УДК 004.056

Самофалов К.Г., чл.-кор. НАН Украины
Марковский А.П., канд. техн. наук
Зюзя А.А.
Льозин А.С.

СТОХАСТИЧЕСКИ ПОЛИМОРФНАЯ РЕАЛИЗАЦИЯ АЛГОРИТМА RIJNDAEL НА МИКРОКОНТРОЛЛЕРАХ И СМАРТ-КАРТАХ

Национальный технический университет Украины "КПИ"

Разработан способ полиморфной реализации алгоритма Rijndael на микроконтроллерах и смарт-картах. Выполнен анализ характеристик алгоритма и его базовых операций с точки зрения их полиморфной реализации. Доказано, что полиморфизм операций ограничен двумя смежными циклами. Предложен и исследован способ полиморфной реализации на основе случайного выбора последовательности программных секций. Показано, что вариация команд не превышает 85%. Предложенный способ позволяет увеличить защищенность ключей алгоритма Rijndael от дифференциального анализа потребляемой мощности

Введение

Расширение интеграции информационных ресурсов на основе компьютерных сетей является определяющим фактором прогресса современных технологий обработки данных. Информационная интеграция обеспечивает качественно новый уровень эффективности применения компьютерных технологий во всех сферах человеческой деятельности. Важным элементом современных сетевых технологий являются терминальные устройства, осуществляющие функции непосредственного сбора информации и реализующие управления объектами. При этом, терминальные устройства должны в полной мере реализовать установленные протоколы сетевого обмена, в том числе, протоколы защиты информации. Для широкого круга применений компьютерных сетей с развитым терминальным оборудованием задачи защиты информации являются актуальными. В первую очередь, это относится к банковским сетям с развитой структурой банкоматов, систем сетевого мониторинга состояния производств и транспортных средств. Вместе с тем, в настоящее время, до 60% терминальный устройств компьютерных сетей составляют встроенные контроллеры и смарт-карты [1]. Это компактные и относительно простые вычислительные устройства, работающие в однопрограммном режиме. Для таких терминальный сетевых устройств существует возможность реконструкции постоянно используемых данных посредством измерения и анализа динамики потребления мощности (АДПМ) [2]. В особенности это касается закрытых элементов (ключей и паролей) сетевых протоколов защиты данных. В настоящее время одним из наиболее перспективных алгоритмов защиты данных является алгоритм *Rijndael* созданный в рамках проекта *AES* [3]. Соответственно при реализации защиты информации в сетях с развитым терминальным оборудованием возникает задача противодействия реконструкции ключей алгоритма *Rijndael* анализом динамики мощности, потребляемой терминальным микроконтроллером при выполнении этого алгоритма. Одним из наиболее эффективных способов решения этой задачи является полиморфная реализация вычислений.

Таким образом, задача эффективной организации полиморфной реализации алгоритма *Rijndael* на терминальных микроконтроллерах сетей является актуальной и имеющей практическую значимость.

Способы противодействия АДПМ

Наиболее распространенной формой реконструкции постоянно используемых в программе кодов (к которым относятся ключи и пароли) на основе измерения параметров технической реализации является анализ динамики потребляемой мощности *(power analysis*) [4]. Различают две его разновидности: простой анализ мощности (*Simple power analysis – SPA*) и дифференциальный анализ мощности (*Differential power analysis – DPA*). Современная измерительная техника позволяет осуществить сопоставление выполняемым командам потребляемой при этом мощности.

Величина тока, потребляемого микроконтроллером в момент выполнения команды, зависит от типа команды и от значения отдельных битов операндов. Следовательно, по величине мощности, затрачиваемой при выполнении команды можно косвенно судить об типе команды и о кодах операндов. Этот вид реконструирования данных эффективен при условии наличия достаточно полной информации о микропроцессоре и о программе, которая реализует этот алгоритм. С использованием *SPA* достаточно просто установить значение бита, по которому осуществляется выполнение условного перехода.

Использование *DPA* предполагает проведение статистической обработки ряда измерений мощности в некотором подмножестве точек выполнения алгоритма при условии, что часть данных не меняется.

Для противодействия *SPA* и *DPA* применяют ряд технологий [3], которые можно разделить на две группы. Первая из них объединяет способы специального выполнения программ, которые затрудняют проведение АДПМ, а вторая – основана на введение при выполнении программы элементов случайности, что снижет, эффективность статистических методов обработки.

К первой группе можно отнести:

– замену условных переходов, зависящих от значения бита другими командами;

– применение специальных кодов данных при программной реализации алгоритмов (использование кодов с равным числом нулей и единиц).

Ко второй группе относятся:

– случайное изменение последовательности выполнения команд при каждой реализации программы (полиморфизм программного кода), если это допускается логикой алгоритма;

– маскирование значений постоянно используемых данных случайными кодами, которые изменяются при каждом запуске программы;

– случайная вставка команд, которые не меняют данных, но которая не затрудняет "привязку" точки измерения мощности во времени к конкретному шагу выполнения алгоритма;

– случайное выполнение фрагмента программы по одному из нескольких программных шаблонов, использующих различные команды.

Указанные способы противодействия технологиям *SPA* и *DPA* могут применяться комбинированно. Их использование, в той или иной мере, сопряжено с увеличением вычислительных ресурсов, затрачиваемых на реализацию алгоритма.

Маскирование является наиболее простым способом противодействия АДПМ, хотя существует сложности со "снятием" маски после окончания алгоритма для получения правильного результата. Однако в последние годы появилась разновидность статической обработки АДПМ – дифференциальный анализ второго порядка [4], использование которого снижает эффективность маскирования.

Эффективность стохастического полиморфизма программной реализации зависит от величины вариации времени выполнения каждой из команд и затрачиваемых для этого ресурсов. Исследования [88] показали, что решение задачи полиморфной реализации алгоритма в общем случае является достаточ-

но сложной задачей, связанной с трудоемким анализом графа зависимости выполняемых операций по данным. Поэтому практическое применение полиморфизма, как средства противодействия АДПМ, ограничивается значительным объемом затрачиваемых ресурсов, существенно большим, чем при использовании других известных способов.

Вместе с тем, следует учесть, что протоколов и алгоритмов, использующих постоянную ключевую информацию относительно невелико. Поэтому обоснованным представляется разработка способа эффективной реализации для каждого алгоритма в рамках общих принципов. Это позволяет учесть в полной мере особенности вычислительных процедур, положенных в основу алгоритмов и получить эффективное решение полиморфной реализации, соизмеримое по затратам ресурсов с другими способами.

Целью работы является исследование алгоритма *Rijndael* с точки зрения возможности стохастического реконфигурирования вычислений и разработка способа его эффективной полиморфной реализации на терминальных устройствах сетей – микроконтроллерах и смарт-картах.

Анализ вычислительных процедур алгоритма Rijndael

При полиморфной реализации алгоритма необходимо обеспечить сложность временной локализации команд, которая практически исключала возможность их сопоставления диаграмме потребления мощности при возможно меньших затратах вычислительных ресурсов на осуществление полиморфизма.

Для осуществления полиморфной реализации алгоритма необходимо:

– выполнить анализ зависимости выполняемого потока команд по данным;

– с учетом означенной зависимости стохастически генерировать последовательность выполнения команд.

Для симметричных блочных алгоритмов, к классу которых относится Rijndael зависимость команд по данным устанавливается достаточно просто и однозначно, поскольку указанный класс алгоритмов не содержит условных операторов.

Основной характеристикой вариации момента времени выполнения i-той команды ($i \in \{1,..,N\}$, где N-общее число команд) является интервал v_i времени между поздним t_{li} и ранним t_{ei} временами ее выполнения: $v_i = t_{li} - t_{ei}$. Обобщенной характеристикой временных вариаций выполнения алгоритма является: средняя вариация T_a момента выполнения команд и минимальное время T_m вариации момента выполнения команд.

$$T_a = \frac{1}{N} \cdot \sum_{i=1}^{N} v_i \,,$$

$$T_m = \min_{i=1,..,N} v_i \,.$$

Для анализа возможностей изменения времени выполнения команд, целесообразным представляется кратко рассмотреть организацию выполнения алгоритма *Rijndael* на микроконтроллерах и смарт-картах. Для такого класса вычислительных устройств команды имеют, в большинстве случаев, формат байта. Поскольку прямое и обратное преобразования в алгоритме *Rijndael* схожи, представляется оправданным подробно исследовать только организацию прямого преобразования. Выполнение обратного преобразования может быть организовано аналогично.

Алгоритм *Rijndael* ориентирован на обработку блоков данных длиной 128, 192 или 256 бит. Байты обрабатываемого блока M организованы в матрицу состояний, состоящую из 4-х строк и 4, 6 или 8 столбцов в зависимости от длины блока. Длина ключа, выбираемая независимо от длины блока данных, также может быть равна 128, 192 или 256 бит. Выполнение алгоритма включает 10, 12 или 14 циклов, количество которых

регламентируется выбором длин информационного блока и ключа. Из инициирующего блока ключей независимой процедурой формируется необходимое число циклических ключей, используемых на каждом из циклов.

Основной цикл алгоритма *Rijndael* составляют операции *XOR* с ключом (*XK-XOR with Key*), замещения байта (*BS-Byte Substituted*) (эти операции выполняются независимо для всех байтов матрицы текущего состояния), циклического сдвига строк (*LR-Lines Rotation*), перемешивания столбцов (*RM-Rows Mix*). Операция сдвига строк может быть исключена при условии, что перемешивание столбцов заменяется перемешиванием диагоналей (*DM-Diagonal Mix*) с записью результата в столбцы новой матрицы. С учетом этого, вычислительная процедура *Rijndael* состоит из 10 идентичных циклов и заключительного преобразования, как это показано на рис. 1. Последнее имеет две отличительные особенности: выполнение сдвига строк вместо перемешивания диагоналей и наличие финишного операции XOR с ключами.

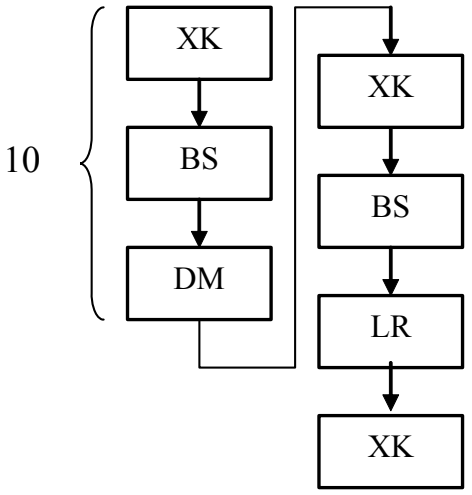

Рис. 1.

В алгоритме *Rijndael* используются четыре базовых операции: прямое и инверсное байтовое замещение, циклический сдвиг строк матрицы состояний, перемешивание столбцов, суммирование по модулю 2 текущей матрицы состояний с соответствующей матрицей расширения ключа.

Операция замещения байта (*BS*) при прямом преобразовании выполняемая над байтом G используется операция получения мультипликативной инверсии $R=G^{-1}\bmod p(x)$ байта на поле Галуа, образованным полиномом $p(x)=x^8+x^3+x+1$, с последующим линейным преобразованием полученной инверсии R путем умножения ее на фиксированную матрицу M_d и суммированием по модулю 2 с фиксированным байтом $B=\{01100011\}$: $D=S(G)=(M_d\times R)\oplus B$. Соответственно, обратное замещение байта D состоит в выполнении обратного линейного преобразования с использованием матрицы M_r: $G=S^{-1}(D)=(M_r\times(D\oplus B))^{-1}\bmod p(x)$.

Описанная операция байтового замещения, представляющая собой систему нелинейных булевых функций от 8-ми переменных в большинстве программных реализаций осуществляется с использованием заранее просчитанных таблиц объемом 256 байт (8×8 бит), сохраняемых в постоянной памяти [4]. Для реализации прямого и обратного преобразований требуется две таблицы.

Операция перемешивания байтов диагоналей матрицы состояний (*DM*) матрицы состояний состоит в линейном преобразовании $Y=P(X)$ байтов диагонали $X=\{x_0,x_1,x_2,x_3\}$:

$$y_0 = 2\cdot x_0 \oplus 3\cdot x_1 \oplus x_2 \oplus x_3$$
$$y_1 = x_0 \oplus 2\cdot x_1 \oplus 3\cdot x_2 \oplus x_3$$
$$y_2 = x_0 \oplus x_1 \oplus 2\cdot x_2 \oplus 3\cdot x_3$$
$$y_3 = 3\cdot x_0 \oplus x_1 \oplus x_2 \oplus 2\cdot x_3$$

где $Y=\{y_0,\ldots,y_3\}$ – выходной столбец матрицы состояний, y_0, y_1, y_2, y_3 – байты его составляющие, а операции умножения 8-разрядных кодов выполняются по правилам умножения на полях Галуа с базовым полиномом $x^8+x^4+x^2+x+1$. Операция восстановления $X=P^{-1}(Y)$ байтов диагонали $X=\{x_0,x_1,x_2,x_3\}$ по коду

$Y=\{y_0,\ldots,y_3\}$ состоит в линейном преобразовании:

$$x_0 = 0Eh \cdot y_0 + 0Bh \cdot y_1 + 0Dh \cdot y_2 + 09 \cdot y_3$$
$$x_1 = 09 \cdot y_0 + 0Eh \cdot y_1 + 0Bh \cdot y_2 + 0Dh \cdot y_3$$
$$x_2 = 0Dh \cdot y_0 + 09 \cdot y_1 + 0Eh \cdot y_2 + 0Bh \cdot y_3$$
$$x_3 = 0Bh \cdot y_0 + 0Dh \cdot y_1 + 09 \cdot y_2 + 0Eh \cdot y_3$$

Таким образом, каждый из циклов состоит из 4-х независимых блоков обработки диагоналей. Блок обработки диагонали включает операции, производимые над 4-мя байтами диагонали матрицы состояний. Структура этих операций показана на рис. 2. На этом рисунке через M2 и M3 обозначены операции умножения байта на 2 и 3 соответственно. Эти операции выполняются по правилам умножения на полях Галуа с базовым полиномом $x^8+x^4+x^2+x+1$.

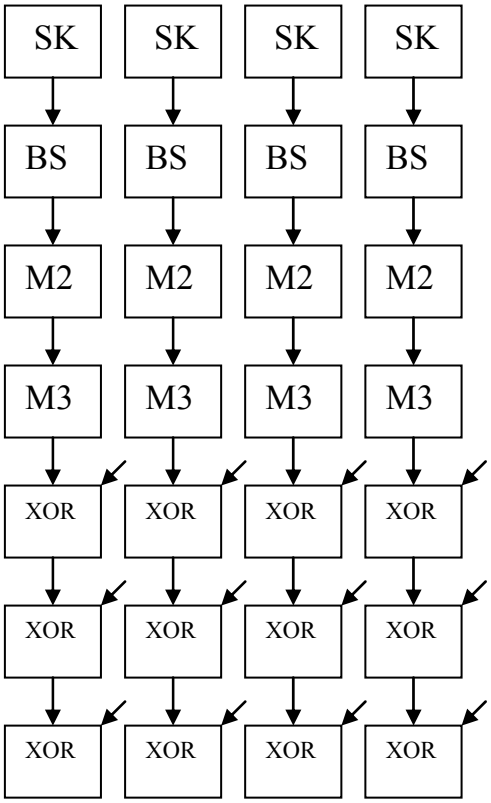

Рис. 2.

Финальные операции *XOR* реализуют перемешивание байтов диагонали и приведенные на рис. 2 стрелочки к операциям XOR указывают на то, что для получения результирующего байта необходимо сложить по модулю 2 все байты диагонали непосредственно или их произведение на 2 и 3. Операции умножения на 2 и 3 могут выполняться либо с использованием операций сдвигов, условных переходов по биту C переноса и операций *XOR*, либо с применением табличной памяти.

Общее число операций, составляющих блок равно 28-ми. Если принять, что среднее время выполнения одной команды равно τ, то максимальная вариация v времени выполнения команды *XK* составляет $3\cdot6\cdot\tau=18\cdot\tau$. Действительно, позднее время выполнения этой команды *XK* для i-того байта диагонали, $i\in\{1,\ldots,4\}$, определяется временем обработки 3-х других байтов диагонали без конечной операции *XOR*, которая использует в качестве операнда результаты обработки i-го байта.

Обработка блоков в рамках одного цикла алгоритма *Rijndael* осуществляется независимо. Однако, при переходе к следующему циклу возникает необходимость в обмене результатами обработки блоков. Структура зависимостей по данным блоков двух смежных циклов алгоритма *Rijndael* показана на рис. 3. Как следует из рис. 3, по готовности результатов одного блока j-го цикла алгоритма может быть начата обработка по одному из байтов каждого из блоков следующего, $(j+1)$-го цикла.

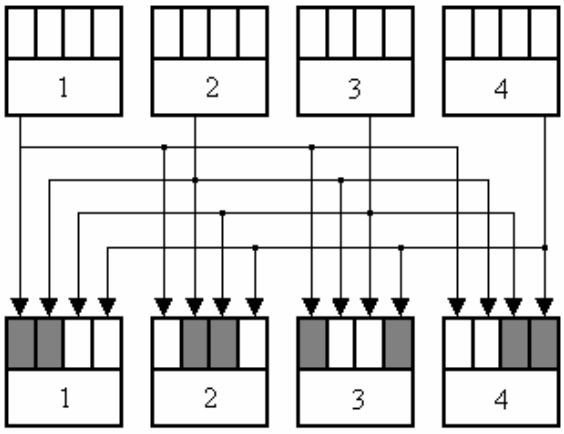

Рис.3.

Это открывает возможность совмещения (с разделением во времени) выполнения блоков j-го и $(j+1)$-го циклов. Блоки

(j+1)-го цикла не могут быть полностью выполнены до окончания обработки всех блоков предыдущего цикла. Из этого следует, что существует возможность совмещения обработки блоков только двух смежных циклов.

Таким образом, проведенный анализ показал, что при полиморфной реализации алгоритма *Rijndael* момент выполнения команд может быть сдвинут на величину, не превышающая время выполнения двух циклов преобразования.

Для воплощения этой возможности с возможно меньшими затратами вычислительных ресурсов необходимо разработать соответствующий способ стохастически полиморфной программной реализации алгоритма.

Стохастическое реконфигурирование с использованием наборов секций

Для реализации стохастической полиморфии при выполнения j-го и (j+1)-го циклов необходимо предусмотреть механизм прерывания обработки блоков (j+1)-го цикла до готовности результатов обработки всех блоков j-го цикла. В качестве такого механизма могут быть использованы как обычные программные прерывания, так и компоновка выполнения блока из нескольких последовательно запускаемых секций.

Учитывая, что предлагаемый способ выполнения алгоритма *Rijndael* ориентирован на простые вычислительные платформы, такие как микроконтроллеры и смарт-карты, предпочтительным является второй вариант.

Для анализа технологии выполнения блока алгоритма *Rijndael* в виде последовательности секций целесообразно ввести кодировку каждой из них в виде тетрады (по числу обрабатываемых байтов) символов. Каждый i-тый, $i \in \{1,..,4\}$, из символов может принимать три значения: 0 - i-тый байт не готов к обработке, 1- i-тый байт готов к обработке, 2 - i-тый байт обработан. Тогда, например, если закончен 1-й блок j-го цикла, то может начаться выполнение секции <1,0,0,0> любого из блоков (j+1)-го цикла. Возможным вариантом выполнения, например, 2-го блока (j+1)-го цикла может быть следующий: по готовности 1-го блока выполняется секция <1,0,0,0>. После этого выполнение 2-го блока приостанавливается до готовности одного из блоков j-го цикла. Продолжение реализации 2-го блока (j+1)-го цикла может осуществляется выполнением секции <2,0,1,0> по готовности результатов 3-го блока j-го цикла. Затем выполнение блока 2 (j+1)-го цикла прерывается до готовности очередного блоков j-го цикла. В это время могут обрабатываться готовые к выполнению секции блоков обоих циклов, выбираемых случайным образом. Пусть в результате такого выбора инициализация блока 2 (j+1)-го цикла выполнится когда 2-й и 4-й блоки предыдущего цикла обработаны. Тогда завершение обработки блока осуществляется выполнением секции <2,1,2,1>.

Очевидно, что общее количество секций определяется количеством различных 4-символьных кодов, содержащих хотя бы одну единицу, то есть $4 \cdot 2^3 + 6 \cdot 2^2 + 4 \cdot 2 + 1 = 65$. Наличие столь значительного числа секций блока требует для их хранения много памяти.

Значительная часть от общего числа секций не допускает полиморфной реализации и, в силу этого, их использование не дает должного эффекта. Так, последовательность команд в секциях, код которых включает только одну единицу, вполне однозначна и не может быть изменена в силу того, что каждая последующая команда зависит по данным от предыдущей. Стартовые секции, код которых содержит только одну единицу (<1000>,<0100>,<0010> и <0001>) содержат малое число команд (от 4-х до 7-ми). Соответственно, их использование имеет следствием увеличение числа прерываний при исполнении блока, каждое из которых

связано с дополнительными затратами времени на формирование (корректировку) списка готовых к выполнению секций и случайный выбор одной из них.

Таким образом, наиболее целесообразным представляется использование в качестве стартовых секции, коды которых содержат две единицы и остальные нули. Тогда и в качестве финишных секций представляется обоснованным использование секций, код которых содержит две единицы. Проведенные экспериментальные исследования подтвердили, что наиболее эффективным вариантом секционирования блоков алгоритма *Rijndael* является использования только секций, код которых содержит 2 единицы. Число таких блоков равно 6·2=12. Соответственно, при реализации блока выполняются только две секции: стартовая, код которой включает два нуля и финишная, с кодом, содержащим два символа ‘2’. Всего существует 6 вариантов стартовых секций и 6 вариантов финишных секций. Стартовые секции включают в себя 10 команд, а финишные – 18. За счет перестановок команд, не зависимых по данным, для каждой из 6-ти стартовых секций может быть образовано 140 ее различных вариантов, а для каждой из 6-ти финишных – 4352 вариантов.

Организация стохастически полиморфной реализации циклов

Для организации стохастически полиморфного выполнения алгоритма Rijndael предлагается использовать такие информационные структуры:

1. Набор секций**,** каждая из которых представляет собой программный модуль. Данные для работы секции зависят от номера блока и задаются базовым адресом байтов диагонали, обрабатываемого блоком.

2. Список R блоков, готовых к обработке. Каждый из элементов списка содержит пять компонент: номер N_c цикла (нуль или единица), номер n_b блока (от 1 до 4-х), код c_s секции, базовый адрес A_s варианта программной реализации секции, базовый адрес A_d обрабатываемой тетрады байтов. С использованием указанного базового адреса сохраняются и результаты работы секции.

3. Код $S_0=\{S_{01},S_{02},S_{03},S_{04}\}$ состояния блоков первого цикла и код $S_1=\{S_{11},S_{12},S_{13},S_{14}\}$ состояния блоков второго цикла. Каждая из 4-х компонент кода состояния характеризует состояние соответствующего блока и принимает пять возможных значений: 0 – блок не готов к выполнению; 1 – блок включен в список для обработки стартовой секции; 2 – обработка блока закончена стартовой секцией, но блок не готов к запуску финишной секции; 3 – блок находится в очереди для обработки финишной секцией; 4 – обработка блока завершена.

4. Список Z блоков второго цикла, обработанных стартовой секцией, но не готовых к обработке второй секцией. Каждый из элементов списка Z содержит номер n_b блока (от 1 до 4-х), код c_s стартовой секции, при помощи которой реализована стартовая обработка и базовый адрес A_d обрабатываемой тетрады байтов.

Для обработки циклов алгоритма *Rijndael* в исходном состоянии в список R готовых к обработке блоков включаются все блоки 1-го цикла. При этом коды секций выбираются случайным образом из 6-ти возможных стартовых. По выбранному коду секции случайным образом выбирается один из u вариантов ее реализации и в список помещается соответствующий адрес A_s начала программного модуля. Соответственно, все компоненты кода состояния первого цикла устанавливаются в единицу: $S_0=\{1,1,1,1\}$, а второго цикла – в нуль: $S_1=\{0,0,0,0\}$. Счетчик выполненных циклов C устанавливается в нуль.

Обработка блока данных выполняется в следующей последовательности:

1. Случайным образом выбирается элемент списка R блоков, готовых к

обработке. По коду A_s выполняется переход на выполнение выбранного варианта секции обработки блока, базовый адрес данных для которой задаются компонентой A_d. Выбранный элемент исключается из списка R.

2. По завершению работы программного модуля секции выполняется анализ ее кода и кода состояния обработанного блока с номером q, $q \in \{1,\ldots,4\}$:

3.1. Если обработанный блок с номером q принадлежит первому циклу и обработан стартовой программной секцией, то его состояние S_{0q} меняется с 1 на 3, случайным образом выбирается код c_s финишной секции, случайным образом выбирается один из u вариантов ее реализации, который определяет адрес A_s начала программного модуля. В список R готовых к исполнению блоков включается запись с компонентами N_c=0, n_b=q,A_s,A_d. Возврат на пп. 2.

3.2. Если обработанный блок с номером q принадлежит первому циклу и обработан финишной программной секцией, то его состояние S_{0q} меняется с 3 на 4. Если среди компонент S_0 есть не менее 2-х компонент равных 4-м, то все блоки второго цикла, состояние которых соответствует коду 0, помещаются в список R готовых к выполнению блоков. При этом единичные компоненты кода c_s стартовой секции определяются компонентами S_0, которые равны 4-м; случайным образом выбирается один из u вариантов реализации секции. Состояние всех рассматриваемых блоков меняется с 0 на 1. Если все компоненты S_0 равны 4-м, то все блоки второго цикла списка Z помещаются в список R, а их состояние меняется с 2 на 3. При включении их в список R код финишной секции определяется путем инвертирования кода стартовой секции. После этого осуществляется корректировка списка R инвертированием полей N_c, и переопределение кодов состояний: S_0=S_1, S_1={0,0,0,0}. Инкрементируется счетчик C выполненных циклов: если он не равен предельному значению, то возврат на пп.1., иначе – конец.

3.3. Если обработанный блок с номером q принадлежит второму циклу и обработан стартовой программной секцией (S_{1q}=1), то анализируется код c_s секции с использованием которой он обработан: если нулевые компоненты кода c_s соответствуют значениям 4 одноименных компонент S_0, то q-тый блок готов в продолжению обработки: его состояние S_{1q} меняется с 1 на 3, случайным образом выбирается код c_s финишной секции, случайным образом выбирается один из u вариантов ее реализации, который определяет адрес A_s начала программного модуля. В список R готовых к исполнению блоков включается запись с компонентами N_c=1, n_b=q,A_s,A_d. Если компоненты S_0 соответствующие единичным компонентам c_s не равны 4, то есть данные, необходимые для продолжения обработки q-того блока не готовы, то его состояние S_{1q} меняется с 1 на 2, блок включается в список Z.

Оценка эффективности

Анализ эффективности выполнялся по теоретической модели предложенного способа полиморфной реализации алгоритма *Rijndael* и по результатам экспериментальных исследований.

Исходя из доказанного выше факта о том, момент выполнения каждой операции алгоритма *Rijndael* может варьироваться внутри временного интервала выполнения двух смежных циклов, в качестве теоретической модели исследовался граф возможных вариантов исполнения двух смежных циклов алгоритма. В рамках исследований рассматривалась возможность реализации каждой из секций в k вариантах. Общее число команд, непосредственно используемых для реализации одного цикла равно 130.

Исследования показали, что при k=1 количество вариантов чередования секций при выполнения пары циклов составило $2.6 \cdot 10^6$. Число операций случайного выбора случайной секции при выполне-

нии одного цикла равно 10. При этом среднее число альтернатив выбора секций в ходе выполнения цикла составляет 31.

Приведенные результаты свидетельствуют о том, что уже при k=1 число вариантов выполнения каждого из циклов практически исключает возможность использования подбора кода программы при ее сопоставлении с данными АДПМ.

Экспериментальные исследования проводились с использованием программы, написанной на Ассеблере для эмулятора однокристального микроконтроллера. Анализ результатов экспериментов показал, что максимальное значение вариации локализации отдельной команды в коде программы составляет 142, что составляет 85% времени выполнения цикла. Эксперименты показали, что применение предложенного способа полиморфной реализации алгоритма *Rijndael* увеличивает время его выполнения на 76%.

Выводы

В результате проведенных исследований предложен способ стохастически полиморфной реализации алгоритма *Rijndael* на микроконтроллерах и смарт-картах. Способ позволяет реконфигурировать последовательность команд программной реализации алгоритма при каждом ее запуске, что противодействует реконструкции постоянно используемых в алгоритме данных технологиями АДПМ. Проведенный анализ вычислительных процедур алгоритма Rijndael, позволил выявить теоретически возможные границы полиморфизма программной реализации и уменьшить время, дополнительно затрачиваемое на реконфигурацию. В разработанной организации стохастически полиморфной реализации вариация локализации команд в рамках одного цикла составляет 85%. Экспериментально показано, что увеличение времени реализации алгоритма за счет введения стохастичекого полиморфизма не превышает 76%.

Применение предложенного способа позволяет существенно повысить надежность защиты информации в компьютерных сетях промышленного и коммерческого назначения.

Список литературы

1. *Kocher P., Jaffe J., Jun B.* Differential Power Analysis // Proceeding of CRYPTO'99. – Springer-Verlag. – 1999. – P. 388–404.

2. *Akkar M-L., Giraud C.* An implementation of DES and AES, Secure against Some Attacks.// Proceeding of 2-th International Workshop "Cryptographic Hardware and Embedded Systems"(CHES-2001), LNCS-1965. – Springer-Verlag. – 2001. – P. 309–318.

3. *Марковский А.П., Абабне О.А., Ияд Мохд Маджид Ахмад Шахрури*. Способ защиты ключей алгоритма ГОСТ 28.147-89 от реконструкции анализом динамики потребляемой мощности // Вісник національного технічного університету України "КПІ". Інформатика, управління та обчислювальна техніка. – 2007. – № 46. – С. 128–138.

4. *Марковский А.П., Стефанская В.А., Грищенко С.В.* Апартная реализация алгоритма Rijndael в FPGA-структурах // Вісник Національного технічного університету України "КПІ". Інформатика, управління та обчислювальна техніка. – К.: ВЕК+. – 2002. – № 38. – С. 24–33.

Подано до редакції 22.03.10