А.А.Волчков, Н.Ю.Де-Мондерик, А.Н.Лебедев

АЛГОРИТМ WICKER’98

Введение.

Представленный в настоящей работе алгоритм Викер-98 разрабатывался как многоцелевой алгоритм обработки данных, предназначенный как для защиты данных, хранимых и/или обрабатываемых внутри компьютера, так и для защиты информации от несанкционированного доступа при ее передачи по открытым каналам связи. Блочная структура представления данных в большинстве числе логических устройств персональных и средних компьютеров делает предпочтительным использование блочного алгоритма защиты информации. Кроме того блочный принцип обработки данных позволяет стандартным способом получить из такого алгоритма алгоритм хэширования, необходимый для защиты данных от несанкционированного изменения и для создания систем цифровой аутентификации электронных документов. Более того, стандартной процедурой блочный алгоритм защиты данных модифицируется в алгоритм гаммирования, применяемый при потоковой обработки информации.

Размер блока и базовые функции алгоритма выбирались с ориентацией на его последующую реализацию на процессорах семейства Пентиум, как наиболее распространенные в настоящее время на российском рынке. Поэтому изначально схема функционирования алгоритма выбиралась так, чтобы обеспечить наиболее полную загрузку обоих конвейеров процессора. При этом преследовалась цель, чтобы оптимизированная ассемблерная программа незначительно превосходила по скорости программу, написанную на языке высокого уровня и оптимизированную стандартными средствами транслятора. В качестве языка высокого уровня, с которым сравнивается ассемблерная реализация, был взят язык СИ, как наиболее популярный среди разработчиков программного обеспечения.

Вместе с тем, несмотря на ориентацию на конкретное семейство процессоров, при разработке алгоритма преследовалась цель сделать алгоритм наиболее удобным для

реализации на маломощных процессорах, например, в интеллектуальных картах, и обеспечить наилучшую производительность при перенесении алгоритма на специализированный микрочип. По этой причине, в частности, в алгоритме отсутствуют операции целочисленного умножения, несмотря на то, что в процессорах семейства Пентиум, они достаточно оптимизированы и выполняются за одну команду.

Характеризуя алгоритм Викер-98 в целом нельзя не отметить, что несмотря на то, что он разрабатывался независимо и из других соображений, чем ранее опубликованные алгоритмы RC-5 и RC-6, последние оказались к нему наиболее идейно близкими: видимо сказалось близость стандартного набора команд микропроцессоров, из которых конструировались алгоритмы. Принципиальным отличием алгоритма Викер-98 от указанных алгоритмов является отсутствие контекстно-зависимого сдвига операндов: все операции циклического сдвига в Викер-98 указаны явно. Это как минимум приводит к большей производительности алгоритма, особенно на малоразрядных процессорах и специализированных чипах, где такому сдвигу просто отвечает фиксированная перемычка между соседними частями конвеера. С другой стороны отсутствие контекстно-зависимого сдвига ведет к меньшей зависимости аргументов на этой операции и приходится специально подбирать и анализировать фиксированные сдвиги параметров, чтобы обеспечить полную зависимость каждого выходного бита от всех входных битов и битов ключа. При этом соответственным образом усложняется линейный и дифференциальный анализ алгоритма.

Как положительный итог этого следует отметить, что фиксация сдвига не позволяет делать предположения о его возможном совпадении с требуемой величиной и стоить на этой основе простые дифференциалы, как это имеет место для RC-6, где именно дифференциальный анализ оказался наиболее мощным инструментом по вскрытию защиты. Другими словами, алгоритм Викер-98 требует более скрупулезного подбора величин циклического сдвига, но зато обеспечивает этим большее противодействие линейному и дифференциальному анализу, чем RC-5 и RC-6.

Алгоритм Викер-98 имеет примерно тоже число циклов (9 - по сравнению с 10), что и RC-6, каждый из которых состоит из 4-х итераций (в RC-6 цикл имеет 2 итерации). Под циклом понимается условно повторяющийся отрезок

вычислений с учетом задействованных аргументов, а под итерацией - без их учета. Как правило, блочный алгоритм строится как последовательность итераций, где при переходе от итерации к итерации формально переставляются используемые блоки памяти (регистры). Тем самым, итерация - элементарное преобразование выполняемое алгоритмом, а цикл - периодичность обработки каждого элемента памяти.

В каждую итерацию Викер-98 изменяется по 2 регистра, как и в RC-6, но фактически удвоенное по сравнению с RC-6 число итераций приводит к удвоенной частоте преобразований регистров. Каждая итерация в Викер-98 состоит фактически из двух независимых частей, что позволяет их выполнять параллельно на двух конвейерах Пентиума. В отличие от Викер-98 в RC-6 каждая итерация содержит два умножение и полное ее распараллеливание не достигается и это приводит к тому , что цикл RC-6 по длительности превосходит цикл Викер-98.

В итоге, алгоритм Викер-98 имея примерно тоже число циклов, что и RC-6, работает быстрее, выполняя при этом почти удвоенное число итераций и как следствие - удвоенное число преобразований регистров, хотя каждая из этих операций несколько проще.

Описание алгоритма Викер-98

В алгоритме используются четыре 32-битных регистра, обозначаемые далее через **a**, **b**, **c** и **d**. Перед началом работы в эти регистры записывается открытый (или закодированный) текст, а по окончанию вычислений из них считывается закодированный (соответственно, декодированный т.е. открытый) текст.

Кодирование и декодирование производится на основе рабочего ключа, состоящего из 44 слов, каждое из которых является 32-битным. Эти слова будут далее обозначаться как KS[0],KS[1], ... , KS[43].

Операции кодирования и декодирования отличаются друг от друга с вычислительной точки зрения, но используют одинаковый рабочий ключ. Рабочий ключ получается из базового (начального) путем его повторения необходимого

числа раз. Другими словами, рабочий ключ является периодической последовательностью 32-битных слов длиной в 44 слова, период которой составляет базовый (начальный) ключ.

Предполагается три основных варианта алгоритма Викер-98 различающиеся только длиной базового ключа. Это варианты по 4, 6 и 8 слов в базовом ключе.

Кодирование и декодирование начинается и заканчивается “наложением” рабочего ключа на регистры a, b, c, d. Собственно вычисления кодирования и декодирования осуществляется за 9 циклов по 4 итерации каждый. Итерации имеют одинаковый вид и различаются конкретным выбором задействованных операций и порядком операций над регистрами. Фактически итерация - элементарное преобразование выполняемое алгоритмом.

Объединение четырех итераций в один цикл преследует цель логически выделить повторяющуюся часть вычислений: ровно за 4 итерации каждый из регистров a, b, c и d успевает побывать на месте каждого из формальных аргументов.

Схема одной итерации кодирования представляется в виде:

$$x = ((x + KS[]) + y) >> t, \quad z = z + (x * v); \qquad (1)$$

Здесь знак + используется для обозначения обратимой операцией над 32-битными словами, а знак * - для необратимой операции. Символы >> обозначают циклический сдвиг слова на t разрядов в сторону младших.

KS[] - очередное слово рабочего ключа.

Буквы x, y, z, v - формальные переменные, отождествляемые в каждой итерации с регистрами a, b, c, d (разным переменным отвечают разные регистры). При переходе от итерации к итерации фактические параметры циклически сдвигаются в алфавитном порядке: a заменяется на b, b - на c, c - на d, d - на a .

Схема кодирования алгоритма Викер-98 имеет вид:

Таблица 1.
Схема алгоритма кодирования.

начальное ключа	наложение	$a = a +_1 KS[0]$, $b = b +_2 KS[1]$, $c = c +_3 KS[2]$, $d = d +_4 KS[3]$

	итерация 1	$c=((c+_{1,1}KS[4])+_{1,2}b)>>t_1, a=a+_{1,3}(c*_1 d)$,
	итерация 2	$d=((d+_{2,1}KS[5])+_{2,2}c)>>t_2, b=b+_{2,3}(d*_2 a)$,
цикл 1	итерация 3	$a=((a+_{3,1}KS[6])+_{3,2}d)>>t_3, c=c+_{3,3}(a*_3 b)$,
	итерация 4	$b=((b+_{4,1}KS[7])+_{4,2}a)>>t_4, d=d+_{4,3}(b*_4 c)$,
	итерация 5	$c=((c+_{5,1}KS[8])+_{5,2}b)>>t_5, a=a+_{5,3}(c*_5 d)$,
	итерация 6	$d=((d+_{6,1}KS[9])+_{6,2}c)>>t_6, b=b+_{6,3}(d*_6 a)$,
цикл 2	итерация 7	$a=((a+_{7,1}KS[10])+_{7,2}d)>>t_7, c=c+_{7,3}(a*_7 b)$,
	итерация 8	$b=((b+_{8,1}KS[11])+_{8,2}a)>>t_8, d=d+_{8,3}(b*_8 c)$,
	...	...
...	...	...
	...	...
	итерация 33	$c=((c+_{33,1}KS[36])+_{33,2}b)>>t_{33}, a=a+_{33,3}(c*_{33} d)$,
	итерация 34	$d=((d+_{34,1}KS[37])+_{34,2}c)>>t_{34}, b=b+_{34,3}(d*_{34} a)$,
цикл 9	итерация 35	$a=((a+_{35,1}KS[38])+_{35,2}d)>>t_{35}, c=c+_{35,3}(a*_{35} b)$,
	итерация 36	$b=((b+_{36,1}KS[39])+_{36,2}a)>>t_{36}, d=d+_{36,3}(b*_{36} c)$,
конечное наложение ключа		$a = a +_5 KS[41], b = b +_6 KS[40],$
		$c = c +_7 KS[43], d = d +_8 KS[42]$

Индексы при знаках + и * и букве t говорят о том, что данные объекты являются параметрами алгоритма Викер-98, которые должны быть явно определены для каждой конкретной реализации алгоритма. При этом знак + указывает на обратимые операции, а знак * - на необратимые. Значения величин t заключено в отрезке от 1 до 31.

В тех же обозначениях алгоритм декодирования имеет итерации вида:

$$z = z - (x * v),\ x = ((x >> t) - y) - KS[] \quad (2)$$

где - обозначает обратную операцию к соответствующей операции + из формулы (1).

Используя знаки - с индексами для обозначения обратных операций к операциям + с тем же индексом, схему алгоритма декодирования можно представит в виде:

Таблица 2.

Схема алгоритма декодирования.

Начальное ключа	наложение	$a = a -_5 KS[41], b = b -_6 KS[40],$
		$c = c -_7 KS[43], d = d -_8 KS[42]$

	итерация 1	$d=d-_{36,3}(b*_{36}c),b=((b>>(32-t_{36})-_{36,2}a)-_{36,1}KS[39]$
	итерация 2	,
цикл 1	итерация 3	$c=c-_{35,3}(a*_{35}b),a=((a>>(32-t_{35})-_{35,2}d)-_{36,1}KS[38],$
	итерация 4	$b=b-_{34,3}(d*_{34}a),d=((d>>(32-t_{34})-_{34,2}c)-_{34,1}KS[37]$
		,
		$a=a-_{33,3}(c*_{33}d),c=c((c>>(32-t_{33})-_{33,2}b)-_{33,1}KS[36$
		]
	итерация 5	$d=d-_{32,3}(b*_{32}c),b=((b>>(32-t_{32})-_{32,2}a)-_{32,1}KS[35]$
	итерация 6	,
цикл 2	итерация 7	$c=c-_{31,3}(a*_{31}b),a=((a>>(32-t_{31})-_{31,2}d)-_{31,1}KS[34],$
	итерация 8	$b=b-_{30,3}(d*_{30}a),d=((d>>(32-t_{30})-_{30,2}c)-_{30,1}KS[33]$
		,
		$a=a-_{29,3}(c*_{29}d),c=c((c>>(32-t_{29})-_{29,2}b)-_{29,1}KS[32$
		]
	...	...
...	...	...
	...	...
	итерация 33	$d=d-_{4,3}(b*_{4}c),b=((b>>(32-t_{4})-_{4,2}a)-_{4,1}KS[7],$
	итерация 34	$c=c-_{3,3}(a*_{3}b),a=((a>>(32-t_{3})-_{3,2}d)-_{3,1}KS[6],$
цикл 9	итерация 35	$b=b-_{2,3}(d*_{2}a),d=((d>>(32-t_{2})-_{2,2}c)-_{2,1}KS[5],$
	итерация 36	$a=a-_{1,3}(c*_{1}d),c=c((c>>(32-t_{1})-_{1,2}b)-_{1,1}KS[4]$
конечное наложение ключа		$a = a -_{1} KS[0], b = b -_{2} KS[1],$
		$c = c -_{3} KS[2], d = d -_{4} KS[3]$

В качестве обратимых операций, обозначенных выше знаком + допускается использовать:

+ - сложение по модулю 2^{32} беззнаковых величин,

- - вычитание по модулю 2^{32} беззнаковых величин,

$\oplus$ - сложение по модулю 2 32-битных векторов.

Первые две из них являются обратными друг к другу, а третья является таковой по отношения самой к себе.

Необратимых операций предполагается использовать две:

& - конъюнкцию (умножение по модулю 2) 32-битных векторов,

$\mid$ - дизъюнкцию (не исключающее “или”) 32-битных векторов.

Как видим, регистры в операциях интерпретируются различными способами: то как беззнаковые числа, то как двоичные вектора. Поэтому, чтобы результаты вычислений не зависели от конкретной вычислительной платформы, придерживаемся соглашения, что младшие байты слова содержат младшие разряды беззнаковых чисел.

Конкретный вариант схемы Викер-98 получается при задании и фиксации всех проиндексированных параметров.

Принципы выбора параметров обсуждаются в следующей главе. Ниже приведен выбор параметров, осуществленный при первой публикации алгоритма.

Таблица 3. Вариант задания параметров схемы Викер-98.

Итерация	$+_1$	$+_2$	$+_3$	*	t
1	+	+	+	&	1
2	⊕	+	+	&	2
3	+	⊕	+	&	4
4	⊕	⊕	+	&	8
5	+	+	⊕	&	16
6	⊕	+	⊕	|	21
7	+	⊕	+	|	6
8	+	+	+	&	12
9	⊕	+	+	|	24
10	+	⊕	+	&	16
11	⊕	⊕	+	&	11
12	+	+	⊕	|	10
13	⊕	+	⊕	&	20
14	+	⊕	⊕	|	8
15	+	+	+	&	16
16	⊕	+	+	|	25
17	+	⊕	+	|	14
18	⊕	⊕	+	|	28
19	+	+	⊕	|	24
20	⊕	+	⊕	|	16
21	+	⊕	⊕	|	19
22	+	+	+	|	22
23	⊕	+	+	&	12
24	+	⊕	+	&	24
25	⊕	⊕	+	|	16
26	+	+	⊕	&	27
27	⊕	+	⊕	|	26
28	+	⊕	⊕	&	20
29	+	+	+	&	8
30	⊕	+	+	&	16
31	+	⊕	+	|	25
32	⊕	⊕	+	|	18
33	+	+	⊕	&	4
34	⊕	+	⊕	&	8
35	+	⊕	⊕	&	16
36	+	+	+	&	1
$+_1 = +$	$+_2 = +$	$+_3 = +$	$+_4 = +$	$+_5 = \oplus$	$+_6 = \oplus$	$+_7 = \oplus$	$+_8 = \oplus$

Обоснование выбора конкретных значений параметров алгоритма Викер-98 .

Алгоритм ориентирован в первую очередь на обработку данных хранимых в персональном компьютере. Данные в компьютере размещаются на логических устройствах, которые в большинстве случаев используют блочное представление данных. Применяемые блоки данных имеют размер выражаемый степенью двойки. Алгоритму удобнее работать с такими данными целыми блоками, а в случае, когда размер блока данных превышает ресурсы алгоритма, - с такими фрагментами данных, длины которых целое число раз укладывается в длину логического блока. Таким образом мы приходим к выводу о необходимости оперировать данными, длина которых является степенью двух.

Более конкретный размер используемых данных получается при фиксации типа микропроцессора, предназначенного для реализации алгоритма. В нашем случае таким микропроцессором является Пентиум. Он имеет 7 регистров общего назначения по 32 бита и еще примерно столько же специализированных регистров. Тем самым, максимальная длина данных (как степень двух) размещаемых в регистрах процессора составляет 256 бит. Однако непосредственно с таким количеством данных работать не возможно: требуется место для вспомогательных вычислений, специализированные регистры нельзя использовать “напрямую”, а - надо предварительно переслать содержимое в регистры общего назначения и т.д. Пересылка из специализированных регистров в регистры общего назначения и обратно так же часто не возможна “напрямую”, а предполагает обращение через общую память, не говоря о том, такие операции выполняются в особых режимах (например, в режиме отладки), что может приводить к конфликтным ситуациям. Использование общей же памяти приводит к замедлению операций по сравнению с операциями регистр-регистр. Из тех же соображений приходится исключить из рассмотрения регистры стека сопроцессора,

поскольку чтение и запись в них возможна только из общей памяти.

Поэтому эффективный алгоритм обработки данных должен не только размещать в регистрах входные данные и производные от них результаты вычислений, но и иметь свободные регистры для промежуточных вычислений. Применительно к процессору Пентиум, это означает , что блок обрабатываемых данных должен состоять из 128 бит, что соответствует 4-м 32-битным регистрам. Кроме всего прочего, такой малое число задействованных регистров позволяет в языках высокого уровня описывать эти данные как регистровые, что максимально приближает время выполнения такой программы к времени работы ассемблерной программы. Другими словами, это позволяет сблизить время реализации алгоритма на разных вычислительных платформах.

Уменьшать далее размер данных с 128 бит до 64 бит или 32 бит не имеет смысла, поскольку при фиксации ключа, блочный алгоритм обработки данных вырождается в подстановку соответствующего размера. Число таких подстановок фактически и является оценкой сверху надежности защиты информации этим алгоритмом. То есть при прочих равных условиях лучше выбирать максимально возможный размер данных. В нашем случае это - 128 бит.

Что касается размера операндов в элементарных операциях алгоритма, то процессор Пентиум выполняет операции над 32-битными словами с такой же скоростью как над 16-битными словами или байтами. Поэтому для ускорения вычислений нет смысла уменьшать размер операндом ниже 32-битной отметки.

Разбиение 32-битных слов на составные части имеет смысл только для улучшения зависимости между данными в ходе вычислений. Но в таком случае для обеспечения зависимости каждого результирующего бита от всех входных битов придется выполнить большее число операций, т.к. при разбиении пропорционально возрастает число аргументов. Поскольку длительность операций не зависит от длины аргументов, это приведет к замедлению алгоритма. Очевидно, что если бы вместо операций над усеченными данными мы выполняли бы операции над целыми 32-битными словами (при соответствующем сдвиге для совмещения прежних частей аргументов), то степень и сложность зависимости только бы возросли. Последнее, хотя и не обязательно, служит косвенным подтверждением возрастания степени защиты, обеспечиваемой алгоритмом. Другими словами, при

правильном подборе операции над целыми 32-битными словами дают большую защищенность, чем операции над их частями.

Конкретные виды операций определяются набором команд процессора. В случае процессора Пентиум у нас имеется достаточно широкий выбор таких операций, но учитывая перспективу реализации алгоритма на маломощных процессорах интеллектуальных карт, мы остановились на стандартном наборе операций: поразрядные булевские функции, операции сложения и вычитания по модулю 2^{32} и циклические сдвиги операндов. Этот список логически дополняют команды целочисленного умножения и деления, так же эффективно реализованные в процессоре Пентиум. Но мы сознательно исключили их из рассматриваемого списка операций, поскольку они наиболее трудоемки при моделировании на процессорах с ограниченным набором команд. Команды, использующие бит переноса, так же не рекомендованы к использованию из-за того, что языки высокого уровня не поддерживают напрямую такие операции и их реализация требует излишних операций. Из этих же соображений отвергнуты и команды, оперирующие с сдвоенными регистрами.

Сужать далее множество допустимых операций не представляется целесообразным: эти операции представлены в большинстве семейств микропроцессоров, и дальнейшее их сокращение не ведет уже к повышению эффективности реализации на соответствующей платформе. Более того, взятый набор команд охватывает две алгебраические структуры: двоичные вектора длиной 32 и целых чисел по модулю 2^{32}. Дальнейшее уменьшение набора чревато тем, что мы окажемся в одной алгебраической структуре и, тем самым, нам будет сложнее обеспечить надежность защиты информации.

Единственное ограничение, которое мы добавляем к выбранному множеству операций, состоит в том, величины сдвига операндов обязаны быть константами. Это свойство позволяет их проще реализовывать на мало мощных процессорах, не имеющих таких команд. И особенно полезны константные сдвиги при реализации алгоритма на специализированном микропроцессоре, где такие сдвиги представляются просто фиксированной коммутацией битов регистра. Кроме того, заметим, что именно переменная контекстно-зависимая величина сдвига в алгоритме RC-6 и позволила эффективно применить дифференциальный анализ.

Поэтому, следует надеяться, что фиксация величин сдвига позволит построить алгоритм с большей надежностью защиты информации, чем у RC-6 при тщательном подборе констант сдвига.

Специфика процессора Пентиум состоит в наличии двух арифметических устройств с соответствующими конвейерами, которые при определенных условиях могут выполнять команды одновременно. Условия параллельного функционирования процессора Пентиума лежат в общем русле распараллеливания операций. Такое распараллеливание достигается путем: использования различных регистров в последовательных операциях, не использования подряд идущих команд, зависящих от результатов предыдущих операций (в том числе - и от флагов, установленных ими), не применением подряд двух команд с обращением к общей памяти и т.д. Грубо говоря, для распараллеливания вычислений программу надо составлять так, чтобы на каждом шагу выполнялись независимые команды. Особенность Пентиума проявляется в том, что операции с памятью занимают столько же времени, как переписывание памяти в регистр с последующей операцией над регистрами. Поэтому при оптимизации вычислений стараются все операции с памятью расписать через чтение/запись памяти и регистровые команды, поскольку распараллеливание тем проще, чем больше имеется операций, тем более таких как чтение/запись.

Принцип независимости подряд идущих операций и реализован в представленном алгоритме Викер-98. Как видно из описания алгоритма, каждая итерация фактически состоит из двух частей: линейного преобразования над одним из регистров и добавлении к другому регистру булевой функции от двух оставшихся регистров. Хотя каждое из этих преобразований требует несколько команд процессора, они могут быть так разнесены во времени, что будут выполняются над независимыми регистрами.

Разработка алгоритма Викер-98 преследовала цель добиться предельной быстроты вычислений в классе блочных алгоритмов при достаточно высокой степени защиты информации. В этом разделе мы стараемся неформально обосновать первую часть утверждения: предельную эффективность вычислений. “Неформальность” проявляется в том, что приходится делать предположения о характеристиках всевозможных блочных алгоритмов защиты информации на основании лишь известных алгоритмов. В

качестве таких характеристик выступают: наименьшее число циклов в алгоритме и минимальная сложность одного цикла.

Если по отношению к еще неизвестным блочным алгоритмам такие предположения могут показаться надуманными, а выводы - спорными, то, по крайней мере, для известных блочных алгоритмов полученные результаты характеризуют алгоритм Викер-98 как наилучший по эффективности.

Цикл в блочном алгоритме обработки информации определяется как повторяющаяся часть вычислений с учетом участвующих аргументов.

Для каждого аргумента нельзя ограничиться только линейной операцией (операциями + и $\oplus$ из формального описания алгоритма Викер-98), поскольку в таком случае все преобразование останется линейным и его легко будет решить, даже не зная ключа.

Переходя к рассмотрению нелинейных операций (операций & и |), сразу же отметим, что непосредственно заменить аргумент на нелинейную функцию от него нельзя, ибо операции должны быть обратимы: наряду с алгоритмом кодирования должен существовать алгоритм декодирования. Для нелинейных операций остается только прибавление к аргументу нелинейной функции от других аргументов. Но одна эта операция хорошо приближается линейной, поэтому и весь алгоритм оказывается уязвимым для злоумышленника, хотя и не в такой степени, как при одних только линейных операциях (см. линейный анализ алгоритма Викер-98 в последующих главах). Более эффективнее сочетать линейные операции и сложение с нелинейной операцией.

Таким образом, минимально возможный набор операций, приходящийся на один аргумент в цикле, состоит из сложения с рассматриваемым аргументом, вычисления нелинейной операции над парой других аргументов и прибавления ее результата к выделенному аргументу и добавления ключевого слова. Как легко видеть, это в точности соответствует схеме алгоритма Викер-98, за исключением одной операции - циклического сдвига, добавленной нами для улучшения степени защиты информации. Без операции циклического сдвига для младших разрядов аргументов не будет разности между побитовыми операциями и операциям по модулю 2^{32}, что несколько уменьшает степень защиты информации и, косвенно, предполагает увеличение числа циклов, что в свою очередь ведет к снижению эффективности алгоритма.

Перейдем к рассмотрению практической реализации алгоритма Викер-98.

Первый взгляд на формальное описание алгоритма наводит на мысль, что распараллеливание фактически невозможно: в каждой итерации в каждой операции участвует один и тот же аргумент, а именно тот, который модифицируется в этой итерации на каждом шаге. На самом деле, все не так безнадежно.

Во-первых, прибавление ключевого слова, предполагает обращение к общей памяти, а последнее по времени выполнения равносильно предварительной загрузке ключевого слова в один из свободных регистров с последующем сложением его с тем же регистром. Загрузка же данных из памяти сочетается с любой операцией и может выполняться параллельно с ней.

Во-вторых, нелинейная операция предполагает копирование одного из аргументов в свободный регистр с последующем выполнения операции над ним и размещением результата в нем же. Операция копирования регистров, так же может быть выполнена параллельно любой операции.

В третьих, последний из модифицируемых в итерации аргументов задействуется потом только в конце следующей итерации, тем самым эту модификацию (и предшествующую ей нелинейную операцию) можно отложить и завершить в следующую итерацию.

Суммируя все сказанное, приведем возможный вариант реализации двух подряд идущих итераций на процессоре Пентиум. Для хранения внутренних параметров a, b, c и d будем соответственно использовать регистры EAX, EBX, ECX, EDX. Для вычисления нелинейной операции будем использовать регистр EDI, а для временного хранения ключевого слова - регистр ESI. Массив с именем K содержит ключевую последовательность.

Далее следует приведена реализация итераций 2 и 3, начиная с завершения операции a + = (c & d) первой итерации. Предполагается, что к началу вычислений значение d уже загружено в регистр EDI. Соответственно, мы вы прервемся в 3-ей итерации на том же месте: значение b уже загружено в EDI, но к операции c + = (a & b) мы не приступали. Текст программы разбит на два столбца, чтобы показать какие из операций будут выполняться параллельно.

Реализация двух подряд идущих итераций (на примере 2-й и 3-й итераций)

```
    MOV     ESI, K[5*4]                 AND EDI, ECX
    XOR     EDX, ESI                    ADD EAX, EDI
    ADD     EDX, ECX                        MOV     EDI,
EAX
    ROR     EDX, 2                      MOV  ESI, K[6*4]
    AND     EDI, EDX                    ADD  EAX, ESI
    ADD     ECX, EDI                    XOR  EAX, EDX
    ROR     EAX, 4                      MOV  EDI, EBX
    - - - - - - - - - - - -                      - - - - - - -
- - - - -
    MOV     ESI,K[7*4]                      AND
EDI,EAX
```

В представленном алгоритме Викер-98 имеется всего 9 циклов. По сравнению с известными блочными алгоритмами это количество фактически минимально.

Поскольку в предыдущих разделах была показана минимальность одного цикла блочного алгоритма защиты информации, то последующие линейный и дифференциальный анализы степени защищенности в какой-то мере можно рассматривать как обоснование минимальности числа таких циклов. Однако это относится только к классу рассмотренных итераций. В общем виде формально обосновать эту минимальность вряд ли удастся, ибо не ясно, какое число циклов потребуется для обеспечения того же уровня защиты при усложнении каждой итерации или всего цикла целиком.

Подводя итоги обоснования эффективности представленного алгоритма Викер-98, приведем реализацию начала и конца вычислений с учетом предварительного и заключительного наложения ключевых слов. Будем придерживаться соглашений из п. 1.2.1.

Начало алгоритма:

```
MOV   ESI, K[1*4]
ADD   EBX, ESI                    MOV  EDI, K[2*4]
ADD   ECX, EDI                    MOV  ESI, K[0]
ADD   EAX, ESI                    MOV  EDI, K[4*4]
ADD   ECX, EDI                    MOV  ESI, K[3*4]
ADD   ECX, EBX                    ADD  EDX, ESI
ROR   ECX, 1                      MOV  EDI, EDX
```

Конец алгоритма:

```
MOV   ESI, K[39*4]
ADD   EBX, ESI
ADD   EBX, EAX
ROR   EBX, 1
AND   EDI, EBX
ADD   EDX, EDI
XOR   EBX, ESI
XOR   EDX, EDI
XOR   ECX, ESI
AND   EDI, EAX
XOR   ECX, EDI
MOV   EDI, ECX
MOV   ESI, K[41*4]
XOR   EAX, ESI
MOV   ESI, K[40*4]
MOV   EDI, K[42*4]
MOV   ESI, K[43*4]
```

Как видим, алгоритм допускает полное распараллеливание на процессоре Пентиум.

В итоге кодирование одного блока информации занимает 135 тактов.

В данном разделе мы обоснуем выбор конкретных значений величин сдвига в итерациях с точки зрения общих (эвристических) подходов к построению надежных алгоритмов защиты информации. В дальнейшем сделанный выбор будет проанализирован в свете известных методов нападения на алгоритмы защиты информации.

Один из распространенных приемов нападения на защиту информации состоит в разбиении исходной задачи на несколько более простых подзадач. Простейшими примерами таких атак являются метод “согласования” и метод “дня рождения”, когда нападение основывается на подборе совместного решения двух соотношений, каждое из которых зависит от своего множества параметров. Очевидными предпосылками для возможности применения таких атак является неполная зависимость знаков кодированного текста от ключевой последовательности и исходного текста, позволяющая после частичного опробования последних разделить на независимые части, связывающие их соотношения.

Чтобы максимально обезопаситься от атак такого вида, алгоритмы защиты информации стоятся так, чтобы обеспечить полную зависимость выходных значений от входного текста и ключа и сделать эту зависимость наиболее сложной.

В нашем случае мы остановимся на подборе величин сдвига так, чтобы обеспечить эту максимальную зависимость

между битами исходного текста и ключевой последовательности.

При рассмотрении зависимости мы перейдем к упрощенной схеме алгоритма, а именно заменим все операции в алгоритме на соответствующие побитовые операции. Таким образом, в модифицированной схеме будет отсутствовать зависимость, возникающая за счет распространения бита переноса по всему регистру. Так алгоритм оперирующий с регистрами как векторами будем далее обозначать через Викер-В.

Таким образом, если мы обеспечим полную зависимость бит в алгоритме Викер-В, то у нас будут все основания считать таковым и исходный алгоритм Викер-98.

Алгоритм Викер-В обладает тем свойством, что все биты одного регистра на каждом шаге вычислений удовлетворяют одному и тому же соотношению: если бит с номером 0 есть некоторая функция f от каких-то битов исходного текста и ключа, то бит с номером i будет выражаться той же функцией f от битов исходного текста и ключа, номера который увеличены на i по модулю 32 по сравнению с номерами, от которых зависел бит с номером 0. Тем самым, зависимость бит от входного текста и ключа в алгоритме Викер-В достаточно рассматривать только для фиксированных бит регистров a, b, c и d, например, нулевых, что мы и будем далее делать.

Введем обозначения для величин сдвига и будем фиксировать зависимость нулевых бит регистров a, b, c и d от номеров бит исходного заполнения регистров и номеров бит ключевой последовательности, т.е. будем перечислять символьные выражения номеров бит соответствующих регистров и ключей, от которых зависит каждый из этих четырех бит на каждом шагу вычислений. При этом, естественно, будем отбрасывать одинаковые символьные выражения, как отвечающие одному и тому же биту.

Нижеследующие таблицы содержат число полученных символьных выражений для каждого из (нулевых разрядов) регистров a, b, c и d для каждой из итераций алгоритма Викер-В в зависимости от длины ключа (128, 196 и 256 бит).

При проведении подсчета на каждой итерации от скольких бит фактически зависит тот или иной бит надо прояснить не происходит ли сокращение (исключение) существенной зависимости в операциях рассматриваемого алгоритма. Другими словами надо выяснить не может ли исчезнуть формальная зависимость при сложении операндов, обладающих ее, или при каких других операциях алгоритма.

Обратимся к схеме алгоритма и будем иметь ввиду, что формальную зависимость мы представляем символьными выражениями номеров бит, от которых существенно зависит нулевой бит одного из взятых регистров. Если взглянуть на схему алгоритма, то мы видим последовательность вида:

... ; ror (x, t_{i-1}) ; y+= x*z ;

z+=x; ror (z, t_i) ; ... ;

где x, y, z - некоторые из регистров a, b, c и d. Из этого фрагмента видно, что после циклического сдвига x все символьные выражения, описывающие существенную зависимость нулевого бита x, будут содержать символ t_{i-1} , что сделает их отличными от всех предшествующих выражений. Следовательно, даже если y и z содержат одинаковые символьные выражения, то в x*z и y они входят в сочетании с разными символьными выражениями и поэтому не может произойти формального сокращения выражений. То есть множество существенных параметров для y будет объединением множеств существенных параметров y, x и z. Аналогично, из факта новизны символьных выражений для z вытекает, что в последующей операции z+=x сокращений символьных выражений так же не происходит и множество символьных параметров z пополняется множеством символьных параметров x.

Эти замечания позволяют оперировать с символьными выражениями для существенных зависимостей как с множествами: объединяя их в арифметических операциях или добавляя к каждому их элементу новый символ в операциях циклического сдвига. Программы для расчета таких зависимостей приведены в Приложении. По результатам их расчетов составлены следующие таблицы символьной зависимости бит регистров.

В столбах таблицы перечислены участвующие в вычислениях исходные данные, как 32-х битные величины. В отвечающих им клетке приведено число символьных выражений номеров (индексов) бит от которых зависит каждый бит регистра, указанного в строке. Символ * использован для указания того факта, что число символьных выражений достигло или превысило максимально возможную величину 32, т.е. * показывает, что формально достигнута полная зависимость всех бит регистра от всех бит рассматриваемых данных. Когда строчки всех четырех регистров начнут содержать одни * , это будет означать, что формально полная зависимость “всех от всех” достигнута.

Таблица 4.
Число существенных параметров
от которых зависят разряды регистров
a, b, c и d при ключе в 256 бит.

Итерация		a	b	c	d	k1	k2	k3	k4	k5	k6	k7	k8
	a:	1	1	1	1	1	1	1	1	0	0	0	0
	b:	0	1	0	0	0	1	0	0	0	0	0	0
	1	0	1	1	0	0	1	1	0	1	0	0	0
	c:												
	d:	0	0	0	1	0	0	0	1	0	0	0	0
	a:	1	1	1	1	1	1	1	1	1	0	0	0
	b:	1	3	2	2	1	3	2	2	2	1	0	0
	2	0	1	1	0	0	1	1	0	1	0	0	0
	c:												
	d:	0	1	1	1	0	1	1	1	1	1	0	0
	a:	1	2	2	2	1	2	2	2	2	1	1	0
	b:	1	3	2	2	1	3	2	2	2	1	0	0
3	**c:**	2	5	4	4	2	5	4	4	4	2	1	0
	d:	0	1	1	1	0	1	1	1	1	1	0	0
	a:	1	2	2	2	1	2	2	2	2	1	1	0
	b:	2	5	4	4	2	5	4	4	4	2	1	1
4	**c:**	2	5	4	4	2	5	4	4	4	2	1	0
	d:	4	10	8	8	4	10	8	8	8	4	2	1
	a:	8	20	16	16	8	20	16	16	16	8	4	2
	b:	2	5	4	4	2	5	4	4	4	2	1	1
5	**c:**	4	10	8	8	4	10	8	8	8	4	2	1
	d:	4	10	8	8	4	10	8	8	8	4	2	1
	a:	8	20	16	16	8	20	16	16	16	8	4	2
	b:	16	*	*	*	16	*	*	*	*	16	8	4
6	**c:**	4	10	8	8	4	10	8	8	8	4	2	1
	d:	8	20	16	16	8	20	16	16	16	8	4	2
	a:	16	*	*	*	16	*	*	*	*	16	8	4
	b:	16	*	*	*	16	*	*	*	*	16	8	4
7	**c:**	*	*	*	*	*	*	*	*	*	*	16	8
	d:	8	20	16	16	8	20	16	16	16	8	4	2
	a:	16	*	*	*	16	*	*	*	*	16	8	4
	b:	*	*	*	*	*	*	*	*	*	*	16	8
8	**c:**	*	*	*	*	*	*	*	*	*	*	16	8
	d:	*	*	*	*	*	*	*	*	*	*	*	16
	a:	*	*	*	*	*	*	*	*	*	*	*	*
	b:	*	*	*	*	*	*	*	*	*	*	16	8

9	**c:**	*	*	*	*	*	*	*	*	*	*	*	16
	d:	*	*	*	*	*	*	*	*	*	*	*	16
	a:	*	*	*	*	*	*	*	*	*	*	*	*
	b:	*	*	*	*	*	*	*	*	*	*	*	*
10	**c:**	*	*	*	*	*	*	*	*	*	*	*	16
	d:	*	*	*	*	*	*	*	*	*	*	*	*
	a:	*	*	*	*	*	*	*	*	*	*	*	*
	b:	*	*	*	*	*	*	*	*	*	*	*	*
11	**c:**	*	*	*	*	*	*	*	*	*	*	*	*
	d:	*	*	*	*	*	*	*	*	*	*	*	*

Таблица 5
Число существенных параметров
от которых зависят разряды регистров
a, b, c и d при ключе в 196 бит.

Итерация	**a**	**b**	**c**	**d**	**k1**	**k2**	**k3**	**k4**	**k5**	**k6**
a:	1	1	1	1	1	1	1	0	0	0
b:	0	1	0	0	0	1	0	0	0	0
1	0	1	1	0	0	1	1	0	1	0
c:										
d:	0	0	0	1	0	0	0	1	0	0
a:	1	1	1	1	1	1	1	1	1	0
b:	1	3	2	2	1	3	2	2	2	1
2	0	1	1	0	0	1	1	0	1	0
c:										
d:	0	1	1	1	0	1	1	1	1	1
a:	1	2	2	2	1	2	2	2	2	1
b:	1	3	2	2	1	3	2	2	2	1
3 **c:**	2	5	4	4	2	5	4	4	4	2
d:	0	1	1	1	0	1	1	1	1	1
a:	1	2	2	2	1	2	2	2	2	1
b:	2	5	4	4	2	5	4	4	4	2
4 **c:**	2	5	4	4	2	5	4	4	4	2
d:	4	10	8	8	4	10	8	8	8	4
a:	8	20	16	16	8	20	17	16	16	8
b:	2	5	4	4	2	5	4	4	4	2
5 **c:**	4	10	8	8	4	10	9	8	8	4
d:	4	10	8	8	4	10	8	8	8	4
a:	8	20	16	16	8	20	17	16	16	8
b:	16	*	*	*	16	*	*	*	*	16

6	**c:**	4	10	8	8	4	10	9	8	8	4
	d:	8	20	16	16	8	20	17	16	16	8
	a:	16	*	*	*	16	*	*	*	*	16
	b:	16	*	*	*	16	*	*	*	*	16
7	**c:**	*	*	*	*	*	*	*	*	*	*
	d:	8	20	16	16	8	20	17	16	16	8
	a:	16	*	*	*	16	*	*	*	*	16
	b:	*	*	*	*	*	*	*	*	*	*
8	**c:**	*	*	*	*	*	*	*	*	*	*
	d:	*	*	*	*	*	*	*	*	*	*
	a:	*	*	*	*	*	*	*	*	*	*
	b:	*	*	*	*	*	*	*	*	*	*
9	**c:**	*	*	*	*	*	*	*	*	*	*
	d:	*	*	*	*	*	*	*	*	*	*

Таблица 6
Число существенных параметров
от которых зависят разряды регистров
a, b, c и d при ключе в 128 бит.

Итерация		**a**	**b**	**c**	**d**	**k1**	**k2**	**k3**	**k4**
	a:	1	1	1	1	2	1	1	1
	b:	0	1	0	0	0	1	0	0
	1	0	1	1	0	1	1	1	0
	c:								
	d:	0	0	0	1	0	0	0	1
	a:	1	1	1	1	2	1	1	1
	b:	1	3	2	2	3	4	2	2
	2	0	1	1	0	1	1	1	0
	c:								
	d:	0	1	1	1	1	2	1	1
	a:	1	2	2	2	3	3	3	2
	b:	1	3	2	2	3	4	2	2
3	**c:**	2	5	4	4	6	7	5	4
	d:	0	1	1	1	1	2	1	1
	a:	1	2	2	2	3	3	3	2
	b:	2	5	4	4	6	7	5	4
4	**c:**	2	5	4	4	6	7	5	4
	d:	4	10	8	8	12	14	10	8
	a:	8	20	16	16	24	28	20	16
	b:	2	5	4	4	6	7	5	4
5	**c:**	4	10	8	8	12	14	10	8
	d:	4	10	8	8	12	14	10	8

	a:	8	20	16	16	24	28	20	16
	b:	16	*	*	*	*	*	*	*
6	**c:**	4	10	8	8	12	14	10	8
	d:	8	20	16	16	24	28	20	16
	a:	16	*	*	*	*	*	*	*
	b:	16	*	*	*	*	*	*	*
7	**c:**	*	*	*	*	*	*	*	*
	d:	8	20	16	16	24	28	20	16
	a:	16	*	*	*	*	*	*	*
	b:	*	*	*	*	*	*	*	*
8	**c:**	*	*	*	*	*	*	*	*
	d:	*	*	*	*	*	*	*	*
	a:	*	*	*	*	*	*	*	*
	b:	*	*	*	*	*	*	*	*
9	**c:**	*	*	*	*	*	*	*	*
	d:	*	*	*	*	*	*	*	*

Таким образом, для получения полной зависимости для 256 битного ключа требуется 11 итераций, а для ключей в 196 и 128 бит - 9 итераций, что в любом случае составляет 3 цикла алгоритма.

Различие символьных выражений не гарантирует различие их значений при подстановке вместо символов их значений. Поэтому полученные количества итераций являются оценкой снизу необходимого числа итераций для обеспечения полной зависимости всех регистров от всех исходных данных.

Переходя к подбору значений сдвигов в алгоритме Викер-98, были поставлены максимальные цели: реальная зависимость бит должна соответствовать таблицам 4 - 6 и кроме того эта же зависимость должна иметь место начиная с любой из итераций (*равномерная максимизация зависимости*). Проектируя блочный алгоритм защиты информации для широкого применения, нельзя изначально зафиксировать число итераций: в разных приложениях могут быть затребованы варианты алгоритма с разным числом итераций в зависимости от необходимого уровня обеспечения безопасности и имеющихся результатов анализа степени защищенности информации. Следовательно равномерность максимизации зависимости надо рассматривать не фиксированном отрезке, а на потенциально бесконечной последовательности. При этом, чтобы не отвлекаться на частный вид начала последовательности, будем считать ее бесконечной в обе стороны.

Экспериментальный поиск последовательности сдвигов для равномерной максимизации зависимости привел к следующему описанию множества решений.

Утверждение 1.

Бесконечная последовательность ... , t_{-2}, t_{-1}, t_0, t_1, t_2, ... натуральных чисел из интервала [1, 31] обеспечивает равномерную максимизацию зависимости в том и только том случае, если выполняются два условия:

а) для любого индекса i числа t_i и t_{i+5} содержат одну и ту же степень 2 в разложении на простые множители (т.е. $t_i = v*2^k$, $t_{i+5} = w*2^n$ и k = n , v, w - нечетные числа);

б) в разложении на простые множители элементов последовательности кратность числа 2 пробегает весь интервал [0,4] (т.е. в последовательности присутствуют все 5 видов чисел: $v*2^k$, k = 0, 1, 2, 3, 4 и v - нечетное).

Достаточность приведенных условий доказывается прямым выписыванием символьных выражений, от которых зависят биты регистров a, b, c и d, и составлением из них неравенств, выражающих то, что эти выражения различны. Эти неравенства затрагивают не более 11 подряд идущих значений t_i причем многие из них являются следствиями друг друга. Фактически число полученных неравенств не превосходит двухсот (в зависимости от размера ключевой последовательности) и их реально проверить для последовательностей описанного вида. Чтобы не утруждать себя выкладками по проверке сотни неравенств, проще переложить на компьютер перебор всех возможных отрезков заявленной последовательности длины 11 и для каждого выбранного набора значений провести проверку как себя ведет зависимость бит регистров a, b, c и d от итерации к итерации. Собственно это и было практически проделано.

Для доказательства необходимости условий а) и б) так же был применен компьютер: с его помощью составлялись списки отрезков последовательности длины 11, которые обеспечивали максимальное нарастание зависимости и которые допускали свое наращивание влево и вправо на 5 шагов с сохранением свойства максимального нарастания зависимости. Ясно, что только такие отрезки могут содержаться в бесконечной в обе стороны последовательности максимального нарастания зависимости. Более точно, в последовательности содержится подмножество этих отрезков, в котором множество собственных начал отрезков совпадает с множеством собственных концов.

Для сокращения перебора были использованы такие свойства:

1) если найден отрезок с максимальным нарастанием зависимости, то умножением на нечетное число по модулю 32 всех его члены, мы снова получаем отрезок максимального нарастания (как легко видеть, прежние равенства и неравенства не изменяются, т.к. отображение взаимно-однозначное);

2) отрезок длины 11 должен содержать нечетный элемент (т.к. при одних четных сдвигах четные и нечетные биты регистров никогда не пересекаются и значит не зависят друг от друга, что не может быть при полной зависимости).

Следствием их является то, что достаточно искать отрезки, где нечетный элемент в точности равен 1, причем расположенный в середине отрезка.

Отметим, что, вообще говоря, искомые последовательности могли бы зависеть от того, какая из таблиц 4, 5 или 6 рассматривается, т.е. какова длина ключа. Однако, проведенные вычисления показали, что всем условием удовлетворяют одни и те же бесконечные последовательности, которые и перечислены в условиях утверждения без относительно к длине ключа.

Возвращаясь к исходной практической задаче подбора последовательности сдвигов в алгоритме Викер-98 отметим, что условие равномерной максимизации зависимости сводится к набору условий относительно близлежащих величин сдвига, т.е. условие носит локальный характер. Поэтому можно утверждать:

Решение поставленной задачи является любой отрезок указанной последовательности требуемой длины. И наоборот, любое решение этой задачи отличается от отрезков данного вида только началом и/или концом длины не свыше 6.

Замечание.

В таблицах 4 - 6 учтено первоначальное наложение ключевой последовательности на защищаемую информацию и, тем самым, это наложение автоматически включилось в понятие равномерной максимизации зависимости. Вместе с тем, в итерации алгоритма Викер-98 такое наложение отсутствует: наличествует только прибавление одного из слов ключа к одному из регистров (что так же входило в равномерную максимизацию зависимости).

Поэтому при выборе внутренних сдвигов алгоритма Викер-98 правильнее было бы использовать уточненное таким

образом понятие равномерности: начальные сдвиги алгоритма выбирается в соответствии с таблицами, а последующие (когда полная зависимость достигнута) подбираются так, что максимизировалась зависимость в соответствии с фактическими вычислениями алгоритма (уже не содержащими начального наложения ключа).

Для так уточненного понятия равномерности были заново построены по приведенной выше методике бесконечные последовательности, которые как оказалось совпали с перечисленными. Тем самым, такое уточнение понятия не меняет описанного множества решений рассматриваемой задачи.

Отметим, что выбранный в алгоритме Викер-98 вариант величин сдвигов не только удовлетворяет сформулированному критерию, но еще более ужесточен: начиная с исходной пятерки сдвигов 1, 2, 4, 8, 16 каждая следующая получается умножением всей пятерки на нечетное число по модулю 32. Предназначение такого ограничения в ходе исследований не выявлено.

Всякое конечное детерминированное преобразование данных можно представить как алгебраическое отображение (над соответствующем алгебраическом объектом: группой, кольцом, полем и т.д.). Аналитические методы нападения на защищаемую информацию состоят в решении тем или иным способом соответствующего алгебраического уравнения относительно выбранных неизвестных (входного текста или ключевой последовательности).

В общем случае, высокая степень нелинейности преобразования обеспечивает защиту от таких аналитических методов нападения.

Наша задача состоит в подборе такого порядка обработки данных на каждой итерации, который обеспечивал бы максимально возможную степень нелинейности всего преобразования. При этом мы считаем заданным саму последовательность операций и их состав а рамках одной итерации. А именно, итерация состоит из линейной операции над регистрами, операции циклического сдвига регистра и прибавлению к одному из регистров нелинейной функции от двух других.

Как и в случае обеспечения максимальной скорости нарастания зависимости, нам удобно перейти к векторной версии алгоритма, изложенной в предыдущем разделе - алгоритму Викер-В. В результате степень нелинейности преобразования для каждого бита регистра будет одинакова.

Кроме того, в этом случае циклический сдвиг выступает как простая перестановка бит не меняющая степени нелинейности преобразования.

На первой итерации степень нелинейности для всех регистров равна 1 и порядок их обработки не имеет значения для степени нелинейности: после первой итерации один из регистров будет иметь степень нелинейности 2, а остальные - 1.

На второй итерации степень нелинейности могла бы достигнуть 4-х, если в нелинейной операции примут участие регистр степени нелинейности 2 с первой итерации и результат линейной операции с ним. Далее тем же способом можно достигнуть степени 4 и т.д. Однако в таком подходе рост степени нелинейности фактически обеспечивается лишь степенями одного из регистров (того, который в первой итерации получил степень 2), а степень нелинейности относительно других регистров растет значительно медленнее.

Более целесообразно на второй итерации ограничится достижением степени нелинейности 3, как результат умножения одного из регистров на регистр степени нелинейности 2 из первой итерации.

На последующих итерациях будем в нелинейной операции задействовать регистры с максимальной степенью нелинейности на тот момент. Это, очевидно, приведет к максимальной скорости нарастания степени нелинейности, которая составит:

Таблица 7.
Максимально возможная
степень нелинейности
преобразования Викер-В.

Итерация	**степень нелинейно**
1	2
2	3
3	5
4	8
5	13
6	21
7	34
8	55

9	89
10	144
11	233
12	377

В соответствии с выбранной стратегии и составлены операции одной итерации в алгоритме Викер.

Отметим, что указанные значения степени нелинейности являются по существу предельными: при фактическом выписывании старших членов и вычислении их произведений возможно повторение сомножителей, от чего степень нелинейности произведения будет меньше суммы степеней нелинейности сомножителей. Этого заведомо не происходит, пока в условиях максимального нарастании зависимости не наступит полная зависимость хотя бы по одному из регистров. Из таблиц 4 - 6 видно, что такая зависимость наступает на 6-ой итерации, что соответствует лишь 13-ой степени нелинейности.

Непосредственное выписывание в символьном виде старших членов регистров показало, что указанное нарастание степени может иметь место только до 8-ой итерации, что соответствует 55 степени нелинейности. Начиная с 9 итерации скорость нарастания степени нелинейности будет обязательно меньше вне зависимости от подбора величин циклических сдвигов.

Для скорости роста степени нелинейности можно так же как и для зависимости битов ввести понятие равномерности и поставить задачу поиска последовательности величин сдвига, обеспечивающих для каждой итерации максимальность роста степени нелинейности.

Результаты исследований возможных величин сдвига с этой точки зрения показали, что бесконечной последовательности для максимизации роста скорости нелинейности на последовательных 8 итераций не существует. Такая бесконечная последовательность существует только для отрезка в 7 итераций, чему отвечает скорость возрастания нелинейности с 1 до 34.

Более того, бесконечная последовательность величин сдвигов при максимизации скорости роста нелинейности на 7 итерациях устроена тем же образом, что и последовательность для равномерной максимизации зависимости (см. Утверждение 1).

Таким образом, Утверждение 1дает правило по выбору величин сдвига в алгоритме Викер-98 обеспечивающему выполнение сразу двух условий: равномерной максимизации зависимости битов регистров и равномерной максимизации скорости нарастания нелинейности.

При разработке алгоритмов защиты информации необходимо обращать внимание не только на само преобразование по кодированию данных, но также надо иметь ввиду и обратную операцию декодирования. Дело в том, что в общем случае нападающий имеет в своем распоряжении наборы пар исходный - закодированный текст и он может выбирать по какому отображению прямому или обратному строить свою атаку.

Поэтому те требования, которые мы предъявляли к процедуре кодирования должны быть обеспечены и в обратной операции декодирования. Но если в прямом отображении мы имели свободу в выборе параметров алгоритма, то обратное преобразование однозначно определяется по нему и нам остается рассмотреть как наши условия выполнены для него.

Как и в операции кодирования будем рассматривать упрощенный алгоритм Викер-В, в котором все операции булевы, а зависимости всех бит регистра отличаются только нумерацией параметров.

Если формально взглянуть на последовательность итераций в процедуре декодирования, то кажется что степень нелинейности должна удваивается за каждые две итерации. Однако это далеко не так. Причина заключается в том, что каждой операции умножения (в которых собственно и происходит повышение степени нелинейности) предшествует операция сложения ее операндов. Если же произведение представить через аргументы до сложения и раскрыть скобки, то оказывается, что один сомножитель умножается сам на себя. При этом этот сомножитель имеет максимальную степень, а поскольку мы оперируем в поле с характеристикой 2, то квадрат этого сомножителя равен самому сомножителю, т.е. удвоение степени нелинейности не происходит.

Если мы учтем эту специфику операций, то возможное нарастание степени нелинейности в операции декодирования будет иметь вид:

Таблица 8.

Максимально возможная
степень нелинейности обратного
преобразования Викер-В.

Итерация	степень нелинейности
1	2
2	2
3	3
4	4
5	5
6	7
7	9
8	12
9	16
10	21
11	28
12	37
13	49
14	65
15	86
16	114
17	151
18	200
19	265

Как видим, скорость нарастания степени нелинейности в процедуре декодирования примерно вдвое меньше чем в прямой операции кодирования.

Для более точной оценки скорости нарастания степени нелинейности мы выписали максимальные по степени мономы в символьном виде через переменные сдвига и формально сравнили полученные символьные выражения. Получилось, что представленная таблица не противоречит выписанным выражениям только по 15 строчку: на 16 итерации обязательно имеет место совпадение сомножителей в произведении старших мономов и поэтому степень нелинейности 114 на этом шаге не достигается ни при каком выборе величин сдвига.

Что касается возможности обеспечить скорость нарастания степени нелинейности в соответствии с первыми 15 строчками таблицы, то для этой цели был осуществлен экспериментальный перебор величин сдвига. Как и для прямой операции кодирования ставилась цель обеспечит равномерность этого свойства: начиная с любой итерации

скорость нарастания нелинейности должна быть максимальна (нелинейность при этом измеряется относительно разрядов регистров с предшествующей итерации). Исследования показали, что такая равномерность может быть обеспечена только для первых 8 строк таблицы: начиная с любой итерации 8 шагов степень нелинейности будет возрастать в соответствии с таблицей.

Для 9 и далее строк такая равномерность отсутствует, хотя и существуют короткие решения с соответствующей таблице степенью нелинейности, но они не могут быть продолжены с сохранением этого свойства дальше.

Отметим, что равномерность для 8 шагов обеспечивается на последовательностях сдвига, перечисленных в Утверждении 1, что лишний раз подчеркивает правильность выбора величин сдвига в алгоритме Викер-98.

Исследование скорости нарастания функциональной зависимости в процедуре декодирования целесообразно, как и ранее, проводить для упрощенной схемы алгоритма - для алгоритма Викер-В, в котором все операции над регистрами поразрядные и, как следствие этого, функциональные зависимости для бит одного регистра отличаются друг от друга лишь сдвигом нумерации по модулю 32.

Оценку сверху на величину функциональной зависимости дает выписывание этой зависимости для нулевых бит регистров через величины сдвигов: совпадение символьных выражений говорит об одинаковой зависимости.

Оперируя символьными выражениями, надо убедится, что прежние зависимости не пропадают шаг за шагом, а лишь пополняются новыми. Для этого обратимся к схеме декодирования. Как видим, операциям прибавления к регистру предшествует операция циклического сдвига этого регистра. Для символьных выражений циклический сдвиг означает, что ко всем выявленным зависимостям добавляется новое слагаемое, отвечающее величине сдвига, т.е. циклический сдвиг делает все прежние символьные выражения для этого регистра - уникальными. Поэтому последующее сложение не может привести к повторению символьных выражений и, тем более, - к их сокращению.

Что касается нелинейной операции, то в ней участвуют два аргумента, которые на предыдущем шаге складывались. Поэтому, если нелинейную операцию выразить через предшествующие сложению значения, то мы фактически получим нелинейную операцию с участием только что циклически сдвинутого регистром и слагаемого. Как уже

отмечалось регистр после сдвига имеет уникальные символьные выражения, поэтому в нелинейной операции прежние символьные выражения будут уже фигурировать как сомножители уникальных выражений, т.е. не может произойти формального сокращения участвующих символьных выражений.

Подсчет числа различных символьных выражений для бит одного регистра имеет смысл, пока это число не превышает длины регистра - 32, начиная с этого места мы формально имеем полную зависимость от всех бит регистра.

Ставя цель добиться равномерного роста функциональной зависимости мы должны иметь ввиду два случая: с учетом начального наложения ключей и без такового.

Результаты символьных вычислений представлены в таблицах 9 - 11, где символ * как и прежде обозначает наступление полной формальной зависимости, а выражения в скобках отвечают случаю вычислений без начального наложения ключей.

Таблица 9.
Оценка сверху
числа существенных параметров
от которых зависят разряды регистров
a, b, c и d в процедуре декодирования
при ключе в 256 бит.

Итерация	**a**	**b**	**c**	**d**	**k1**	**k2**	**k3**	**k4**	**k5**	**k6**	**k7**	**k8**
a:	1	0	0	0	0	1(0)	0	0	0	0	0	0
b:	1	1	0	0	1(0)	1(0)	0	0	0	0	0	1
1	0	0	1	0	0	0	0	1(0)	0	0	0	0
c:												
d:	0	1	1	1	1	0	1(0)	1(0)	0	0	0	0
a:	1	1	1	1	1(0)	1(0)	1(0)	1(0)	0	0	1	0
b:	1	1	0	0	1(0)	1(0)	0	0	0	0	0	1
2	1	1	1	0	1(0)	1(0)	0	1(0)	0	0	0	1
c:												
d:	0	1	1	1	1(0)	0	1(0)	1(0)	0	0	0	0
a:	1	1	1	1	1(0)	1(0)	1(0)	1(0)	0	0	1	0
b:	2	2	1	1	2(0)	2(0)	1(0)	1(0)	0	0	1	1
3 **c:**	1	1	1	0	1(0)	1(0)	0	1(0)	0	0	0	1
d:	1	2	2	1	2(0)	1(0)	1(0)	2(0)	0	1	0	1

	a:	2	3	2	2	3(0)	2(0)	2(0)	2(0)	0	1	1	1
	b:	2	2	1	1	2(0)	2(0)	1(0)	1(0)	0	0	1	1
4	**c:**	3	3	2	1	3(0)	3(0)	1(0)	2(0)	1	0	1	2
	d:	1	2	2	1	2(0)	1(0)	1(0)	2(0)	0	1	0	1
	a:	2	3	2	2	3(0)	2(0)	2(0)	2(0)	0	1	1	1
	b:	4	5	3	3	5(0)	4(0)	3(0)	3(1)	0	1	2	2
5	**c:**	3	3	2	1	3(0)	3(0)	1(0)	2(0)	1	0	1	2
	d:	3	4	3	2	4(0)	3(0)	2(0)	3(0)	1	1	1	2
	a:	5	7	5	4	7(0)	5(0)	4(1)	5(0)	1	2	2	3
	b:	4	5	3	3	5(0)	4(0)	3(0)	3(1)	0	1	2	2
6	**c:**	5	6	4	3	6(0)	5(0)	3(0)	4(1)	1	1	2	3
	d:	3	4	3	2	4(0)	3(0)	2(0)	3(0)	1	1	1	2
	a:	5	7	5	4	7(0)	5(0)	4(1)	5(0)	1	2	2	3
	b:	7	9	6	5	9(0)	7(0)	5(1)	6(1)	1	2	3	4
7	**c:**	5	6	4	3	6(0)	5(0)	7(0)	3(1)	1	1	2	3
	d:	8	10	7	5	10(0)	8(1)	5(0)	7(1)	2	2	3	5
	a:	10	13	9	7	13(0)	10(1)	7(1)	9(1)	2	3	4	6
	b:	7	9	6	5	9(0)	7(0)	5(1)	6(1)	1	2	3	4
8	**c:**	12	15	10	8	15(1)	12(0)	8(1)	10(2)	2	3	5	7
	d:	8	10	7	5	10(0)	8(1)	5(0)	7(1)	2	2	3	5
	a:	10	13	9	7	13(0)	10(1)	7(1)	9(1)	2	3	4	6
	b:	17	22	15	12	22(0)	17(1)	12(2)	15(2)	3	5	7	10
9	**c:**	12	15	10	8	15(1)	12(0)	8(1)	10(2)	2	3	5	7
	d:	15	19	13	10	19(1)	15(1)	10(1)	13(2)	3	4	6	9
	a:	25	*	22	17	*(1)	25(2)	17(2)	22(3)	5	7	10	15
	b:	17	22	15	12	22(0)	17(1)	12(2)	15(2)	3	5	7	10
10	**c:**	22	28	19	15	28(1)	22(1)	15(2)	19(3)	4	6	9	13
	d:	15	19	13	10	19(1)	15(1)	10(1)	13(2)	3	4	6	9
	a:	25	*	22	17	*(1)	25(2)	17(2)	22(3)	5	7	10	15
	b:	*	*	28	22	*(1)	*(2)	22(3)	28(4)	6	9	13	19
11	**c:**	22	28	19	15	28(1)	22(1)	15(2)	19(3)	4	6	9	13
	d:	*	*	*	25	*(2)	*(2)	25(3)	*(5)	7	10	15	22

	a:	*	*	*	*	*(2)	*(3)	*(4)	*(6)	9	13	19	28
	b:	*	*	28	22	*(1)	*(2)	22(3)	28(4)	6	9	13	19
12	**c:**	*	*	*	*	*(2)	*(3)	*(5)	*(7)	10	15	22	*
	d:	*	*	*	*	*(2)	*(2)	*(3)	*(5)	7	10	15	22
	a:	*	*	*	*	*(2)	*(3)	*(4)	*(6)	9	13	19	28
	b:	*	*	*	*	*(3)	*(5)	*(7)	*(10)	15	22	*	*
13	**c:**	*	*	*	*	*(2)	*(3)	*(5)	*(7)	10	15	22	*
	d:	*	*	*	*	*(3)	*(4)	*(6)	*(9)	13	19	28	*
	a:	*	*	*	*	*(5)	*(7)	*(10)	*(15)	22	*	*	*
	b:	*	*	*	*	*(3)	*(5)	*(7)	*(10)	15	22	*	*
14	**c:**	*	*	*	*	*(4)	*(6)	*(9)	*(13)	19	28	*	*
	d:	*	*	*	*	*(3)	*(4)	*(6)	*(9)	13	19	28	*
	a:	*	*	*	*	*(5)	*(7)	*(10)	*(15)	22	*	*	*
	b:	*	*	*	*	*(6)	*(9)	*(13)	*(19)	28	*	*	*
15	**c:**	*	*	*	*	*(4)	*(6)	*(9)	*(13)	19	28	*	*
	d:	*	*	*	*	*(7)	*(10)	*(15)	*(22)	*	*	*	*
	a:	*	*	*	*	*(9)	*(13)	*(19)	*(28)	*	*	*	*
	b:	*	*	*	*	*(6)	*(9)	*(13)	*(19)	28	*	*	*
16	**c:**	*	*	*	*	*(10)	*(15)	*(22)	*	*	*	*	*
	d:	*	*	*	*	*(7)	*(10)	*(15)	*(22)	*	*	*	*
	a:	*	*	*	*	*(9)	*(13)	*(19)	*(28)	*	*	*	*
	b:	*	*	*	*	*(15)	*(22)	*	*	*	*	*	*
17	**c:**	*	*	*	*	*(10)	*(15)	*(22)	*	*	*	*	*
	d:	*	*	*	*	*(13)	*(19)	*(28)	*	*	*	*	*
	a:	*	*	*	*	*(22)	*	*	*	*	*	*	*
	b:	*	*	*	*	*(15)	*(22)	*	*	*	*	*	*
18	**c:**	*	*	*	*	*(19)	*(28)	*	*	*	*	*	*
	d:	*	*	*	*	*(13)	*(19)	*(28	*	*	*	*	*

							)						
	a:	*	*	*	*	*(22)	*	*	*	*	*	*	*
	b:	*	*	*	*	*(28)	*	*	*	*	*	*	*
19	**c:**	*	*	*	*	*(19)	*(28)	*	*	*	*	*	*
	d:	*	*	*	*	*	*	*	*	*	*	*	*
	a:	*	*	*	*	*	*	*	*	*	*	*	*
	b:	*	*	*	*	*(28)	*	*	*	*	*	*	*
20	**c:**	*	*	*	*	*	*	*	*	*	*	*	*
	d:	*	*	*	*	*	*	*	*	*	*	*	*
	a:	*	*	*	*	*	*	*	*	*	*	*	*
	b:	*	*	*	*	*	*	*	*	*	*	*	*
21	**c:**	*	*	*	*	*	*	*	*	*	*	*	*
	d:	*	*	*	*	*	*	*	*	*	*	*	*

Таблица 10
Оценка сверху
числа существенных параметров
от которых зависят разряды регистров
a, b, c и d в операции декодирования
при ключе в 196 бит.

Итерация		**a**	**b**	**c**	**d**	**k1**	**k2**	**k3**	**k4**	**k5**	**k6**
	a:	1	0	0	0	0	0	0	0	0	1(0)
	b:	1	1	0	0	0	0	0	1	1(0)	1(0)
	1	0	0	1	0	0	1(0)	0	0	0	0
	c:										
	d:	0	1	1	1	1(0)	1(0)	0	0	1(0)	0
	a:	1	1	1	1	1(0)	1(0)	1	0	1(0)	1(0)
	b:	1	1	0	0	0	0	0	1	1(0)	1(0)
	2	1	1	1	0	0	1(0)	0	1	1(0)	1(0)
	c:										
	d:	0	1	1	1	1(0)	1(0)	0	0	1(0)	0
	a:	1	1	1	1	1(0)	1(0)	1	0	1(0)	1(0)
	b:	2	2	1	1	1(0)	1(0)	1	1	2(0)	2(0)
3	**c:**	1	1	1	0	0	1(0)	0	1	1(0)	1(0)

	d:	1	2	2	1	1(0)	2(1)	0	1	2(0)	1(0)
	a:	2	3	2	2	2(0)	2(1)	1	1	3(0)	2(0)
	b:	2	2	1	1	1(0)	1(0)	1	1	2(0)	2(0)
4	**c:**	3	3	2	1	1	2(0)	1	2	3(0)	3(0)
	d:	1	2	2	1	1(0)	2(1)	0	1	2(0)	1(0)
	a:	2	3	2	2	2(0)	2(1)	1	1	3(0)	2(0)
	b:	4	5	3	3	3(0)	3(1)	2	2	5(0)	4(1)
5	**c:**	3	3	2	1	1	2(0)	1	2	3(0)	3(0)
	d:	3	4	3	2	2(1)	3(1)	1	2	4(0)	3(0)
	a:	5	7	5	4	4(1)	5(2)	2	3	7(1)	5(0)
	b:	4	5	3	3	3(0)	3(1)	2	2	5(0)	4(1)
6	**c:**	5	6	4	3	3(1)	4(1)	2	3	6(0)	5(1)
	d:	3	4	3	2	2(1)	3(1)	1	2	4(0)	3(0)
	a:	5	7	5	4	4(1)	5(2)	2	3	7(1)	5(0)
	b:	7	9	6	5	5(1)	6(2)	3	4	9(1)	7(1)
7	**c:**	5	6	4	3	3(1)	4(1)	2	3	6(0)	5(1)
	d:	8	10	7	5	5(2)	7(2)	3	5	10(0)	8(0)
	a:	10	13	9	7	7(2)	9(3)	4	6	13(1)	10(1)
	b:	7	9	6	5	5(1)	6(2)	3	4	9(1)	7(1)
8	**c:**	12	15	10	8	8(2)	10(3)	5	7	15(1)	12(2)
	d:	8	10	7	5	5(2)	7(2)	3	5	10(0)	8(1)
	a:	10	13	9	7	7(2)	9(3)	4	6	13(1)	10(1)
	b:	17	22	15	12	12(3)	15(5)	7	10	22(2)	17(2)
9	**c:**	12	15	10	8	8(2)	10(3)	5	7	15(1)	12(2)
	d:	15	19	13	10	10(3)	13(4)	6	9	19(1)	15(2)
	a:	25	*	22	17	17(5)	22(7)	10	15	*(2)	25(3)
	b:	17	22	15	12	12(3)	15(5)	7	10	22(2)	17(2)
10	**c:**	22	28	19	15	15(4)	19(6)	9	13	28(2)	22(3)
	d:	15	19	13	10	10(3)	13(4)	6	9	19(1)	15(2)
	a:	25	*	22	17	17(5)	22(7)	10	15	*(2)	25(3)
	b:	*	*	28	22	22(6)	28(9)	13	19	*(3)	*(4)
11	**c:**	22	28	19	15	15(4)	19(6)	9	13	28(2)	22(3)
	d:	*	*	*	25	25(7)	*(10)	15	22	*(3)	*(5)
	a:	*	*	*	*	*(9)	*(13)	19	28	*(4)	*(6)
	b:	*	*	28	22	22(6)	28(9)	13	19	*(3)	*(4)
12	**c:**	*	*	*	*	*(10)	*(15)	22	*	*(5)	*(7)
	d:	*	*	*	25	25(7)	*(10)	15	22	*(3)	*(5)
	a:	*	*	*	*	*(9)	*(13)	19	28	*(4)	*(6)
	b:	*	*	*	*	*(15)	*(22)	*	*	*(7)	*(10)
13	**c:**	*	*	*	*	*(10)	*(15)	22	*	*(5)	*(7)
	d:	*	*	*	*	*(13)	*(19)	28	*	*(6)	*(9)
	a:	*	*	*	*	*(22)	*	*	*	*(10)	*(15)
	b:	*	*	*	*	*(15)	*(22)	*	*	*(7)	*(10)

14	**c:**	*	*	*	*	*(19)	*(28)	*	*	*(9)	*(13)
	d:	*	*	*	*	*(13)	*(19)	28	*	*(6)	*(9)
	a:	*	*	*	*	*(22)	*	*	*	*(10)	*(15)
	b:	*	*	*	*	*(28)	*	*	*	*(13)	*(19)
15	**c:**	*	*	*	*	*(19)	*(28)	*	*	*(9)	*(13)
	d:	*	*	*	*	*	*	*	*	*(15)	*(22)
	a:	*	*	*	*	*	*	*	*	*(19)	*(28)
	b:	*	*	*	*	*(28)	*	*	*	*(13)	*(19)
16	**c:**	*	*	*	*	*	*	*	*	*(22)	*
	d:	*	*	*	*	*	*	*	*	*(15)	*(22)
	a:	*	*	*	*	*	*	*	*	*(19)	*(28)
	b:	*	*	*	*	*	*	*	*	*	*
17	**c:**	*	*	*	*	*	*	*	*	*(22)	*
	d:	*	*	*	*	*	*	*	*	*(28)	*
	a:	*	*	*	*	*	*	*	*	*	*
	b:	*	*	*	*	*	*	*	*	*	*
18	**c:**	*	*	*	*	*	*	*	*	*	*
	d:	*	*	*	*	*	*	*	*	*(28)	*
	a:	*	*	*	*	*	*	*	*	*	*
	b:	*	*	*	*	*	*	*	*	*	*
19	**c:**	*	*	*	*	*	*	*	*	*	*
	d:	*	*	*	*	*	*	*	*	*	*

Таблица 12
Оценка сверху
числа существенных параметров
от которых зависят разряды регистров
a, b, c и d в операции декодирования
при ключе в 128 бит.

Итерация	**a**	**b**	**c**	**d**	**k1**	**k2**	**k3**	**k4**
a:	1	0	0	0	0	1(0)	0	0
b:	1	1	0	0	1(0)	1(0)	0	1(0)
1	0	0	1	0	0	0	0	1(0)
c:								
d:	0	1	1	1	1(0)	0	1(0)	1(0)
a:	1	1	1	1	1(0)	1(0)	1	1(0)
b:	1	1	0	0	1(0)	1(0)	0	1
2	1	1	1	0	1(0)	1(0)	0	1
c:								
d:	0	1	1	1	1(0)	0	1(0)	1(0)

	a:	1	1	1	1	1(0)	1(0)	1	1(0)
	b:	2	2	1	1	2(0)	2(0)	1	1
3	**c:**	1	1	1	0	1(0)	1(0)	0	1
	d:	1	2	2	1	2(0)	1	1(0)	2(1)
	a:	2	3	2	2	3(0)	2(1)	2(1)	2(1)
	b:	2	2	1	1	2(0)	2(0)	1	1
4	**c:**	3	3	2	1	3(1)	3(0)	1	2
	d:	1	2	2	1	2(0)	1	1(0)	2(1)
	a:	2	3	2	2	3(0)	2(1)	2(1)	2(1)
	b:	4	5	3	3	5(0)	4(1)	3(2)	3(2)
5	**c:**	3	3	2	1	3(1)	3(0)	1(1)	2(2)
	d:	3	4	3	2	4(1)	3(1)	2(1)	3(2)
	a:	5	7	5	4	7(1)	5(2)	4(2)	5(3)
	b:	4	5	3	3	5(0)	4(1)	3(2)	3(2)
6	**c:**	5	6	4	3	6(1)	5(1)	3(2)	4(3)
	d:	3	4	3	2	4(1)	3(1)	2(1)	3(2)
	a:	5	7	5	4	7(1)	5(2)	4(2)	5(3)
	b:	7	9	6	5	9(1)	7(2)	5(3)	6(4)
7	**c:**	5	6	4	3	6(1)	5(1)	3(2)	4(3)
	d:	8	10	7	5	10(2)	8(2)	5(3)	7(5)
	a:	10	13	9	7	13(2)	10(3)	7(4)	9(6)
	b:	7	9	6	5	9(1)	7(2)	5(3)	6(4)
8	**c:**	12	15	10	8	15(2)	12(3)	8(5)	10(7)
	d:	8	10	7	5	10(2)	8(2)	5(3)	7(5)
	a:	10	13	9	7	13(2)	10(3)	7(4)	9(6)
	b:	17	22	15	12	22(1)	17(2)	12(3)	15(4)
9	**c:**	12	15	10	8	15(2)	12(3)	8(5)	10(7)
	d:	15	19	13	10	19(3)	15(4)	10(6)	13(9)
	a:	25	*	22	17	*(5)	25(7)	17(10)	22(15)
	b:	17	22	15	12	22(3)	17(5)	12(7)	15(10)
	10	22	28	19	15	28(4)	22(6)	15(9)	19(13)
	c:								
	d:	15	19	13	10	19(3)	15(4)	10(6)	13(9)
	a:	25	*	22	17	*(5)	25(7)	17(10)	22(15)
	b:	*	*	28	22	*(6)	*(9)	22(13)	28(19)
	11	22	28	19	15	28(4)	22(6)	15(9)	19(13)
	c:								
	d:	*	*	*	25	*(7)	*(10)	25(15)	*(22)
	a:	*	*	*	*	*(9)	*(13)	*(19)	*(28)
	b:	*	*	28	22	*(6)	*(9)	22(13)	28(19)
	12	*	*	*	*	*(10)	*(15)	*(22)	*
	c:								
	d:	*	*	*	25	*(7)	*(10)	25(15)	*(22)

a:	*	*	*	*	*(9)	*(13)	*(19)	*(28)
b:	*	*	*	*	*(15)	*(22)	*	*
13	*	*	*	*	*(10)	*(15)	*(22)	*
c:								
d:	*	*	*	*	*(13)	*(19)	*(28)	*
a:	*	*	*	*	*(22)	*	*	*
b:	*	*	*	*	*(15)	*(22)	*	*
14	*	*	*	*	*(19)	*(28)	*	*
c:								
d:	*	*	*	*	*(13)	*(19)	*(28)	*
a:	*	*	*	*	*(22)	*	*	*
b:	*	*	*	*	*(28)	*	*	*
15	*	*	*	*	*(19)	*(28)	*	*
c:								
d:	*	*	*	*	*	*	*	*
a:	*	*	*	*	*	*	*	*
b:	*	*	*	*	*(28)	*	*	*
16	*	*	*	*	*	*	*	*
c:								
d:	*	*	*	*	*	*	*	*
a:	*	*	*	*	*	*	*	*
b:	*	*	*	*	*	*	*	*
17	*	*	*	*	*	*	*	*
c:								
d:	*	*	*	*	*	*	*	*

Из таблиц 9 - 12 видно, что скорость нарастания функциональной зависимости для операции декодирования уже изначально примерно в полтора раза меньше той же скорости для прямой процедуры кодирования.

В таблицах учтена лишь разность между символьными выражениями, выражающими зависимость. Чтобы указанные оценки стали достижимы необходимо подобрать величины сдвига так, что для них значения символьных выражений действительно стали бы различны.

Поиск таких величин сдвига велся перебором значений. При этом, как и в прямой операции кодирования исследовалась возможность обеспечить равномерность скорости нарастания зависимости по итерациям.

Результаты поиска таковы:

а) В полном объеме скорость нарастания зависимости в соответствии с таблицами 9 - 12 достигнуть не удается.

б) Равномерность скорости нарастания зависимости можно обеспечить только для первых пяти строчек таблиц. Имеются решения, которые поддерживают скорость

нарастания зависимости более, чем для 5-ти строчек таблиц, но эти решения не допускают неограниченного их продолжения так, чтобы на каждом их отрезке скорость зависимости была не меньше, чем на 5-ти строчках таблиц.

в) Последовательности сдвигов, обеспечивающие равномерность нарастания зависимости по первым 5 строкам таблиц, устроены в соответствии с Утверждением 1.

г) Для таких последовательностей фактическая скорость нарастания зависимости отличается от табличной не более чем на 4 итерации, т.е. на один цикл.

Таким образом, произведенный выбор величин сдвига в алгоритме Викер-98 обеспечивает для операции декодирования равномерность и максимальность нарастания зависимости на каждых 5 итерациях и общую скорость нарастания зависимости лишь на цикл меньшую от предельно возможной. Эта скорость почти вдвое меньше скорости нарастания зависимости в прямой операции кодирования.

1. Каждая итерация в алгоритме Викер-98 содержит минимально возможный набор операций.
2. Число циклов в алгоритме Викер-98 одно из самых меньших среди блочных алгоритмов защиты информации.
3. Последовательность обработки данных в каждой итерации подобрана так, чтобы максимально увеличить рост степени нелинейности преобразования от итерации к итерации.
4. Сами операции в каждой итерации легко разделяются по независимым операндам, что дает возможность полностью распараллелить вычисления на процессорах семейства Пентиум. Для них вся процедура кодирования или декодирования укладываются в 135 тактов.
5. Применение фиксированных величин циклических сдвигов с одной стороны упрощает реализацию алгоритма на слабых микропроцессорах и специализированных чипах, а с другой - дает возможность подобрать их с позиции обеспечения надежности защиты информации.
6. Эффективно описана последовательность величин сдвига, на которой достигается максимум функциональной зависимость разрядов кодированного текста от исходного текста и ключа, причем этом максимум обеспечивается начиная с любой из итераций (равномерность функциональной зависимости).
7. Та же последовательность является оптимальной и для обратной операции декодирования.

8. Алгоритм Викер-98 имеет ярко выраженные асимметрические свойства: прямая операция кодирования превосходит обратную операцию декодирования по скоростям роста нелинейности и функциональной зависимости почти в два раза. Эти свойства обратной операции не могут быть принципиально улучшены за счет подпора другой последовательности циклических сдвигов.