Архитектурный подход к защите программ

Николай Вяххи, Анатолий Коваль

Аннотация. Методы аппаратной («железной») защиты программ обычно более устойчивы к взлому, чем программная обфускация, потому что исследовать физическое устройство труднее и дороже, чем программу в уже известной архитектуре.

В данной статье рассмотрено совмещение двух перспективных направлений аппаратной защиты — eXecute Only Memory (ХОМ) и Dynamic Address Table (DAT). Они по отдельности хорошо сочетают в себе защищенность, прозрачность для программ и удобство использования. Но, главное, они дополняют друг друга, допуская совместное использование в одной архитектуре. Это позволяет защитить код программы от попыток его раскрытия какими либо средствами системы, что является шагом к достижению Black Box Security и избеганию многих атак, связанных с анализом исходного кода.

1. Введение

Каким образом можно защитить программу с помощью «железа» (то есть, не дать прямой доступ к исходному коду)? Можно попытаться «спрятать» некоторые данные или даже целые алгоритмы «внутри» аппаратуры. Но что значит спрятать? Разве программа, находясь в оперативной памяти (то есть «внутри» аппаратуры), не достаточно «спрятана»? На самом деле оперативная память, как и любой ресурс доступный приложениям, не является хорошим тайником, другие приложения могут читать код и данные из памяти нашего приложения (если наша архитектура предоставляет защиту приложений друг от друга, мы все равно не можем доверять операционной системе).

Далее в статье мы сначала рассмотрим общие положения архитектурного подхода и его особенности. Затем будут разобраны алгоритмы работы ХОМ и DAT. Во второй половине статьи будет предпринята попытка совместить ХОМ и DAT, исследовать полученную архитектуру на стойкость, оценить эффективность исполнения на ней программ, возможность дальнейших улучшений.

2. Архитектурный подход: общие положения

Неплохо было бы сделать новый компьютер для каждого нового приложения. Сделав его (с собственной уникальной архитектурой и системой команд) можно очень хорошо «спрятать» приложение, но такой подход противоречит принципу построения многоцелевых машин и стоит больших затрат.

Можно «сузить» решение следующим образом: разрабатывать не компьютер целиком, а только центральный процессор (который будет присоединяться к стандартным интерфейсам ввода/вывода). Идея продавать программно-аппаратные пакеты (software hardware package = SH-package) для защиты от незаконного тиражирования программ, была предложена в [Best79] и [Kent80], но, как можно заметить, не приобрела популярности за 20–25 лет существования. Само появление «железа» в программном проекте, существенно увеличивает стоимость и время разработки, стоимость конечного продукта (даже в том случае, когда аппаратура имеет очень маленькую функциональность). И понятно, что чем больше функциональности мы возложим на аппаратуру, тем больше будет стоить наш продукт.

Итак, существует два подхода архитектурной защиты программ:

- **реализовывать программу аппаратно (в железе)** — дорог в разработке и распространении, но более безопасен, ведь исследовать цельный процессор и понять, как он работает, очень трудоемко и ресурсоемко.
- **реализовывать ее программно (имея в железе универсальный механизм защиты любых возможных программ)** — дешевизна разработки нового ПО, дешевизна самого устройства (из-за большого количества одинаковых устройств для различных задач), но возможны проблемы с безопасностью при передаче данных и, так как устройство одинаково для всех программ, то его взлом более вероятен.

Мы рассмотрим два вида защиты, принадлежащие второй группе подходов, следовательно являющиеся более доступными и перспективными. При этом, они хорошо дополняют друг друга, позволяя повысить надежность системы.

Но сначала зададимся вопросом, зачем же аппаратная защита нужна, если есть обычная, программная?

Программная обфускация пока что обладает только такими эвристическими методами, как запутывания потока, переименования переменных и т.п. Криптография может использоваться только очень ограниченно. В то время как аппаратная защита способна использовать криптографию в полном объеме, потому что может размещать важные блоки в сложно-доступном для взлома месте — в железе. В целом, можно сделать такой вывод:

Преимущества:

- Можно оценить степень защиты (в отличие от полностью программных подходов, где на данный момент нет удовлетворительных результатов в области оценок защиты).
- Как правило цена взлома может быть напрямую выражена в затратах на покупку специальных дорогостоящих устройств или затратах при использовании лабораторий.
- Существующие решения, полностью исключают внутренние методы атаки, то есть со стороны других программ или операционной системы.

Недостатки:

- Сложность разработки высока (как следствие высока стоимость конечного устройства и стоимость разработки).
- Ограничение на распространение (некоторые программно-аппаратные структуры могут применяться только во встроенных системах, другие не поддерживают многозадачность).
- При наличии соответствующей аппаратуры можно взломать любую программно-аппаратную систему.

Необходимо отметить существование работ направленных на устранение последнего недостатка (возможность взлома сколь угодно сложной системы). Например, статья [Yuval03], рассматривает интегральную схему, как набор элементарных логических единиц (вентилей И, НЕ) и соединительных линий. Далее определяется t -adversary — противник, который на каждом такте может наблюдать не более t линий, соединяющих элементарные логические единицы. В статье приводится алгоритм запутывания, работающий против t -adversary (алгоритм зависит от t).

Качество алгоритма оценивается числом дополнительных вентиляей, добавляемых в микросхему. Необходимо заметить, что число дополнительных логических единиц довольно велико (как минимум линейно относительно произведения числа вентиляей в исходной микросхеме на t).

3. eXecute Only Memory

ХОМ (eXecute Only Memory) — модель (настройка над процессором), где программа выполняется, но не может глубоко анализироваться даже из операционной системы и каким-либо программным путем. Обладает свойствами, которые мы опишем ниже. Поддерживает криптографию на аппаратном уровне, допускает многозадачные среды.

ХОМ не является целым вычислительным устройством, следовательно его разработка не настолько дорога, как разработка целого процессора. Также, возможна его интеграция с уже существующими процессами, хотя ни одной известной практической реализации ХОМ еще нет.

Достаточно полное описание ХОМ можно прочитать в [Lie00] и [Lie04]. Тут мы остановимся на основных принципах.

3.1. Основные принципы ХОМ

ХОМ являясь настройкой над процессором, читает пошагово программу и передает ее на исполнение, обеспечивая нужный уровень защиты исходного кода и данных в памяти. Рассмотрим все его действия поподробнее.

Каждый ХОМ-процессор имеет свой **открытый и закрытый ключи**, которые присваиваются ему при изготовлении. Все **инструкции программы изначально (автором) зашифрованы** открытым ключом ХОМ'а (несущественные для защиты куски могут быть не зашифрованы вообще). Хотя, для эффективности открытым ключом можно шифровать и только начальный блок программы, содержащий симметричный ключ (compartment key), которым шифруется оставшаяся часть (симметричное шифрование намного быстрее). **Никто не может расшифровать программу не зная закрытого ключа.**

ХОМ расшифровывает нужную инструкцию и посылает ее на исполнение процессору. Далее расшифровывается следующая необходимая инструкция и так далее... Но, так как программа постоянно должна оперировать с данными, а не только исполнять свой код, то встает проблема **сохранения этих данных в незащищенную область памяти**. Это решается

также с помощью стандартных методов криптографии. Данные, записываясь в память, шифруются, подписываются вместе с адресом ячейки и идентификатором исполняемого процесса (определяемого по compartment key). При чтении данных из памяти программой, они расшифровываются, их идентификаторы сравниваются, адрес ячейки проверяется. Если что-то не совпадает, то программа-читатель блокируется. Никакой другой процесс не может расшифровать данные или скопировать их в другую ячейку, не зная уникального идентификатора.

Также нужно уметь **сохранять регистры**, чтобы поддерживать переключение контекста (context switch). Это делается по аналогии с хранением данных в памяти: **все регистры** подписываются вместе с идентификатором процесса, **шифруются** и записываются в память. А при переключении обратно к этому процессу, регистры расшифровываются и проводятся нужные проверки правильности. Против атаки повторениями (replay attack) **используются постоянно сменяющиеся ключи шифровки регистров** — последний используемый ключ запоминается для каждого процесса. Против этой же атаки над данными, записываемыми в память, надежной защиты нет, так как там генерировать каждый раз новый ключ не удастся. Ведь данные используются постоянно, а не один раз записали все сразу — один раз прочитали, как регистры.

ХОМ — криптографическая система с доказуемой секретностью, в отличие от многих программных методов обфускации. Ниже в статье, после рассмотрения DAT, будет дан более развернутый вывод про оба алгоритма сразу.

3.2. Критика

ХОМ предоставляет защиту, близкую к black box security, потому что полностью скрывает исходный код программы, не предоставляя возможностей для его статического анализа. То есть, без низкоуровневого вмешательства в ХОМ невозможно расшифровать код программы и ее данные (естественно, при стандартных криптографических предположениях).

Конечно, на практике достаточно будет сломать одну машину (аппаратно), узнать ее секретный ключ и расшифровывать программы под нее. После этого опубликовывать взломанные программы в открытом виде. То есть, тут необходим некий централизованный контроль за взломанными машинами.

Например, использование водяных знаков в программе — для какой именно машины она поставлялась. И если эта программа публикуется в открытом виде, то соответствующий ХОМ-процессор оказывается в черном списке для разработчиков ПО и для него больше не поставляются программы.

Поэтому, остается открытым вопрос, как раздавать ключи ХОМ-ам.

Недостатки:

- каждому свой — очень трудоемко и неудобно создателям ПО, сложности с защитой;
- всем один — никакой выгоды в защите;
- зонально (как DVD), партиями и т.д.

4. Динамическая таблица адресов

Опираясь на криптографию, мы получили систему, полностью защищенную от внутренних атак и некоторых внешних атак наблюдения (например, получение информации из памяти). Тем не менее, осталась еще одна уязвимость, которой можно воспользоваться, не прилагая особых денежных усилий. Не смотря на то, что вне процессора все данные шифруются, адреса памяти, по которым расположены эти данные, остаются теми же. Иными словами, взломщик может следить за потокам управления и потоком данных, отслеживая адреса появляющиеся на шине. Исследование последовательности адресов может дать достаточно подробную структуру о графе потока управления (выдать структуру программного обеспечения). Именно этой проблеме посвящена статья [Xiatong04]. Рассмотрим эту проблему поподробнее:

4.1. Утечки

Определение 1. *Layout leakage* — получение данных о линейных участках кода благодаря принципу последовательного выполнения (две последовательные команды имеют два последовательных адреса).

Определение 2. *Recurrence leakage* — получение данных за счет принципа обращения к памяти (для двух обращений, в процессе выполнения, к одной и той же переменной на шине дважды появляется один и тот же адрес), с помощью этого типа «утечек» можно восстановить структуру циклов, уловных выражений.

4.2. Статическая таблица адресов

Решения для первого типа утечек в следующем: достаточно сделать в процессоре статическую таблицу трансляции адресов (SAT static address table). SAT логически располагается между шиной и процессором, так, что любой адрес памяти который процессор выдает на шину проходит через преобразование SAT. Статическая таблица трансляции — абсолютно прозрачная структура, логически представляющая перестановку адресов памяти. SAT можно представить себе как виртуальную память (где любой линейный адрес проходит прозрачно через страничное преобразование и выдается на шину), с той разницей, что SAT статична (задается при производстве процессора и не может быть изменена в дальнейшем). Необходимо отметить, что при таком решении возникают трудности для разработчиков ПО. А именно, разработчик должен «знать», каким образом записать ПО в память («знать» в том смысле, что либо он должен знать функцию перестановки, которую реализует SAT, либо ему должно быть предоставлено устройство, которое запишет приложение по правильным адресам в физической памяти). Начальная запись кода и данных по правильным физическим адресам — единственная проблема, после того как приложение записано в память оно работает обычным образом (фактически можно пользоваться прежними средами разработки, правильно записанное приложение не «заметит» SAT).

Существуют некоторые промышленные чипы, реализующие этот подход (например, DS5000, DS5002FP производства Dallas Semiconductor описаны в [Dallas]). Авторы статьи [Xiatong04] отмечают слабость такой защиты (информация о возвратах к одному и тому же адресу по-прежнему легко доступна). Таким образом, основной интерес для нас, представляет борьба с recurrence leakage...

4.3. Динамическая таблица адресов

Идеи измененного алгоритма: дополнить каждое чтение из памяти записью в память (теперь взломщик будет почти всегда видеть пары циклов чтение/запись), заменить SAT на DAT (dynamic address table — в некотором смысле виртуальная память, так как теперь позволено менять таблицу трансляции), пытаться перемешивать блоки памяти на протяжении работы процессора (при этом корректируя DAT).

Итак, помимо таблицы адресов (изначально тождественной) в системе присутствует небольшая буферная память, которая дополняет основную (Shuffle Buffer). То есть блок может находиться либо в нем, либо в оперативной памяти (но не одновременно в двух местах). Алгоритм делится на две стадии:

- **стадия разгона**
- **стадия работы**

На стадии разгона происходит линейное заполнение Shuffle Buffer прочитанными блоками (в DAT помечается, тот факт, что блок находится в Shuffle Buffer, а не в памяти). На стадии работы для любого прочитанного блока происходит случайный выбор блока из буфера, который меняется с ним местами (из буфера записывается в основную память и наоборот). Далее происходит корректировка DAT.

Авторы DAT (Xiatong Zhuang, Tao Zhang, Hsien-Hsin S. Lee, Santosh Pande) приводят вероятность $1/M^n$ для обнаружения атаки n-recurrence (обнаружение n повторений одной команды), где M — количество блоков, находящихся одновременно в Shuffle Buffer.

4.4. Критика

DAT-таблица и буфер по своим размерам не могут полностью содержаться внутри процессора, тогда часть таблицы будет храниться в памяти (авторы рассматривают такой вариант, но не рассматривают как это отразится на «утечках» информации, ведь часть таблицы DAT находящаяся в памяти не будет проходить через Shuffle Buffer). То есть необходимо усовершенствование таблицы и методов ее хранения. Хотя, если оценка $1/M^n$ правдоподобна, то тогда достаточно буфера из всего 512-1024 команд, чтобы даже обнаружение 2-recurrence уже было маловероятно.

5. XOM + DAT

Попробуем совместить два описанных подхода и оценить получаемое преимущество.

Архитектура XOM гарантирует стойкость против попыток узнать что именно исполняется на процессоре, в то время как DAT против исследования потока управления программы. То есть, они могут гармонично дополнять друг друга, если не будут конфликтовать.

Если расположить Shuffle Buffer между ядром процессора и ХОМ-надстройкой, тогда данные, содержащиеся в нем не будут шифроваться, что может быть плохо для стойкости. Следовательно, разместим буфер после всего процессора. Динамическая таблица адресов будет располагаться, как и в классической модели, прямо в процессоре. Тогда, жизненный цикл программы будет выглядеть так. Сначала с помощью закрытого ключа ХОМ-процессора расшифровывается compartment key программы, ей присваивается ХОМ ID. Затем, программа последовательно исполняется, причем при обращении в память адреса транслируются с помощью DAT, и во время исполнения некоторые ячейки в памяти меняются с ячейками в Shuffle Buffer в случайном порядке. То есть, эти две архитектуры работают прозрачно для друг друга, не замечая каких-то изменений.

Рассмотрим стойкость против взлома. ХОМ нам гарантирует секретность выполняемых процессором инструкций, DAT — секретность потока исполнения. То есть, если раньше (при использовании одного метода) можно было узнав одно (например поток исполнения), с помощью него узнать другое (например что исполняется — зная некоторые стандартные библиотечные последовательности инструкций для функций, можно их обнаружить расположенными подряд в памяти). То теперь это исключено, и программа будет полноценно защищена на архитектурном уровне. В результате сложность взлома сильно возрастает, а цена изготовления устройства является лишь суммой цены ХОМ и DAT.

Попробуем применить атаку пошаговым исполнением к полученной системе. Например, мы исполняем программу до какой-нибудь явно известной и обнаруживаемой инструкции (например вывести символ на экран). При этом мы узнаем, как выглядит зашифрованная инструкция вывода и где хранятся интересные нам данные в памяти. Дает ли это нам хоть какую-то информацию о программе, ее ключе, содержимом таблицы адресов или буфера? Очевидно, что если и дает, то намного меньше чем при программной обфускации. Из-за применения в архитектуре стандартных криптографических алгоритмов (как симметричных, так и несимметричных) полученная информация никак не поможет нам расшифровать или понять оставшуюся часть программы.

6. Заключение

По сравнению с программной обфускацией на нынешнем уровне, аппаратная защита более перспективна. Обфускация пока предоставляет только практические приемы, как, например, запутывание потока, переименование переменных и т.п., что не является гарантией защищенности программ. И, если необходимо узнать, где в программе выполняется определенный кусок, это в большинстве случаев можно сделать несложным пошаговым исполнением инструкций. Аппаратные же методы дают гарантированную защиту, а в сочетании ХОМ + DAT и поток, и данные оказываются надежно скрыты.

Литература

- [Best79] *Best R* "Microprocessor for Executing Encrypted Programs"US Patent 4,168396 September 1979.
- [Kent80] *Kent S.T* "Protecting Externally Supplied Software in Small Computers"Ph.D. Thesis MIT/LCS/TR-255 1980.
- [Yuval03] *Yuval Ishai, Amit Sahai, David Wagner* "Private Circuits: Securing Hardware Against Probbing Attacks"2003.
- [Dallas] "DS5002FP Secure microprocessor chip data sheet", Dallas Semiconductor.
- [Lie00] *D.Lie, C.A.Thekkath, M.Mitchell, P.Lincolny, D.Boneh, J.Mitchell, M.Horowitz* "Architectural Support for Copy and Tamper Resistant Software", 2000.
- [Lie04] *D.Lie, J.Mitchell, C.A.Thekkath, M.Horowitz* "Specifying and Verifying Hardware for Tamper-Resistant Software", 2004.
- [Xiatong04] *Xiatong Zhuang, Tao Zhang, Hsien-Hsin S. Lee, Santosh Pande* "Hardware Assisted Control Flow Obfuscation for Embedded Processors", 2004.