МАТЕМАТИЧЕСКИЕ ОСНОВЫ ИНФОРМАТИКИ И ПРОГРАММИРОВАНИЯ

УДК 004.056.5

АВТОМАТИЧЕСКАЯ ГЕНЕРАЦИЯ ХЭШ-ФУНКЦИЙ ДЛЯ ОБФУСКАЦИИ ПРОГРАММНОГО КОДА

Р. К. Лебедев

Новосибирский государственный университет, г. Новосибирск, Россия

Рассмотрены особенности применения хэш-функций для запутывания программного кода, а также проблемы использования в этих целях существующих хэш-функций. С учётом этих особенностей и проблем предлагается метод автоматической генерации хэш-функций, основанный на подходе генетического программирования. Предложены методы оценки устойчивости хэш-функций к автоматическим атакам поиска первого прообраза, основанным на использовании SMT-решателей, и к случайным коллизиям. Проведена оценка генерируемых функций, а также предложен метод быстрого обнаружения слабых экземпляров, позволяющий значительно повысить устойчивость получаемых хэш-функций к атакам.

Ключевые слова: *обфускация, хэш-функция, генетическое программирование, лавинный эффект, SMT-решатель.*

DOI 10.17223/20710410/50/8

AUTOMATIC GENERATION OF HASH FUNCTIONS FOR PROGRAM CODE OBFUSCATION

R. K. Lebedev

Novosibirsk State University, Novosibirsk, Russia

E-mail: n0n3m4@gmail.com

The specifics of hash function applications in code obfuscation are considered, as well as the disadvantages of currently existing hash functions for this purpose. Considering these specifics and disadvantages, an automatic hash function generation method is proposed, that is based on a genetic programming approach. Methods of measuring the hash function resilience to automated preimage attacks based on SMT solvers usage and random collision resistance are also proposed. Generated functions were evaluated and a method of weak function detection is proposed, that allows to increase the resilience of generated functions to attacks notably.

Keywords: *obfuscation, hash function, genetic programming, avalanche effect, SMT solver.*

Введение

Обфускация — процесс запутывания кода программ с целью затруднения их анализа. Анализ исполняемого кода программ применяется как потенциальными нарушителями авторского права, так и антивирусными аналитиками, поэтому как методы обфускации, так и методы противодействия ей становятся темой многих научных работ [1, 2]. Одним из наиболее важных свойств обфускации является сохранение функциональности программы: для любых входных данных результат выполнения обфусцированной и оригинальной программы должны совпадать. От обфусцированной программы обычно требуют близкого к оригинальной программе размера и быстродействия, однако конкретные требования могут варьироваться в широких диапазонах в зависимости от задач [3].

Среди применяемых для обфускации подходов используются и основанные на криптографии, в том числе на криптографических хэш-функциях. Например, в [4] представлен подход, использующий свойство односторонности таких хэш-функций для запутывания условных переходов в программе.

Однако широко используемые хэш-функции, такие, как MD5, SHA-1 и SHA-2, зачастую могут быть распознаны в машинном коде по используемым в их реализации константам (например, `0xd76aa478` для MD5). Из-за этого эффективность их использования в обфускации подвергается критике [1]: аналитик сможет узнать функцию и сделать выводы об используемом методе обфускации, даже не разбирая её код. Современные инструменты обратной разработки программ также позволяют распознать популярные функции исключительно по машинному коду, без использования констант [5]. Существуют и методы обнаружения криптографических функций, основанные на динамическом анализе [6].

Проблемой применения распространённых хэш-функций в обфускации является не только тот факт, что атакующий получит представление о сути механизма обфускации, но и то, что переборные атаки на такие хэш-функции могут осуществляться при помощи готовых оптимизированных инструментов с очень высокой производительностью (порядка 10^{10} попыток в секунду) [7]. Поскольку многие константы в исходном коде программ являются 32-битными целочисленными значениями или строками, содержащими осмысленные слова, атака полным перебором в ряде случаев будет практически применимой.

Целью данной работы является решение упомянутых проблем путём автоматической генерации большого семейства ранее неизвестных случайных хэш-функций, которые можно использовать в процессе обфускации.

Использование таких хэш-функций призвано обеспечить следующие свойства:

1) Затруднение ручного анализа кода защищённых программ при помощи дизассемблеров и декомпиляторов. Такие функции не будут автоматически распознаны, даже сам факт того, что определённая часть кода является хэш-функцией, аналитику придётся устанавливать вручную.
2) Невозможность использования существующего программного обеспечения для атак полным перебором.
3) Возможность использования множества различных функций при обфускации разных мест программы, что кратно увеличит требуемые усилия по ручному анализу и созданию инструментов для переборных атак.

Среди существующих результатов по автоматической генерации хэш-функций наибольший интерес представляет работа [8]. Некоторые из описанных в ней подходов

нашли применение и в данном исследовании, хотя авторы [8] преследовали другую цель, в результате чего найденная ими хэш-функция gp-hash оказалась не слишком сложной для ручного анализа, а также тривиально обратимой. Продолжением [8] является работа [9], в которой описано создание блочного шифра Wheedham на основе сети Фейстеля при помощи нелинейных функций, подобных найденным в [8]. При помощи конструкции Миагучи — Пренеля из этого шифра, в свою очередь, была создана криптографическая хэш-функция MPW-512. Однако в силу использования готовых конструкций она не является сложной для обнаружения и анализа в исполняемом коде, что делает её малоподходящей для нужд обфускации. Кроме того, [8, 9] имеют целью поиск одной лучшей функции, а не семейства, что в значительной степени отличает их от данной работы.

1. Требования к хэш-функциям для обфускации

1.1. Механизм обфускации

Рассмотрим хэш-функции, применимые для механизма обфускации, описанного в [4]. Его идея состоит в следующем: значения констант в коде условий заменяются результатами выполнения хэш-функций, благодаря чему восстановление их первоначальных значений становится вычислительно сложной задачей. Пример использования подобного механизма обфускации некоторой функции проверки лицензионного ключа приведён в листингах 1 и 2.

```
1 int check_serial(char* serial) {
2     if (!strcmp(serial, "S0M3S3CR3TK3Y"))
3         return 1;
4     else
5         return 0;
6 }
```

Листинг 1. Функция до обфускации

```
1 int check_serial(char* serial) {
2     // 0x8cd28b8294f34457 == hash("S0M3S3CR3TK3Y")
3     if (hash(serial) == 0x8cd28b8294f34457)
4         return 1;
5     else
6         return 0;
7 }
```

Листинг 2. Функция после обфускации

1.2. Сравнение с криптографическими хэш-функциями

Хотя к криптографическим хэш-функциям предъявляется множество различных требований [10], для целей обфускации не все из них строго обязательны [4]. Рассмотрим основные требования к криптографическим хэш-функциям и их применимость в данной работе:

1) Стойкость к поиску первого прообраза — сложность нахождения для данного h такого m, что $f(m) = h$. Это требование очень важно для качественной обфускации, так как неразрывно связано с необратимостью хэш-функции. При невыполнении данного требования атакующий сможет восстановить исходные значения констант, нивелируя эффективность обфускации.

2) Стойкость к поиску второго прообраза — сложность нахождения для данного m_1 такого m_2, что $f(m_1) = f(m_2)$ и $m_1 \neq m_2$. В отличие от стойкости к поиску первого прообраза, данное требование не столь критично для обфускации. Существование таких коллизий неизбежно приведёт к нарушению свойства сохранения функциональности, но не повлияет негативно на стойкость программы к обратной разработке. Заметим, что в контексте сохранения функциональности наиболее интересны случайные коллизии, а не намеренно подобранные в результате криптографической атаки (если речь не идёт об участках кода, связанных с аутентификацией или другой обработкой недоверенного ввода, строго требующей отсутствия коллизий).
3) Стойкость к коллизиям — сложность нахождения таких m_1 и m_2, что $f(m_1) = f(m_2)$ и $m_1 \neq m_2$. Из данного требования следует стойкость к поиску второго прообраза, поэтому его также можно считать необязательным для специально выбранных m_1 и m_2. Однако именно это требование удобно использовать для проверки полученной функции при случайных m_1 и m_2. Если оно соблюдается, можно утверждать о сохранении функциональности с достаточной для стабильной работы обфусцированной программы точностью.

1.3. Специфические для обфускации требования

В то время как для криптографических хэш-функций наличие понятного и открытого описания является желательным согласно принципу Керхгоффса, в обфускации ценно запутывание любого рода, способное замедлить обратную разработку [11]. Соответственно для целей работы принцип Керхгоффса можно нарушить, не только скрыв от аналитика исходное сообщение, но и сделав саму функцию максимально непрозрачной. Для этого следует потребовать отсутствия закономерностей в её описании, например повторяющихся математических операций. Хорошим примером функции, нарушающей данное условие, является функция gp-hash, представленная в [8], в которой операция циклического сдвига вправо применяется к аргументу 18 раз подряд, что позволяет заменить эту цепочку операций одиночным сдвигом на 18 бит, сильно упростив код функции. Реализация этой функции и её упрощённый вариант представлены в листингах 3 и 4, где `ror` и `ror_18` — операция циклического сдвига вправо на 1 и 18 бит соответственно.

```
1 int gphash(int h, int m) {
2     return 0x6CF575C5 * ror(ror(ror(ror(ror(ror(ror(ror(ror
    (ror(ror(ror(ror(ror(ror(ror(ror(ror(0x6CF575C5 * (h + m
    ))))))))))))))))));
3 }
```

Листинг 3. Реализация исходного варианта функции gp-hash

```
1 int gphash(int h, int m) {
2     return 0x6CF575C5 * ror_18(0x6CF575C5 * (h + m));
3 }
```

Листинг 4. Упрощенная реализация функции gp-hash

Выполнение хэш-функций подразумевается в каждом из обфусцируемых условий, соответственно необходимо обеспечить высокую скорость их работы. Среди способов добиться этого можно отметить следующие:

1) Отсутствие раундовой структуры. Так как с ростом числа раундов пропорционально растёт количество исполняемых инструкций, не имеет смысла использовать множество раундов, если функция способна обеспечить требуемые свойства за один. Наряду с этим наличие повторяющихся раундов нарушает требование максимальной запутанности кода.
2) Использование в реализации операций, которые можно скомпилировать в одну машинную инструкцию целевой процессорной архитектуры. Некоторые операции, имеющие достаточно простую математическую запись (например, деление с остатком), могут требовать для реализации выполнения множества машинных инструкций, что делает их менее привлекательными для использования.

1.4. Итоговые требования к хэш-функциям для обфускации

Объединив требования к хэш-функциям, применимым в обфускации, поставленные в п. 1.2 и 1.3, получим следующий список:

1) стойкость к поиску первого прообраза — необратимость хэш-функции;
2) стойкость к случайным коллизиям — малая вероятность совпадения значений хэш-функции для случайных сообщений;
3) запутанность описания функции — отсутствие в описании хэш-функции закономерностей, позволяющих упростить её исследование;
4) высокая скорость работы — скорость работы, сравнимая с распространёнными некриптографическими хэш-функциями.

1.5. Количественные оценки

Одним из важных свойств криптографических хэш-функций является лавинный эффект — значительное изменение результата при изменении малого количества бит в исходном сообщении. Это свойство связано с понятием диффузии Шеннона и препятствует установлению связи между входными и выходными значениями, а значит, может оказаться полезным при обеспечении стойкости к поиску первого прообраза.

Математически лавинный эффект определяется следующим образом: при изменении одного бита входного сообщения в выходном значении должна измениться в среднем половина бит:

$$\forall x, y\,(\mathrm{d}(x, y) = 1 \;\Rightarrow \mathrm{d}(F(x), F(y)) \approx N/2).$$

Здесь $\mathrm{d}(x, y)$ — расстояние Хэмминга между x и y; $F(x)$ — функция, обладающая лавинным эффектом; N — размер её результата в битах.

Существует и более строгое условие — строгий лавинный критерий [12], при соблюдении которого автоматически соблюдается и условие лавинного эффекта. Он звучит так: при изменении одного бита входного сообщения каждый бит выходного сообщения должен измениться с вероятностью $1/2$; его можно представить в другом виде: общее количество изменившихся бит должно соответствовать биномиальному распределению с параметрами $(N, 1/2)$:

$$\forall x, y\,(\mathrm{d}(x, y) = 1 \;\Rightarrow\; \mathrm{d}(F(x), F(y)) \sim B(N, 1/2)). \tag{1}$$

Для данной работы выбрано именно это условие как более строгое. Соответствие полученного распределения образцовому оценивалось при помощи критерия согласия Пирсона.

Отсутствие закономерностей в описании хэш-функций, в свою очередь, связано с понятием колмогоровской сложности — минимально возможным алгоритмом, гене-

рирующим описание хэш-функции. Так как колмогоровская сложность является невычислимой, в качестве её замены можно использовать алгоритмы сжатия общего назначения, например Deflate, как предложено в [13]. Чем больший размер имеет функция в сжатом виде, тем больше оценка минимальной длины её описания, а значит, тем сложнее она для упрощения и анализа.

2. Алгоритм поиска хэш-функций для обфускации

2.1. Структура генерируемых хэш-функций

В качестве структуры хэш-функций выбран упрощённый вариант структуры Меркла — Дамгора: ввиду того, что стойкость к намеренному поиску коллизий не является обязательной для целей обфускации, этап с дополнением сообщения его длиной можно пропустить. Тогда функция будет выглядеть следующим образом:

$$H_0 = IV, \quad H_i = F(H_{i-1}, m_i), \quad i = 1, 2, \ldots \tag{2}$$

Здесь m_i — части исходного сообщения M одинакового размера; $F(h, m)$ — функция сжатия; IV — некоторое начальное значение функции. Значение H_n для сообщения из n частей является результатом хэш-функции от всего сообщения M. В рамках данной работы длина m_i и H_i в битах считается равной.

Таким образом, задача поиска хэш-функций сводится к задаче поиска функций сжатия $F(h, m)$.

2.2. Генетическое программирование

Цель работы состоит в автоматической генерации функций, поэтому используемый алгоритм должен быть способен осуществлять поиск в пространстве всех возможных функций. Одним из немногих подходов, пригодных для решения задач подобного рода, является подход генетического программирования [14]. Обычно алгоритмы генетического программирования оперируют функциями, представленными в виде деревьев, где внутренние узлы являются операциями, а листья — операндами (или терминалами). На каждом шаге поиска решения к некоторой популяции функций применяются генетические операторы мутации и скрещивания, после чего в соответствии с требованиями, сформулированными в виде оптимизируемой функции приспособленности, формируется следующее поколение. После некоторого количества итераций поиск завершается и лучшая функция популяции выбирается решением задачи.

В качестве программной реализации выбрана библиотека DEAP для языка Python, содержащая готовые реализации описанного подхода и требующая только задания требований конкретной задачи [15].

2.3. Набор терминалов и операций

В соответствии со структурой хэш-функций (2) терминалами являются значение функции h, полученное на предыдущем этапе, и m — текущий блок хэшируемого сообщения. В качестве операций выбраны операции, которые быстро выполняются на современных процессорах и являются достаточно простыми, с целью соблюдения предъявленного в п. 1.3 требования к производительности.

В результате выбраны следующие операции:

1) or(x, y), and(x, y), xor(x, y) — побитовые операции ИЛИ, И и исключающее ИЛИ;
2) add(x, y), mul(x, y) — беззнаковые сложение и умножение;
3) not(x) — побитовое отрицание;
4) ror_1(x), ror_half(x) — операции циклического сдвига вправо на 1 бит и на половину размера переменной: хотя вторую операцию можно выразить через

первую, многократный повтор операции противоречит условию максимальной запутанности из п. 1.3;

5) dynror(x, y) — циклический сдвиг x вправо на y бит; хотя эта операция и выглядит менее тривиально, чем предыдущие, на архитектуре x86, широко используемой на существующих компьютерах, она кодируется одной инструкцией `ROR r/m8, CL`.

2.4. Функция приспособленности

Искомая функция $F(h, m)$ применяется к двум аргументам, поэтому условие (1) уточнено следующим образом: потребуем соответствия функции строгому лавинному критерию для обеих переменных одновременно (как если бы h и m были одним аргументом x двойного размера для функции $F(x)$).

Для количественного измерения соответствия условию (1) последовательно осуществляются следующие шаги:

1) генерация пары h_0, m_0 при помощи генератора псевдослучайных чисел;
2) подсчёт значения $F_0 = F(h_0, m_0)$;
3) изменение одного случайного бита в одной из переменных h_0 или m_0, новая пара значений после этой операции обозначается как h_1, m_1;
4) подсчёт значения $F_1 = F(h_1, m_1)$;
5) подсчёт и запоминание числа различающихся в F_0 и F_1 бит $\mathrm{d}(F_0, F_1)$.

После проведения достаточного числа итераций полученная выборка $\mathrm{d}(F_0, F_1)$ проверяется на соответствие распределению $B(N, 1/2)$ при помощи критерия согласия Пирсона: получаем значение χ^2:

$$\chi^2 = \sum_{i=0}^{N} \frac{(O_i - E_i)^2}{E_i}.$$

Здесь O_i — наблюдаемое количество итераций, где изменилось i бит, а E_i — ожидаемое количество для распределения $B(N, 1/2)$. В данной работе использован генератор случайных чисел MT19937 (вихрь Мерсенна), а число итераций, обеспечивающее достаточную точность оценки лавинного эффекта, было экспериментально установлено равным 1024 (оно должно быть значительно больше числа бит функции).

Для оценки запутанности кода полученной функции необходимо подвергнуть его сжатию. Для этого сначала необходимо преобразовать дерево функции $F(h, m)$ в линейное представление, что можно сделать средствами библиотеки DEAP. Затем каждый элемент (операция или терминал) заменяется численным представлением — индексом данной операции или терминала, в результате чего получается список чисел. Так как общее количество операций и терминалов не превышает 2^8, каждый элемент занимает один байт, а значит, список можно тривиально преобразовать в байтовую строку. Полученная строка подвергается сжатию при помощи алгоритма Deflate, а длина сжатых данных является характеристикой S функции $F(h, m)$.

Для обеспечения выполнения обоих условий из п. 1.5 значение χ^2 должно быть подвергнуто минимизации, а S — максимизации. Поэтому возникает вопрос объединения данных характеристик в одну функцию, оптимизируемую при помощи генетического программирования. Поскольку условие строгого лавинного критерия менее тривиально, решено отдать ему приоритет и только при достижении им некоторого целевого значения χ^2_{tgt} обеспечивать отсутствие в функции закономерностей.

Так как S намного меньше χ^2 для условий данной работы, итоговая функция приспособленности, подвергающаяся минимизации, имеет следующий вид:

$$F_{\text{opt}} = \max(\chi^2 - \chi^2_{\text{tgt}}, 0) - S.$$

В качестве значения χ^2_{tgt} можно взять характеристику χ^2 любой современной криптографической хэш-функции, стойкость которой к атакам поиска первого прообраза не вызывает опасений. В данной работе использована функция SHAKE256 из семейства SHA-3, удобная тем, что позволяет получить результат произвольной длины [16].

2.5. Описание алгоритма

Приведём разработанный метод в алгоритме 1. Здесь *measureSAC* и *measureComplexity* — функции оценки лавинного эффекта и запутанности кода (см. п. 2.4); *createPopulation* и *generateOffspring* — функции создания популяции и генерации потомков, предоставленные библиотекой генетического программирования DEAP. Функция *check* служит для дополнительной проверки функций на применимость в обфускации, принцип её работы описан далее в п. 4.4.

Алгоритм 1. Алгоритм поиска хэш-функций для обфускации

params := параметры генетического программирования;
$score_{\text{tgt}}$:= целевое значение функции;
$\chi^2_{\text{tgt}} := measureSAC(\text{SHAKE256})$;
population := *createPopulation*(*params*).
Повторять
  population := *generateOffspring*(*population*, *params*).
  Для всех $ind \in population$
    $ind.score := \max(measureSAC(ind) - \chi^2_{\text{tgt}}, 0) - measureComplexity(ind)$.
  $score_{\min} := \min\limits_{ind \in population}(ind.score)$.
Пока $score_{\min} \geqslant score_{\text{tgt}}$
$ind_{\text{best}} := \arg\min\limits_{ind \in population}(ind.score)$.
Если $check(ind_{\text{best}})$, **то**
  Вывести ind_{best}.

3. Проверка применимости полученных хэш-функций в обфускации

3.1. Метод автоматического анализа на обратимость

В программном обеспечении для осуществления деобфускации активно применяются подходы символьного исполнения и SMT-решатели [17, 18]. Использующие их инструменты angr и S2E способны автоматически генерировать входные данные, необходимые для выполнения различных ветвей условий программы, а значит, эти инструменты могут быть использованы и для атак на механизм обфускации, описанный в п. 1.1. По этой причине крайне важна проверка полученных функций на устойчивость к SMT-решателям. Проверка хэш-функций осуществлялась при помощи следующих инструментов:

1) Microsoft Z3 (версия 4.8.7) — наиболее важный решатель для целей данной работы, потому что именно он используется в инструменте автоматического анализа кода angr по умолчанию [17, 19];

2) CVC4 (версия 1.8) — решатель, который также поддерживается инструментом angr [20];
3) Yices2 (версия 2.6.2) — победитель трека QF_BV (application) в соревнованиях SMT-COMP 2018 [21];
4) Boolector (версия 3.2.1) — победитель трека QF_ABV в соревнованиях SMT-COMP 2018 и SMT-COMP 2019 [22].

Основой предлагаемого метода проверки хэш-функций является сравнение времени поиска m для определённых $H = F(h, m)$ и h при помощи SMT-решателей и полного перебора: если первое из времён больше (или решение вовсе не найдено), можно говорить об устойчивости функции к автоматическому анализу, так как SMT-решатели не способны ускорить процесс поиска.

Соответственно для проверки функции на обратимость достаточно осуществить следующие операции:

1) найти T_{bf} — худшее возможное время атаки полным перебором. Это значение соответствует суммарному времени подсчёта $F(h, m)$ для всех возможных m при фиксированном h;
2) найти T_{smt} — время, которое лучший из SMT-решателей затратит на решение задачи нахождения m по значению $H = F(h, m)$ для известного h;
3) сравнить T_{smt} и T_{bf} и сделать вывод об устойчивости функции, если $T_{\text{smt}} > T_{\text{bf}}$.

Для реализации программы перебора использовался язык Си в сочетании с компилятором GCC версии 10.1 с уровнем оптимизации O3. Перебор осуществлялся в однопоточном режиме.

Время анализа при помощи SMT-решателей измерялось путём их параллельного запуска на разных ядрах процессора. Время, через которое первый из SMT-решателей справлялся с задачей, являлось искомым T_{smt}. Длительность работы SMT-решателей сложно спрогнозировать, а время экспериментов не может быть неограниченным, поэтому максимальное время их работы ограничивалось некоторым T_{smtlim}, по достижении которого задача объявлялась неразрешимой за допустимое время.

3.2. Метод анализа устойчивости к случайным коллизиям

Традиционным способом анализа устойчивости к случайным коллизиям является использование парадокса дней рождения: при хэшировании случайных значений идеальной хэш-функцией длины N бит через $\approx 2^{N/2}$ операций будет обнаружена коллизия. Данный способ исследован Д. Кнутом в [23], в результате чего установлено выражение для среднего количества операций до первой коллизии: $C_{\text{ideal}} = 1 + Q(2^N)$. Функция $Q(x)$, в свою очередь, изучена в работе [24], где для неё предложено следующее приближение:

$$Q(x) \approx \sqrt{\frac{\pi x}{2}} - \frac{1}{3} + \frac{1}{12}\sqrt{\frac{\pi}{2x}} - \frac{4}{135x} + \dots$$

Именно это приближение использовано в данной работе.

Для тестируемых функций среднее количество хэшированій до коллизии C_{bday} может отличаться в меньшую сторону от C_{ideal} в силу их неидеальности, например в случае, если реальная область значений хэш-функции меньше чем $\{0, \dots, 2^N - 1\}$. Оценить количество потерянных бит области значений N_{lost} можно при помощи выражения $C_{\text{bday}} = 1 + Q(2^{N - N_{\text{lost}}})$

Предложим следующее условие устойчивости к случайным коллизиям: если для хэш-функции N_{lost} мало в сравнении с N, то она является достаточно устойчивой

к случайным коллизиям. Метод анализа устойчивости к случайным коллизиям состоит из следующих шагов:

1) поиск C_{bday} — количества хэшированний до первой коллизии. Для этого необходимо генерировать значения H хэш-функции от случайных сообщений до первого совпадения с одним из ранее полученных значений, число выполненных итераций будет искомым значением. Здесь $H = F(F(IV, r_1), r_2)$ в соответствии со структурой хэш-функций (2), где r_1 и r_2 — случайные значения; IV — некоторое постоянное значение. Для увеличения точности следует провести несколько экспериментов, усредняя C_{bday};
2) подсчёт N_{lost} с использованием выражения $C_{\text{bday}} = 1 + Q(2^{N-N_{\text{lost}}})$. Осуществляется численно, так как $Q(x)$ в любом случае является приближением;
3) сравнение N_{lost} и N: если $N_{\text{lost}} \ll N$, то функция устойчива к случайным коллизиям.

Заметим, что в данном методе хэшируется сообщение из двух блоков (r_1 и r_2), а не из одного. Это связано с тем, что размер аргумента m совпадает с размером функции $F(h, m)$, а значит, вероятность повтора случайных m достаточно высока, чтобы повлиять на итоговое C_{bday}: она столь же высока, как и вероятность равенства выходных значений идеальной хэш-функции. Нас же интересуют только коллизии, где $f(m_1) = f(m_2)$, но $m_1 \neq m_2$, поэтому необходимо хэшировать сообщения длины, как минимум вдвое большей, чем длина значения функции: тогда вероятность равенства исходных сообщений становится пренебрежимой.

4. Экспериментальная проверка хэш-функций

4.1. Условия экспериментов

Для того чтобы полный перебор, требуемый для оценки устойчивости к автоматическому анализу, был практически осуществимым, в качестве длины аргументов и результата функции решено выбрать 32 бита. Хотя 32-битные хэш-функции, очевидно, мало подходят для защиты от подбора первоначальных значений, полученные с их использованием результаты могут быть распространены на функции большей разрядности.

Время выполнения итоговой функции определяется общим количеством операций, поэтому решено не вносить ограничений на высоту дерева функции в процессе генетического программирования, а использовать ограничение на общее число узлов. С целью обеспечения производительности, близкой к существующим некриптографическим хэш-функциям, это значение выбрано равным 32. Тогда для 32-битных функций для хэширования одного байта сообщения потребуется не более 8 инструкций (учитывая использованный в п. 2.3 набор операций), для сравнения — широко распространённой функции FNV требуется не меньше трёх, без учёта более частых условных переходов [25].

В отличие от традиционного использования генетического программирования для задач оптимизации, было решено не ограничивать общее число поколений, заменив его целевым значением функции приспособленности, равным -30, что соответствует достижению лавинных характеристик, сравнимых с характеристиками криптографических хэш-функций (в данном случае SHAKE256) и высокой степени запутанности (здесь целевое S зависит от разрешенного максимального числа узлов).

Итоговые параметры, использованные для генетического программирования, приведены в табл. 1. Для экспериментов применялся компьютер с ОС Ubuntu Linux 18.04, процессором Intel Core i7-5820K (4 ГГц) и 32 ГБ ОЗУ.

Таблица 1

Параметры генетического программирования

Параметр	**Значение**
Размер популяции	200
Вероятность скрещивания	0,9
Вероятность мутации	0,1
Максимальное число узлов	32
Набор операций	or, and, xor, add, mul, not, ror_1, ror_half, dynror
Набор терминалов	h, m
Функция приспособленности	$\max(\chi^2 - \chi^2_{\text{tgt}}, 0) - S$
Целевое значение функции	-30
Отбор	Турнирный отбор размера 3
Эволюционная стратегия	(1+1)-стратегия

4.2. Результаты автоматического анализа

При помощи генетического программирования было сгенерировано 256 функций, подвергнутых проверке. Среднее наихудшее время полного перебора для сгенерированных функций оказалось равным $T_{\text{bf}} = 6{,}65\,\text{с}$. Так как SMT-решатели могут выполнять исследуемую функцию медленнее, чем оптимизированная программа на языке C, было решено взять максимальное время их работы равным $T_{\text{smtlim}} = 180\,\text{с}$. Наибольший интерес с практической точки зрения имеют времена решений меньшие, чем T_{bf}, поэтому значение T_{smtlim} может быть выбрано приблизительно.

С учётом данных значений осуществлена проверка при помощи SMT-решателей, результаты которой приведены в табл. 2 и на рис. 1. Можно видеть, что, несмотря на ограниченное пространство поиска, в 159 случаях SMT-решатели не смогли найти ответ за время T_{smtlim}. Количество функций, в которых время решения T_{smt} оказалось меньше T_{bf}, составило 53. Для этих функций можно считать, что алгоритм поиска хэш-функций совершенно не справился с задачей.

Таблица 2

Результаты проверки 32-битных функций при помощи SMT-решателей

Категория	**Количество**
Функции, для которых $T_{\text{smt}} < T_{\text{bf}}$	53
Функции, для которых $T_{\text{smt}} < T_{\text{smtlim}}$	97
Функции, для которых $T_{\text{smt}} \geqslant T_{\text{smtlim}}$	159

Таким образом, в 62 % случаев алгоритм смог сгенерировать 32-битные функции, которые не поддаются анализу при помощи SMT-решателей за допустимое время, причём в 17 % оставшихся случаев время решения оказалось больше, чем худшее время полного перебора. Проанализировав гистограмму на рис. 1, заметим, что наибольшее количество решений происходит за небольшое время, причём за пределами 120 с количество решений сократилось до нуля. Если совместить поиск функций с их быстрой проверкой в течение времени T_{bf}, можно повысить долю неразрешимых за время T_{smtlim} функций до 78 %.

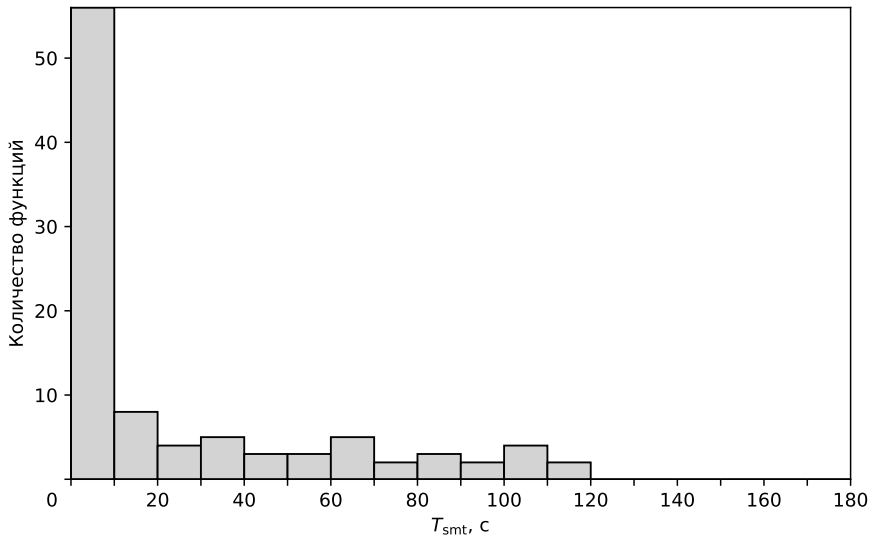

Рис. 1. Гистограмма количества 32-битных функций, для которых SMT-решатели нашли решение за указанное время, без учёта случаев $T_{\text{smt}} > T_{\text{smtlim}}$

4.3. Устойчивость к случайным коллизиям

Поскольку устойчивость к случайным коллизиям является второстепенным свойством после устойчивости к автоматическому анализу, проверке подвергались только те функции, которые не были обращены при помощи SMT-решателей за время меньшее T_{bf}. За счёт небольшой области значений функций описанная в п. 3.2 проверка может быть выполнена с достаточной для экспериментов производительностью. Для каждой функции измерялось среднее значение C_{bday} за 512 попыток, после чего для него рассчитывалось приблизительное значение N_{lost}.

Для 203 проверенных функций среднее значение C_{bday} составило 78080, наименьшее равно 63641. Отсюда среднее число потерянных бит $N_{\text{lost}} = 0{,}15$ и наихудшее — 0,74. Таким образом, в наихудшем случае хэш-функция имеет не больше коллизий, чем идеальная хэш-функция с длиной значения 31 бит.

4.4. Функции большей разрядности

Хотя многие 32-битные функции оказались устойчивы к анализу при помощи SMT-решателей, они уязвимы к переборным атакам (которые атакущий может провести на обычном процессоре в силу малого пространства поиска), вследствие чего было решено проанализировать функции с длинами значений, практически применимыми в обфускации: 64 и 128 бит. Для таких функций использование полного перебора затруднительно, поэтому в отличие от сценария, описанного в п. 3.1, в этих экспериментах имеет значение не сравнение T_{bf} и T_{smt}, а принципиальная достижимость решения за какое бы то ни было разумное время.

Из соображений продолжительности эксперимента значение T_{smtlim} оставлено прежним (180 с), для каждой размерности сгенерировано по 256 функций. Результаты проверки для 64- и 128-битных функций представлены на рис. 2 и 3 соответственно. Явление, наблюдавшееся для 32-битных функций, здесь ещё более выражено: для 64-битных функций максимальное время решения, меньшее T_{smtlim}, составило 17 с, а для 128-битных — 8 с. Таким образом, можно отсеивать крайне слабые функции, запуская сразу после их генерации дополнительную проверку, заключающуюся в анализе при помощи SMT-решателей в течение 17 и 8 с соответственно: если решение в течение этого времени найдено, функцию можно считать слабой. Этот способ позволяет повы-

сить долю неразрешимых за время T_{smtlim} функций до 100 % и может быть реализован в функции *check* алгоритма 1.

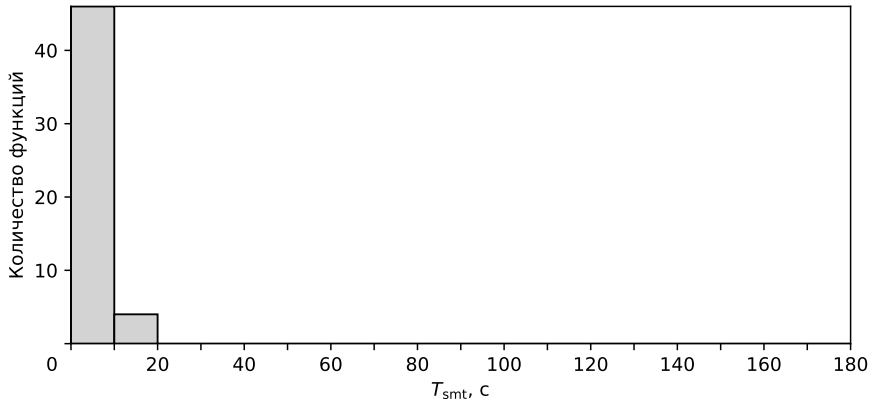

Рис. 2. Гистограмма количества 64-битных функций, для которых SMT-решатели нашли решение за указанное время, без учёта случаев $T_{\text{smt}} > T_{\text{smtlim}}$

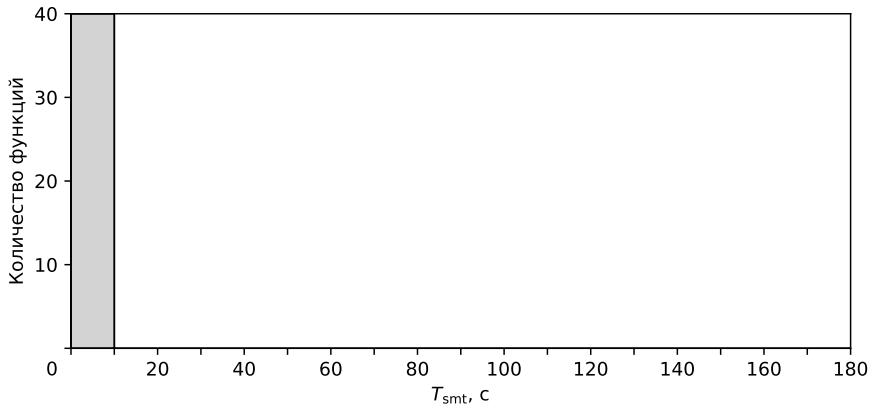

Рис. 3. Гистограмма количества 128-битных функций, для которых SMT-решатели нашли решение за указанное время, без учёта случаев $T_{\text{smt}} > T_{\text{smtlim}}$

Ввиду того, что может возникнуть вопрос адекватности выбора T_{smtlim}, было решено проверить несколько случайных функций, устойчивых к анализу за время T_{smtlim}, в течение намного большего времени. Для этого было выбрано по 16 соответствующих 64- и 128-битных функций, каждая из них подверглась анализу при помощи SMT-решателей в течение 4 ч. В результате эксперимента решение не было найдено ни для одной из функций, что позволяет утверждать, что данные функции являются устойчивыми к атакам с практической точки зрения.

Заключение

Сформулированы требования к хэш-функциям, применимым для использования в обфускации, и показаны различия между ними и требованиями, предъявляемыми к криптографическим хэш-функциям. Данные требования выражены в виде количественных оценок. Предложен метод поиска хэш-функций для обфускации с использованием генетического программирования и этих оценок. Найденные функции проверены на устойчивость к атаке поиска первого прообраза при помощи SMT-решателей и на устойчивость к случайным коллизиям.

Полученные результаты позволяют утверждать, что генерируемые предложенным алгоритмом функции применимы для защиты от инструментов автоматического анализа кода, а также от обратной разработки в целом. Хотя сам по себе алгоритм не всегда генерирует устойчивые к автоматическому анализу функции, в сочетании со способом исключения слабых функций он даёт хорошие результаты. С небольшим общим временем генерации функций алгоритм генерирует 64- и 128-битные хэш-функции, устойчивые к анализу в течение достаточно длительного времени во всех проверенных случаях.

ЛИТЕРАТУРА

1. *Banescu S., Collberg C. S., Ganesh V., et al.* Code obfuscation against symbolic execution attacks // Proc. 32nd Ann. Conf. Computer Security Appl. 2016. P. 189–200.
2. *Udupa S., Debray S., and Madou M.* Deobfuscation: Reverse Engineering Obfuscated Code. 12th Working Conf. WCRE'05. 2005. 10 p.
3. *Junod P., Rinaldini J., Wehrli J., and Michielin J.* Obfuscator-LLVM — software protection for the masses // IEEE/ACM 1st Intern. Workshop Software Protection. 2015. P. 3–9.
4. *Sharif M., Lanzi A., Giffin J., and Lee W.* Impeding malware analysis using conditional code obfuscation // Proc. NDSS. San Diego, California, USA, 2008.
5. `https://www.hex-rays.com/products/ida/lumina/` — IDA: Lumina server. 2020.
6. *Xu D., Ming J., and Wu D.* Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping // IEEE Symp. Security Privacy (SP). Los Alamitos, CA, USA, 2017. P. 921–937.
7. *Qiu W., Gong Z., Guo Y., et al.* GPU-based high performance password recovery technique for hash functions // J. Inform. Sci. Eng. 2016. V. 32. No. 1. P. 97–112.
8. *Estebanez C., Hernandez-Castro J., Ribagorda A., and Isasi P.* Finding state-of-the-art non-cryptographic hashes with genetic programming // LNCS. 2006. V. 4193. P. 818–827.
9. *Tapiador J., Hernandez-Castro J., Peris-Lopez P., and Ribagorda A.* Automated design of cryptographic hash schemes by evolving highly-nonlinear functions // J. Inform. Sci. Eng. 2008. V. 24. No. 5. P. 1485–1504.
10. *Menezes A., van Oorschot P., and Vanstone S.* Handbook of Applied Cryptography. CRC Press, 1996. 661 p.
11. *Collberg C. S. and Thomborson C.* Watermarking, tamper-proofing, and obfuscation — tools for software protection // IEEE Trans. Software Eng. 2002. V. 28. No. 8. P. 735–746.
12. *Webster A. F. and Tavares S. E.* On the design of S-boxes // LNCS. 1985. V. 218. P. 523–534.
13. *Cilibrasi R. and Vitanyi P. M. B.* Clustering by compression // IEEE Trans. Inform. Theory. 2005. V. 51. No. 4. P. 1523–1545.
14. *Koza J. R.* Genetic programming as a means for programming computers by natural selection // Stat. Comput. 1994. V. 4. No. 2. P. 87–112.
15. *Fortin F. A., De Rainville F. M., Gardner M. A. G., et al.* DEAP: Evolutionary algorithms made easy // J. Machine Learning Res. 2012. V. 13. No. 1. P. 2171–2175.
16. *Dworkin M. J.* SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. National Institute of Standards and Technology, 2015.
17. *Shoshitaishvili Y., Wang R., Salls C., et al.* SOK: (State of) The art of war: Offensive techniques in binary analysis // IEEE Symp. Security Privacy (SP). 2016. P. 138–157.
18. *Chipounov V., Kuznetsov V., and Candea G.* S2E: A platform for in-vivo multi-path analysis of software systems // ACM SIGPLAN Notices. 2011. V. 46. No. 3. P. 265–278.
19. *De Moura L. and Bjorner N.* Z3: An efficient SMT solver // LNCS. 2008. V. 4963. P. 337–340.

20. *Barrett C., Conway C. L., Deters M., et al.* CVC4 // Intern. Conf. Comput. Aided Verification. 2011. P. 171–177.
21. *Dutertre B.* Yices 2.2 // Intern. Conf. Comput. Aided Verification. 2014. P. 737–744.
22. *Brummayer R. and Biere A.* Boolector: An efficient SMT solver for bit-vectors and arrays // LNCS. 2009. V. 5505. P. 174–177.
23. *Knuth D. E.* The Art of Computer Programming. V. 3: Sorting and Searching. 2nd Ed. Addison-Wesley Professional, 1998. 800 p.
24. *Flajolet P., Grabner P. J., Kirschenhofer P., and Prodinger H.* On Ramanujan's Q-function // J. Computat. Appl. Math. 1995. V. 58. No. 1. P. 103–116.
25. `http://www.isthe.com/chongo/tech/comp/fnv/` — FNV Hash. 1991.

REFERENCES

1. *Banescu S., Collberg C. S., Ganesh V., et al.* Code obfuscation against symbolic execution attacks. Proc. 32nd Ann. Conf. Computer Security Appl., 2016, pp. 189–200.
2. *Udupa S., Debray S., and Madou M.* Deobfuscation: Reverse Engineering Obfuscated Code. 12th Working Conf. WCRE'05, 2005. 10 p.
3. *Junod P., Rinaldini J., Wehrli J., and Michielin J.* Obfuscator-LLVM — software protection for the masses. IEEE/ACM 1st Intern. Workshop Software Protection, 2015, pp. 3–9.
4. *Sharif M., Lanzi A., Giffin J., and Lee W.* Impeding malware analysis using conditional code obfuscation. Proc. NDSS. San Diego, California, USA, 2008.
5. `https://www.hex-rays.com/products/ida/lumina/` — IDA: Lumina server. 2020.
6. *Xu D., Ming J., and Wu D.* Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping. IEEE Symp. Security Privacy (SP), Los Alamitos, CA, USA, 2017, pp. 921–937.
7. *Qiu W., Gong Z., Guo Y., et al.* GPU-based high performance password recovery technique for hash functions. J. Inform. Sci. Eng., 2016, vol. 32, no. 1, pp. 97–112.
8. *Estebanez C., Hernandez-Castro J., Ribagorda A., and Isasi P.* Finding state-of-the-art non-cryptographic hashes with genetic programming. LNCS, 2006, vol. 4193, pp. 818–827.
9. *Tapiador J., Hernandez-Castro J., Peris-Lopez P., and Ribagorda A.* Automated design of cryptographic hash schemes by evolving highly-nonlinear functions. J. Inform. Sci. Eng., 2008, vol. 24, no. 5, pp. 1485–1504.
10. *Menezes A., van Oorschot P., and Vanstone S.* Handbook of Applied Cryptography. CRC Press, 1996. 661 p.
11. *Collberg C. S. and Thomborson C.* Watermarking, tamper-proofing, and obfuscation — tools for software protection. IEEE Trans. Software Eng., 2002, vol. 28, no. 8, pp. 735–746.
12. *Webster A. F. and Tavares S. E.* On the design of S-boxes. LNCS, 1985, vol. 218, pp. 523–534.
13. *Cilibrasi R. and Vitanyi P. M. B.* Clustering by compression. IEEE Trans. Inform. Theory, 2005, vol. 51, no. 4, pp. 1523–1545.
14. *Koza J. R.* Genetic programming as a means for programming computers by natural selection. Stat. Comput., 1994, vol. 4, no. 2, pp. 87–112.
15. *Fortin F. A., De Rainville F. M., Gardner M. A. G., et al.* DEAP: Evolutionary algorithms made easy. J. Machine Learning Res., 2012, vol. 13, no. 1, pp. 2171–2175.
16. *Dworkin M. J.* SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. National Institute of Standards and Technology, 2015.
17. *Shoshitaishvili Y., Wang R., Salls C., et al.* SOK: (State of) The art of war: Offensive techniques in binary analysis. IEEE Symp. Security Privacy (SP), 2016, pp. 138–157.
18. *Chipounov V., Kuznetsov V., and Candea G.* S2E: A platform for in-vivo multi-path analysis of software systems. ACM SIGPLAN Notices, 2011, vol. 46, no. 3, pp. 265–278.

19. *De Moura L. and Bjorner N.* Z3: An efficient SMT solver. LNCS, 2008, vol. 4963, pp. 337–340.
20. *Barrett C., Conway C. L., Deters M., et al.* CVC4. Intern. Conf. Comput. Aided Verification, 2011, pp. 171–177.
21. *Dutertre B.* Yices 2.2. Intern. Conf. Comput. Aided Verification, 2014, pp. 737–744.
22. *Brummayer R. and Biere A.* Boolector: An efficient SMT solver for bit-vectors and arrays. LNCS, 2009, vol. 5505, pp. 174–177.
23. *Knuth D. E.* The Art of Computer Programming. vol. 3: Sorting and Searching. 2nd Ed. Addison-Wesley Professional, 1998. 800 p.
24. *Flajolet P., Grabner P. J., Kirschenhofer P., and Prodinger H.* On Ramanujan's Q-function. J. Computat. Appl. Math., 1995, vol. 58, no. 1, pp. 103–116.
25. `http://www.isthe.com/chongo/tech/comp/fnv/` — FNV Hash. 1991.