УДК 004.056 /002:006.354
ББК 78.34

МЕТОД ПОСТРОЕНИЯ АЛГОРИТМА ХЕШИРОВАНИЯ СТОЙКОГО К УЯЗВИМОСТИ ПОИСКА КОЛЛИЗИЙ НА SIMD АРХИТЕКТУРЕ ВИДЕОКАРТ

Фархадов М.П. [1], Надеинский Л.В. [2], Ситар А.А. [3]
(*Федеральное государственное бюджетное учреждение науки Институт проблем управления им.В.А. Трапезникова Российской академии наук, Москва*)

В статье рассмотрена методика борьбы с уязвимостью хранения паролей в хешированном виде. Предлагаемый алгоритм хеширования основывается на специфических особенностях архитектуры современных графических процессоров.
В целях противодействия распараллеливанию, необходимо и достаточно организовывать раунды хеширования таким образом, чтобы на каждые несколько тактов работы минипроцессора (не более 800 тактов) осуществлялся доступ к памяти в произвольном порядке. Показано, что алгоритм хеширования, обладающий этими свойствами, не поддается распараллеливанию на современной SIMD архитектуре.
Предложена программная реализация такой криптографической хеш-функции, скорость работы которой сравнимой с реализациями алгоритмами SHA-2.

Ключевые слова: Хеш-функция, брутфорс, алгоритм хеширования, GPU, SSE.

---

[1] *Маис Паша-Оглы Фархадов, заведующий лабораторией, доктор технических наук (mais@ipu.ru), (Москва, ул. Профсоюзная, д. 65, тел. (495) 334-87-10).*.
[2] *Лев Андреевич Надеинский, старший инженер-программист*, (*superpub@yandex.ru*).
[3] *Александр Александрович Ситар, инженер,* (*sitar.alex@gmail.com).*

1. Введение

В научно-технической литературе рассматривается активно используемая уязвимость хранения паролей в хешированном виде, основанная на одном из методов распараллеливания последовательных алгоритмов [6-9].

В ходе работы по исследованию особенностей современной SIMD архитектуры выявлены такие свойства, как латентность памяти, возможность чтения из памяти только последовательно блоками по 128 байт, осуществление только последовательного обращения к массивам процессорными нитями. Эти особенности применены для разработки и реализации класса стойких к распараллеливанию на SIMD архитектуре криптографических хеш-функций. Перенос алгоритма на SIMD архитектуру GPU может быть осуществлен различными способами. Доказано, что для противодействия распараллеливанию на SIMD архитектуре алгоритмов хеш-функций необходимо, чтобы мощность множество упорядоченных наборов раундов хеширования была не меньше, чем мощность множества всевозможных хеш-кодов, а количество различных реализаций вариантов раундов превышало количество минипроцессоров на GPU [4, 5].

В 2008-ом году была выявлена уязвимость хранения паролей в виде хэш-кодов [1]. С развитием архитектуры графических процессоров решение задачи нахождения второго прообраза для хеш-кодов MD5 [2], SHA [3] стало возможным в кротчайшие сроки (при условии наличия современной графической видеокарты) даже на домашнем компьютере.

Многие системы используют базу данных для хранения паролей и существует несколько способов для хранения паролей в виде хеш-кодов [1]:

1. Хранятся только хэш-коды (MD5 [2], SHA [3]) паролей. Не существует полиномиального алгоритма для нахождения второго прообраза (коллизии [1]) к нему. Но при условии использования несложного, популярного или просто «несчастливого» пароля (который встречался ранее и занесён в некоторые заранее заготовленные «радужные таблицы» [7]) такая задача решается за доли секунды.

2. Хранятся хеш-коды паролей и несколько случайных символов. К каждому паролю добавляется некоторая строка инициализации - несколько случайных символов (их ещё называют «salt» или «соль») и результат ещё раз хешируется. Например, md5(md5(pass)+word). Найти пароль с помощью «радужных таблиц» таким методом не получится.

Все эти методы хранения ключевой информации позволяют минимизировать последствия успешной атаки на систему: если злоумышленник получил доступ к базе данных аккаунтов (таблицы, в которых хранятся имена пользователей и их пароли в виде хэш-кодов), то ему не удастся использовать эти данные для получения доступа к аккаунту, так как задача нахождения прообраза или коллизии хэшированного пароля для него будет неразрешимой за полиномиальное время.

Для использования параллелизма архитектуры видеокарты существует несколько подходов преобразования последовательных программ в параллельные:

1. Поток входных данных разделяется на независимые блоки, которые обрабатываются параллельно.

2. Внутренние статические данные алгоритма разделяются между процессорными нитями видеокарты.

При переносе вычислений, связанных с перебором, на видеокарту алгоритм хеширования не подвергается распараллеливанию, а высокая скорость достигается за счет выполнения одинаковых операций над достаточно большим объемом независящих друг от друга строк схожей длины (множество предполагаемых паролей). Среднее время генерации хеш-кода при использовании видеокарты в 1000 и более раз меньше, чем при использовании центрального процессора. Другими словами, если центральный процессор находит коллизию за 100лет, то графический способен сделать это за 30 дней и менее в домашних условиях, что ставит под угрозу большинство систем хранения паролей, электронные подписи и все алгоритмы, опирающиеся на криптографические хеш-функции.

Цель статьи — сформировать подход к созданию криптостойкой функции хеширования не поддающейся атаке перебором на графической карте.

Задачи:

1) изучить особенности архитектуры GPU;
2) изучить особенности архитектуры x86 и её расширений;
3) сформировать список требований к разрабатываемому алгоритму;

описать работу алгоритма.

2. Архитектурные особенности GPU

Несмотря на то, что пиковая производительность GPU в домашнем компьютере может достигать 32TFlops [10], GPU не является заменой центральному процессору в силу своих особенностей. Для усложнения обработки хеш-функции на графическом процессоре были выявлены следующие особенности его архитектуры:

1. При каждом обращении к памяти всегда считываются 128 байт последовательно, даже если нужен только 1 байт

Кроме того, один поток за один раз может получить доступ только к 16 байтам из этих 128 считанных.

В результате получается, что пропускная способность памяти больше 150Гб/с, но только при условии, что все 128 байт используются постоянно. Если же каждый поток должен считать одну большую структуру, которая занимает 40 байт, то каждый поток должен сделать 3 запроса к памяти и считать 3*128 байт. А если данные для каждого потока располагаются в разных местах (а поток получает указатель на них и затем загружает - вполне нормальная ситуация для CPU, в случае, когда память расходуется рационально), то полезная пропускная способность памяти получается 40*32/(128*3*32), то есть всего 10% от теоретической.

2. Латентность (задержка обращения к ресурсу) памяти составляет примерно 800 циклов для каждого запроса

В предыдущем примере для получения данных всеми процессами надо сделать 3*32 запросов (почти 80 тысяч циклов). В это время можно выполнять другие потоки.

3. На все активные потоки минипроцессоров выделяется, в общей сложности, 32 тысячи регистров

32 тысячи регистров выделяются на все активные потоки, а не только на выполняющиеся (причем они выделяются статически по максимуму: сколько надо регистров в самом худшем ветвлении, столько и будет выделено). А активных потоков должно быть 1536, чтобы скрывать латентность памяти, то есть остается 21 регистр на поток. С другой стороны, можно попробовать использовать меньше, чем полторы тысячи активных потоков.

4. Если поток использует больше 64 регистров, то они хранятся в глобальной памяти

То есть появляются дополнительные запросы к памяти. Для борьбы с использованием регистров и оптимизацией загрузки есть еще разделяемая память (shared). Но ее только 16/48Кб и она разделяется между всеми активными группами, то есть если каждая группа потребляет 25кб памяти, то больше одной группы запустить не получится.

5. Нити должны обращаться либо к 32-битовым словам, давая при этом в результате один 64-байтовый блок (транзакцию), либо к 64-битовым словам, давая при этом один 128-байтовый блок (транзакцию)

6. Нити должны обращаться к элементам памяти последовательно, каждой следующей нити должно соответствовать следующее слово в памяти (некоторые нити могут вообще не обращаться к соответствующим словам)

7. Запуск каждого ядра сопровождается небольшой задержкой

8. Последовательное выполнение операций дает многократное замедление выполняемого алгоритма относительно центрального процессора

9. Частота графического процессора ограничена 1.5ГГц

Что в 2 раза меньше чем частота CPU

Эти особенности не могут быть исправлены в данной архитектуре, так как физический размер кристалла графического процессора не сильно отличается от кристалла CPU, точность технического процесса одинакова и потребление энергии отличается не значительно. Даже для того что бы снабдить каждую нить

GPU достаточным объемом памяти потребуется огромный объем памяти (в 1000 больше). Такие улучшения не помогут обрабатывать видео информацию (основная цель работы видеокарты), но существенно увеличат стоимость, энергопотребление и занимаемый объем.

3. Архитектурные особенности SIMD центрального процессора

Следует отметить, что современные CPU имеют SIMD модуль, который сильно отличается от видеокарт. Нельзя просто усложнить алгоритм хеширования до такой степени, что бы он медленно выполнялся и на видеокарте, так как еще более медленное выполнение алгоритма на CPU повлечет серьезные проблемы связанные с возможностью вывода серверов из строя (DdoS), заставляя их многократно выполнять тяжелый алгоритм. Это означает, что разрабатываемый алгоритм хеширования не должен использовать видеокарту и не должен иметь возможность её использовать эффективно, так как на большинство серверов видеокарта не устанавливается вообще. Но все сервера оснащаются процессорами с потоковыми расширениями набора команд.

Современная архитектура x86 тоже умеет выполнять операции с векторными типами. Потоковое расширение x86 архитектуры называется SSE (SSE, SSE2, SSE3, SSSE3, SSE4, SSE4A, SSE4.1, SSE4.2 и некоторые другие) и реализует на уровне микросхем несколько сотен SIMD инструкций, например:

1) работа с 256-битными регистрами;
2) заполнение всего регистра одним и тем же загруженным значением;
3) условная загрузка/сохранение float/double чисел в регистр в зависимости от знака чисел в другом регистре;
4) обнуление старших 128 бит всех регистров;
5) вставка/получение любой 128-битной части 256-битного регистра;
6) перестановка 128-битных частей 256-битного регистра. Параметр перестановки задаётся статически;

7) перестановка float/double чисел внутри 128-битных частей 256-битного регистра. При этом параметры перестановки берутся из другого регистра;

8) Высокоскоростной обмен информацией с CPU с низким значением задержки;

9) Два регистра операнда рассматривается как одно беззнаковое промежуточное значение удвоенной размерности из которого извлекается 64-х/128-х битное значение начиная с байта указанного в непосредственном аргументе-константе команды;

10) Перестановка байт: каждый байт результата есть некоторый байт из первого аргумента определяемый по соответствующему байту из второго аргумента (если байт отрицательный, то в байт результат прописывается ноль, иначе используются младшие 3 или 4 бита как номер байта в первом аргументе);

11) Большое количество разнообразных схем умножения;

12) Горизонтальные сложения/вычитания целых большой длинны;

13) Подсчет CRC32 для 64бит.

Эти аппаратные возможности позволяют выполнять векторные операции, которые нельзя быстро выполнить на GPU, с высочайшей скоростью в несколько потоков.

3.1. ТРЕБОВАНИЯ К АЛГОРИТМУ

Учитывая особенности архитектуры GPU и прогнозы развития технологий был сформирован список требований к алгоритму хеширования, выполнив которые его нельзя будет ускорить на графическом процессоре:

1. Внутри алгоритма необходимо организовать многократный вызов различных функций;
2. Доступ к памяти должен осуществляться в случайном порядке;
3. Необходимо организовать ветвления алгоритма в зависимости от промежуточных результатов хеширования;

4. Для противодействия взлому с помощью «радужных таблиц» алгоритм должен обладать возможностью модификации для каждой системы

5. Использование нескольких зависящих друг от друга алгоритмов для организации ветвления;

6. Алгоритм должен обладать возможностью масштабирования;

7. Использование векторных преобразований на SSE.

3.2. ПОШАГОВОЕ ОПИСАНИЕ АЛГОРИТМА

Введем некоторые обозначения:

Пусть X — множество сообщений. X является подмножеством конечных последовательностей символов конечного алфавита A.

Пусть Y — множество двоичных векторов фиксированной длины n. Тогда хеш-функция — есть функция $f: X \rightarrow Y$

Стоит отметить, что криптографическая хеш-функция должна отвечать следующим требованиям[1]:

- Необратимость
- Стойкость к коллизиям первого рода
- Стойкость к коллизиям второго рода

Время выполнения алгоритма и объем используемой памяти не должно резко отличаться от существующих криптостойких алгоритмов.

Ускорением на GPU будем называть отношение времени работы алгоритма на CPU к времени работы алгоритма на GPU

Утверждение: Для получения значения ускорения на GPU не превышающего отношения тактовой частоты GPU к тактовой частоте CPU необходимо и достаточно, чтобы мощность множества упорядоченных наборов раундов хеширования была не меньше, чем мощность множества всевозможных хеш-кодов, а количество различных реализаций вариантов раундов превышало количество минипроцессоров на GPU.

Доказательство:

Из ограничения 8 и 9 следует, что последовательное выполнение программного кода на GPU занимает примерно в 2 раза

больше времени, чем на центральном процессоре. Предположим, что сам алгоритм раунда хеширования распараллеливанию на SIMD архитектуре видеокарты не поддается, так как либо не использует операнды и операции, к которым можно применить распараллеливание, либо распараллеливание операций не позволяет скрыть латентность памяти.

Очевидно, что если для каждой входной последовательности будет применен различный алгоритм, то получить значение ускорения больше 0.5 (отношение частоты процессора видеокарты к частоте центрального процессора) будет невозможно используя распараллеливание потока входных данных. Можем уменьшить количество различных алгоритмов раундов, показав отсутствие возможности объединения различных входных последовательностей в группы с общим алгоритмом обработки. Для этого рассмотрим множество выходных последовательностей, так как в рамках алгоритма хеширования не целесообразно рассматривать большее количество различных вариантов получения хеш-кода, для объединения входных последовательностей в вышеописанные группы. В силу того, что мощность множества различных хеш-кодов значительно превышает количество минипроцессоров на GPU объединения по хеш-кодам быть не может. В случае если на видеокарте есть только 1 минипроцессор достаточно организовать различные алгоритмы путем последовательного применения 2x различных раундов, зависимых друг от дуга, в произвольном порядке достаточное количество раз. В случае если GPU состоит из большего количества минипроцессоров, то распараллеливание можно произвести, выполняя различные алгоритмы раундов на различных минипроцессорах, но такой вариант можно исключить в силу ограничений 2 и 7 если использовать различные алгоритмы раундов в количестве превышающем количество минипроцессоров.

При длине хеш-кода в N бит получаем до $2N$ различных хэш-кодов. Это означает, что необходимо использовать $s \approx \log m\ (2N)$ различных раундов, где m — длина последовательности применения раундов. При этом ускорение выполнения такого алгоритма на GPU заведомо уменьшится минимум в s раз, а при хране-

нии промежуточных результатов может оказаться, что распараллеливание на GPU вообще не целесообразно.

Объем обрабатываемой информации в одном раунде должен превышать объем регистров графической нити (32 768 байт). Это не позволит скрыть латентность глобальной памяти.

Обычно (MD5 , MD4, SHA-1) в алгоритмах проходит несколько циклов (3-5), состоящих из 16-20 шагов. Сообщение x∈X разбивается на *N* блоков по m бит(M_1,...,M_N), в конце каждого блока записывается индекс длины сообщения. Блоки преобразуются в вектора H длины k. Вектора разбиваю на 4-5 составляющих (abcde).

$$H_0 = 0$$
$$H_i = E_{M_i}(M_i + H_j), i = 1, \ldots, N$$
$$h(M) = H_N$$

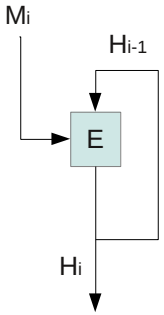

Рис. 1. Пример одного шага хеширования

Циклы MD5, SHA1 выглядят следующим образом:

- (b∧c)∨(¬b∧d)
- (b∧c)∨(b∧d)∨(c∧d)
- b+c+d
- b+(c∨¬d)
- (b∧d)∨(c∧¬d)

где b,c,d — инициализированные вектора [2].

В предлагаемом алгоритме хеширования будем использовать разбиение вектора H на *n* блоков (по умолчанию *n*=8 или 16), такое количество блоков нужно для создания необходимого количества различных раундов хеширования. Такое количество блоков не будет превышать количества SSE регистров, что позволит организовать сверхбыстрые, сложные операции.

Длина каждого блока равна 64 – 256 б. Блоки будем именовать буквами латинского алфавита (a,b,c,d,e,f,g,h). Длина блока обусловлена возможностями SSE.

В таблицу T будем записывать промежуточные результаты выполнения различных раундов.

E_t - *t*-й раунд хеширования. На вход поступает шесть векторов. Выбор алгоритма осуществятся из списка доступных раундов:

1. $B \oplus C \oplus D \oplus E \oplus F \oplus G$.

2. Горизонтальные сложения/вычитания целых:

(Input — {A0, A1, A2, A3}, {B0, B1, B2, B3}.

Output — {A0-A1 A2-A3 … B0-B1 B2-B3 …}).

3. Аргументы A и B рассматриваются как вектора 16-ти битных знаковых чисел с фиксированной запятой представленных в диапазоне [-1,+1) (то есть 0x4000 это 0.5, а 0xa000 это −0.75 и т. д.), которые перемножаются друг с другом с корректным округлением:

(Input — {A0, A1…}, {B0, B1…}.

Output — {A0 * B0, A1 * B1…}).

4. Производится побайтное перемножение векторов A и B, промежуточные 16-ти битные результаты попарно складываются между собой с насыщением и выдаются как результат:

(Input — {A0, A1, A2, A3,..}, { B0, B1, B2, B3, ... }.

Output — {(A0*B0+A1*B1), (A2*B2+A3*B3), …}).

5. Перестановка байт, каждый байт результата есть некоторый байт из первого аргумента определяемый по соответствующему байту из второго аргумента (если байт отрицательный, то в байт результат прописывается ноль, иначе используются младшие 3 или 4 бита как номер байта в первом аргументе):

(Input — {A0, A1, A2,.. A7/A15}, {B0, B1, B2,.. B7/B15}.

Output — {[AB0 AB1 AB2 …]}).

6. $\left((B \wedge C) \oplus \left(\overline{B} \wedge D\right) \vee E\right) \oplus \left(\overline{E} \wedge F\right) \oplus \left(\overline{F} \wedge G\right)$

6.1. $(\alpha_1 \wedge \alpha_2) \oplus \left(\overline{\alpha_1} \wedge \alpha_3\right)$, где α — B, C, D, E, F либо G

6.2. $(\alpha_1 \wedge \alpha_2) \oplus (\alpha_2 \wedge \alpha_3) \oplus (\alpha_3 \wedge \alpha_4) \oplus (\alpha_2 \wedge \alpha_4) \oplus (\alpha_1 \wedge \alpha_4) \oplus (\alpha_1 \wedge \alpha_3)$, где α — B,C,D,E,F либо G

6.3. $(\alpha_1 \vee \alpha_2) \oplus (\alpha_2 \vee \alpha_3) \oplus (\alpha_3 \vee \alpha_4) \oplus (\alpha_2 \vee \alpha_4) \oplus (\alpha_1 \vee \alpha_4) \oplus (\alpha_1 \vee \alpha_3)$

7. ROTR ROTL — циклические сдвиги вправо или влево. Два регистра операнда рассматривается как одно без знаковое промежуточное значение удвоенной размерности, из которого извлекается 64-х/128-х битное значение, начиная с байта указанного в непосредственном аргументе-константе команды.

(Input — {A0, A1}, {B0, B1}, imm8.

Output — {B1_B0_A1_A0 >> (imm8 * 8)}).

Возможно добавление дополнительных алгоритмов раундов.

Первым шагом алгоритма хеширования является добавление недостающих бит. Сообщения X увеличивается таким образом, чтобы его длина была кратна (*n**64 - 128) по модулю *n**64. Добавление осуществляется всегда, даже если сообщение уже имеет нужную длину. Таким образом, число добавляемых битов находится в диапазоне от 1 до *n**64.

На втором шаге к полученному сообщению прибавляется величина равная длине исходного сообщения по модулю 128. В результате получено сообщение X'.

Третий шаг характеризуется инициализацией начальных значений векторов a, b, c, ... , h, ...

Начальные значения векторов вычисляются как абсолютные значения функции 264*sin(*i*), где *i* является номером вектора в радианах.

Четвертый шаг — выбор начального раунда. На этом шаге на основе входных данных определяется функция E1 .

Пятый шаг — обработка последовательности.

Основой алгоритма является цикл, который повторяется не менее 100 раз. С увеличением количества циклов растет размер

таблицы T, следует выбирать количество циклов исходя из необходимой стойкости к распараллеливанию на современной графической архитектуре. Обработка последовательности выглядит следующим образом.

X' разбивается на блоки длины n*64. (X'0,X'1,...X'l-1) всего l блоков.

Каждый цикл (раунд) алгоритма выглядит следующим образом (рис. 2):

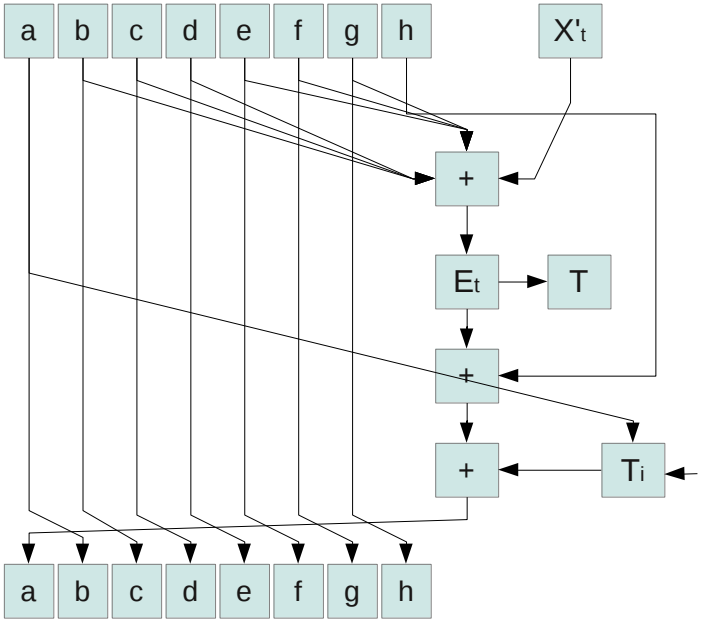

Рис. 2. Схема раунда хеширования

Алгоритм включат в себя *p* различных преобразований E_j , где $j \in \overline{1,p}$. На рисунке показана схема алгоритма выбора

следующего преобразования.

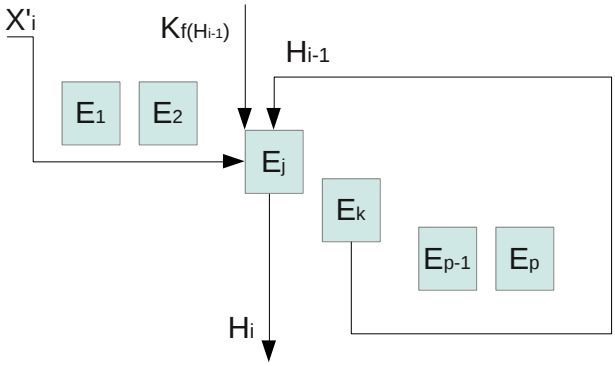

Рис. 3. Преобразование с использованием SSE оптимизации

На рисунке 3 на вход алгоритма *Ej* подаются:

- блок X'*i*
- Промежуточный результат H*i*-1
- Ключ, полученный при выполнении E*f*(H*i*-1), где *f* — полиномиальная функция над полем GF.

Заключительный шаг. В результате преобразований вектор abcdefgh — является хешкодом или дайджестом.

4. Заключение

Предлагаемый алгоритм обладает особенностями, которые значительно увеличивают стойкость к перебору паролей, не влияя на производительность на обычном процессоре.

Основным преимуществом является теоретическое обоснование стойкости и её сохранение с развитием микропроцессорной техники.

Предполагается, что представленный алгоритм будет эффективен в криптографических приложениях, которые выполняются на серверной части.

Развитие подхода может позволить создать целое семейство алгоритмов, обладающих стойкостью к атакам перебора паролей на видеокартах.

Литература

1. ЧЕРЕМУШКИН А.В *«Криптографические протоколы. Основные свойства и уязвимости»,* Академия, 2009 год.
2. «*Хэш-функции и аутентификация сообщений. Часть 1*» http://www.intuit.ru/department/security/networksec/8/3.html (дата обращения: 18.01.2013).
3. «*Хэш-функции и аутентификация сообщений. Часть 2*» http://www.intuit.ru/department/security/networksec/9/2.html (дата обращения: 16.11.2012).
4. *Документация по работе с SIMD архитектурой AMD* http://developer.amd.com/sdks/AMDAPPSDK/assets/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf (дата обращения: 18.01.2010).
5. *Документация к GPU Nvidia* http://developer.nvidia.com/nvidia-gpu-computing-documentation (дата обращения: 18.01.2010).
6. *Program to recover/crack SHA1, MD5 & MD4 hashes* http://www.golubev.com/files/ighashgpu/readme.htm
7. *Проект «Взлом с использованием радужных таблиц»* http://project-rainbowcrack.com/ (дата обращения: 21.04.2013).
8. Проект «Взлом MD5 на GPU» http://bvernoux.free.fr/md5/index.php (дата обращения: 18.01.2007).
9. *Проект «World Fastest MD5 cracker BarsWF»* http://3.14.by/ru/md5 (дата обращения: 18.01.2007).
10. URL: http://www.amd.com/ru/products/desktop/graphics/7000/7970/Pages/radeon-7970.aspx#3 (дата обращения: 17.04.2013).
11. Наборы процессорных комманд URL: http://ru.wikipedia.org/wiki/CLMUL_instruction_set#Intel (дата обращения: 28.07.2011).

THE METHOD OF CONSTRUCTING A HASH ALGORITHM POSTS TO VULNERABILITY IN SEARCH OF COLLISION SIMD ARCHITECTURE VIDEO CARD

Mais Farkhadov, Institute of Control Sciences of RAS, Moscow, Doctor of Science, assistant professor (mais@ipu.ru).
Lev Nadeinsky, Institute of Control Sciences of RAS, Moscow, Senior Software Engineer, professor (Moscow, Profsoyuznaya st., 65, (superpub@yandex.ru).
Alexander Sitar, Institute of Control Sciences of RAS, Moscow, Moscow Institute of Physics and Technology, Moscow, Senior Engineer (sitar.alex@gmail.com).

Abstract: *The article describes a technique to combat vulnerability storage of passwords in hashed form. The proposed hashing algorithm is based on the specifics of the architecture of modern GPUs.*
In order to counter the parallelization is necessary and sufficient to organize hash rounds in such a way that for every few bars of miniprotses the dirt (no more than 800 cycles) to access the memory in any order. It is shown that the hash algorithm has these properties cannot be parallelized on modern SIMD architecture.
We propose a software implementation of a cryptographic hash function, the speed of which is comparable with implementations of SHA-2 algorithms.

Keywords: Hash function, brute force, the algorithm is hashed-tion, GPU, SSE.

Статья представлена к публикации
членом редакционной коллегии ...заполняется редактором…

Поступила в редакцию ...заполняется редактором…
Опубликована ...заполняется редактором…