Киев,
просп. Академика Глушкова, 2, корп. 6
Тел.: (044) 259 0129
E-mail: deanoffice@unicyb.kiev.ua

Иванов Иннокентий Юркевич
аспирант, инженер-программист

Место работы автора
ТОВ “Софтпанорама плюс плюс”.
03187, Киев,
ул. Академика Заболотного, 12, кв. 55.
Тел.: (044) 522 2957
E-mail: innokentiy@gmail.com