Для каждой новой схемы необходима генерация новой пары ключей доказательства и верификации. Их генерация выполняется вне блокчейна, поскольку в алгоритме генерации используется параметр безопасности, зная который, можно создавать некорректные доказательства, которые будут приняты верификатором как корректные.

Таким образом, создана система, которая позволяет разработчикам реализовывать произвольные алгоритмы на основе добавленных базовых схем непосредственно в коде смарт-контрактов. Метод позволяет сократить размер кода смарт-контрактов и, кроме того, оказывается более вычислительно эффективным.

ЛИТЕРАТУРА

1. *Ben-Sasson E., Chiesa A., Genkin D., et al.* SNARKs for C: Verifying program executions succinctly and in zero knowledge // CRYPTO'2013. LNCS. 2013. V. 8043. P. 90–108.
2. *Кондырев Д. О.* Разработка метода сокрытия приватных данных для системы тендеров на основе технологии блокчейн // Прикладная дискретная математика. 2020. № 48. С. 63–81.
3. *Eberhardt J. and Tai S.* ZoKrates — scalable privacy-preserving off-chain computations // IEEE Intern. Conf. Blockchain. Halifax, Canada, 2018. P. 1084–1091.
4. `https://github.com/scipr-lab/libsnark` — libsnark: a C++ library for zkSNARK proofs.

УДК 004.021 DOI 10.17223/2226308X/14/29

ДЕОБФУСКАЦИЯ CONTROL FLOW FLATTENING СРЕДСТВАМИ СИМВОЛЬНОГО ИСПОЛНЕНИЯ

В. В. Лебедев

Метод обфускации Control Flow Flattening заменяет в коде программы все условные и безусловные переходы на переход в специальный управляющий блок — диспетчер, который определяет, куда на самом деле будет передано управление в программе. Это делает невозможным исследователю быстро определить, в какой последовательности исполняется код в программе. Предлагается алгоритм восстановления исходной логики программ, обфусцированных этим методом. В основе алгоритма лежит символьное исполнение.

Ключевые слова: *реверс-инжиниринг, символьное исполнение, обфускация, control flow flattening.*

Введение

Control Flow Flattening [1] — техника обфускации, с помощью которой скрываются ветвления в коде. Вместо последовательного выполнения базовых блоков (линейных участков кода) каждому из них присваивается определённый номер. Вместо прямого перехода на следующий блок номер этого блока записывается в управляющий регистр, затем делается переход в специальный управляющий блок — диспетчер, который, исходя из номера блока, делает на него переход (рис. 1). В коде на языке Си это выглядит как switch внутри цикла while (рис. 2).

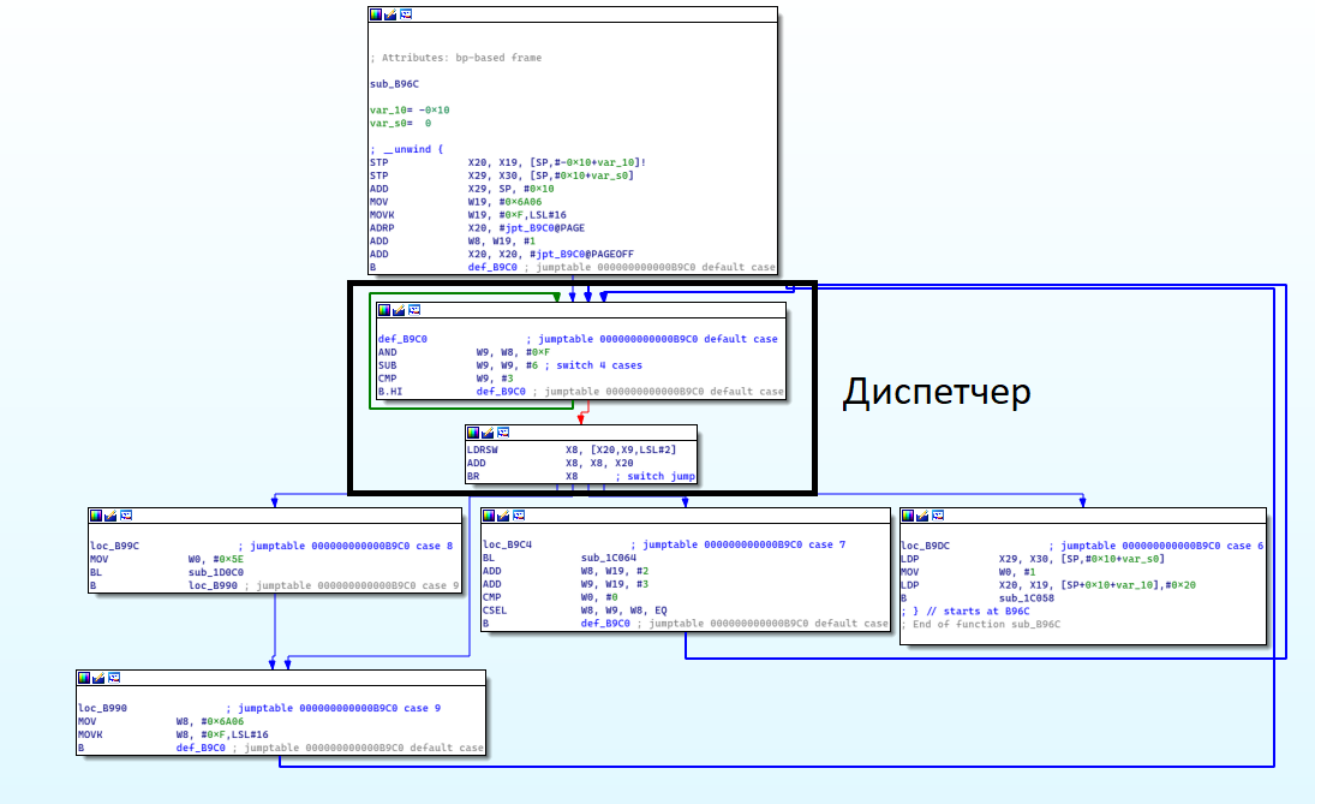

Рис. 1. Граф потока исполнения обфусцированной функции

```
__int64 sub_B96C()
{
  char v0; // w8

  v0 = 7;
  while ( 1 )
  {
    switch ( v0 & 0xF )
    {
      case 6:
        return sub_1C058(1LL);
      case 7:
        if ( (unsigned int)sub_1C064() )
          v0 = 8;
        else
          v0 = 9;
        continue;
      case 8:
        sub_1D0C0(94LL);
        goto LABEL_2;
      case 9:
LABEL_2:
        v0 = 6;
        break;
      default:
        continue;
    }
  }
}
```

Рис. 2. Декомпилированная обфусцированная функция

1. Алгоритм деобфускации Control Flow Flattening

За основу алгоритма деобфускации взята концепция символьного исполнения [2]. Оно применяется для построения множества $Jumps$ — его элементами являются кортежи вида (a_i, x_i, b_i), где a_i — адрес, откуда был совершён переход на диспетчер; x_i — значение управляющего регистра, при котором диспетчер перейдёт на адрес b_i. Затем множество Jumps преобразуется в набор патчей — информацию об адресах, инструкции по которым нужно заменить в исполняемом файле (алгоритм 1).

Алгоритм 1. Деобфускация Control Flow Flattening

Вход: обфусцированный исполняемый файл вместе с адресом функции, которую необходимо деобфусцировать; функция обнаружения диспетчера $f\colon State \to \to (s, e, c)$, где $State$ — символьное состояние; s — адрес первой инструкции диспетчера, e — адрес последней инструкции диспетчера, c — название управляющего регистра. Функция f определена частично для некоторых значений $State$.

Выход: Набор патчей, упрощающих исполняемый файл. Может содержать адреса, которых изначально нет в программе, для них нужно создать новый исполняемый сегмент.

1: $Jumps = \varnothing$.
2: Символьно исполнять программу с заданного адреса. При исполнении игнорировать вызовы других функций. Если символьные состояния, возможные для исполнения, отсутствуют, перейти на п. 8.
3: Если обнаружен диспетчер (функция f определена для текущего состояния), получить: адрес a, с которого был совершён переход на него, и текущее символьное состояние (регистры и память) $State_0$ вместе с множеством потенциальных значений X управляющего регистра (УР). Так как исполнение символьное, УР в $State_0$ содержит некоторое символьное выражение, которое может быть конкретизировано в множество X допустимых значений УР.
4: Для каждого $x_i \in X$ изменить УР в $State_0$ на x_i, получив тем самым $State_{0i}$.
5: Для каждого $State_{0i}$ продолжать символьное исполнение до тех пор, пока не дойдём до адреса b_i, не относящегося к диспетчеру. Должно быть выполнено условие: для любого $State_{0i}$ существует единственный b_i (так как исполнение символьное, возможных b_i может быть больше одного, но b_i должен зависеть только от значения УР, а оно зафиксировано на шаге 4, поэтому нарушение данного условия свидетельствует о нарушении ограничений, накладываемых на деобфусцируемый код).
6: Добавить все возможные варианты (a, x_i, b_i) к множеству $Jumps$.
7: Перейти на п. 2.
8: // Символьное исполнение закончено, имеется множество $Jumps$, содержащее все возможные переходы для всех обнаруженных диспетчеров.
9: Сгенерировать патчи для каждого уникального a, такого, что $(a, x_i, b_i) \in Jumps$: если для заданного a существует единственная пара (x_i, b_i), заменить инструкцию по адресу a на безусловный переход по адресу b_0. Если пар (x_i, b_i) несколько, создать новый исполняемый сегмент программы и заменить инструкцию по адресу a на переход в него. Сгенерировать в новом сегменте логику сравнения УР с каждым x_i и делать переход на b_i, если и только если УР равен x_i.

2. Накладываемые ограничения на код исполняемого файла

1) Необходимые для работы диспетчера данные находятся либо в read-only памяти, либо инициализируются в той же функции, в которой находится диспетчер.
2) Значения управляющих регистров не зависят от функций, вызываемых внутри деобфусцируемой функции, а также от других функций, не связанных с ней.
3) В каждом диспетчере адрес перехода должен зависеть от управляющего регистра, и только от него.
4) Диспетчер представляет собой непрерывный участок кода.
5) Если в исполняемом файле есть «мёртвый код», который в действительности никогда не будет исполнен, он будет деобфусцирован, как и обычный код. Его обнаружение и удаление выходит за рамки данной работы.

3. Похожие работы

1) Automatic Deobfuscation of Android Native Binary Code [3] — нацелен конкретно на обфускатор OLLVM. Предлагаемый подход более общий и не делает никаких предположений относительно структуры обфусцированной функции.
2) SATURN [4] — не делает совсем никаких предположений относительно кода, вместо этого пытается оптимизировать программу целиком и рекомпилировать её. Такой подход может вносить нежелательные изменения в участки кода программы, свободные от обфускации.

4. Преимущества данного алгоритма

1) Частичное символьное исполнение — не нужно исполнять код целиком. За один раз исполняется только одна функция, даже если она содержит вызовы других функций. Такой подход позволяет работать с очень объёмными программами, на полное символьное исполнение которых требуется слишком много времени и ресурсов.
2) Не делает никаких предположений относительно структуры функции, в отличие от деобфускаторов, которые работают с OLLVM. Все предположения делаются только относительно диспетчеров.

5. Практическая реализация

Алгоритм 1 реализован на языке Python на основе фреймворка для символьного исполнения angr [5]. Реализация поддерживает исполняемые файлы для архитектуры AArch64. Тесты проводились для библиотек Android-приложений, в которых наиболее актуально частичное символьное исполнение, так как в коде присутствует большое количество вызовов Java Native Interface, символьное исполнение которых является проблемой. Наибольшую трудность представляло автоматическое обнаружение диспетчеров и обработка случаев с каскадным расположением диспетчеров, при котором один диспетчер делает переход на другой.

ЛИТЕРАТУРА

1. *Wang C., Hill J., Knight J., and Davidson J.* Software Tamper Resistance: Obstructing Static Analysis of Programs. Technical Report. University of Virginia, USA, 2000.
2. *Boyer R. S., Elspas B., and Levitt K. N.* SELECT — a formal system for testing and debugging programs by symbolic execution // Proc. Intern. Conf. Reliable Software. Los Angeles, California: Association for Computing Machinery, 1975. P. 234–245.
3. *Kan Z., Wang H., Wu L., et al.* Automated Deobfuscation of Android Native Binary Code. 2020. `https://arxiv.org/pdf/1907.06828.pdf`.

4. *Peter Garba, Matteo Favaro* SATURN — software deobfuscation framework based on LLVM // 3rd Intern. Workshop Software Protection, Nov 2019, London. https://arxiv.org/abs/1909.01752.
5. *Shoshitaishvili Y., Wang R., Salls C., et al.* SOK: (State of) The art of war: Offensive techniques in binary analysis // IEEE Symp. Security Privacy. 2016. P. 138–157.

УДК 004.056.5 DOI 10.17223/2226308X/14/30

ПРИМЕНЕНИЕ РАСШИРЕНИЙ АРХИТЕКТУРЫ X86 В ЗАЩИТЕ ПРОГРАММНОГО КОДА

Р. К. Лебедев, И. А. Корякин

Предложен новый подход к защите программного кода от таких инструментов обратной разработки, как декомпиляторы и инструменты символьного исполнения программ. В рамках данного подхода разработан метод запутывания констант, основанный на использовании набора расширений AES-NI процессорной архитектуры x86. Метод реализован для компилятора Clang при помощи инфраструктуры LLVM и протестирован на таких инструментах обратной разработки, как IDA, Ghidra и angr.

Ключевые слова: *защита программного кода, обратная разработка, декомпиляция, символьное исполнение, архитектура x86.*

На сегодняшний день существует множество инструментов, облегчающих обратную разработку программного обеспечения, что увеличивает риски разработчиков, связанные с нарушением авторского права. Обратная разработка применяется для обхода защиты от копирования, заимствования программного кода конкурентами, поиска уязвимостей и многого другого.

К наиболее популярным инструментам обратной разработки относятся декомпиляторы, отладчики и инструменты символьного исполнения программ. В то время как для противодействия отладке существует множество способов, так как отладка ощутимо влияет на процесс выполнения программы, противодействие декомпиляторам и инструментам символьного исполнения не так распространено.

Для противодействия декомпиляторам обычно используются общие методы обфускации [1], не предотвращающие декомпиляцию, но делающие вывод декомпилятора длинным и менее удобным для прочтения аналитиком. В то же время декомпилированный код может оставаться полностью корректным, что позволяет злоумышленнику использовать его даже без понимания принципа действия.

Против инструментов символьного исполнения существуют специфические подходы, использующие проблему экспоненциального взрыва и односторонние функции [2–4]. Однако они влекут дополнительные накладные расходы, связанные с выполнением большого числа ветвлений или односторонних функций, что может делать их неприменимыми для защиты кода, чувствительного к размеру и производительности.

В данной работе предложен новый подход, не оказывающий значимого влияния на производительность программ и обеспечивающий полную неработоспособность распространённых декомпиляторов и инструментов символьного исполнения. Его основой является использование редких процессорных инструкций, поддержка которых может быть не реализована в инструментах обратной разработки.

Процессорная архитектура x86 имеет более тысячи различных инструкций [5]. В обычных программах используется лишь малая часть этого набора, поэтому инстру-