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-

менты обратной разработки вполне могут поддерживать не все инструкции. Соответственно, если искусственно ввести такие инструкции в программу, это может привести к их неработоспособности.

Рассмотрено влияние на эти инструменты инструкций из набора расширений AES-NI, который используется для аппаратного ускорения операций шифрования AES и поддерживается всеми относительно современными процессорами как в 64-битной, так и в 32-битной версии архитектуры x86 [6]. Одной из основных инструкций этого набора расширений является инструкция AESENC, реализующая один раунд шифрования AES:

$$\mathrm{AESENC}(\mathrm{data}, \mathrm{key}) = \mathrm{MixColumns}(\mathrm{ShiftRows}(\mathrm{SubBytes}(\mathrm{data}))) \oplus \mathrm{key}.$$

Здесь SubBytes, ShiftRows и MixColumns — соответствующие операции шифрования AES; data — шифруемый блок; key — раундовый ключ.

Данная инструкция нечасто используется в программном обеспечении (за исключением библиотек для реализации шифрования), поэтому инструменты обратной разработки могут обрабатывать её некорректно, что останется незамеченным в большинстве сценариев использования.

Для проверки этой гипотезы реализовано простое преобразование — запутывание констант в коде. Оно осуществляется путём замены исходного значения константы x выражением времени исполнения x_{obf} следующего вида:

$$x_{\mathrm{obf}} = \mathrm{AESENC}(x', 0).$$

Это выражение будет подсчитываться всякий раз, когда программе понадобится значение x. В свою очередь, x' рассчитывается во время компиляции по следующей формуле:

$$x' = \mathrm{AESDEC}(x, 0).$$

Здесь AESDEC — операция, обратная AESENC.

Преобразование реализовано на уровне промежуточного представления LLVM [7], что обеспечило возможность проверить эффективность метода на реальных программах, например написанных на языке Си (при помощи компилятора Clang).

Для проверки эффективности метода использовалась простая программа на языке Си, имитирующая проверку лицензионного ключа через сравнение ввода с константой (листинг 1), а также декомпиляторы IDA и Ghidra и инструмент символьного исполнения angr [8–10]. В результате ни IDA, ни Ghidra не смогли во время декомпиляции восстановить изначальное значение константы, причём в случае IDA операция сравнения вовсе исчезла, что ещё более затрудняет анализ. Инструмент символьного исполнения angr также не смог выполнить приложенную программу, сообщив об отсутствии поддержки инструкции AESENC.

```
1 #include <stdio.h>
2 int main() {
3     int k;
4     printf("Please enter secret key: ");
5     scanf("%d", &k);
6     if (k == 1337)  printf("Correct license key\n");
7     else  printf("Wrong license key\n");
8 }
```

Листинг 1. Тестовая программа

Таким образом, предложенный метод оказался эффективен против популярных инструментов обратной разработки. Полученные результаты могут быть в дальнейшем использованы для замены всех инструкций программы на альтернативные конструкции, использующие процессорные расширения, что может сделать защищаемую программу недоступной для декомпиляции и символьного исполнения до тех пор, пока в соответствующих инструментах не будет реализована полная поддержка всех расширений.

ЛИТЕРАТУРА

1. *Junod P., Rinaldini J., Wehrli J., and Michielin J.* Obfuscator-LLVM — software protection for the masses // 2015 IEEE/ACM 1st Intern. Workshop Software Protection. 2015. P. 3–9.
2. *Wang Z., Ming J., Jia C., and Gao D.* Linear obfuscation to combat symbolic execution // Proc. European Symp. Research Computer Security. 2011. P. 210–226.
3. *Seto T., Monden A., Yucel Z., and Kanzaki Y.* On preventing symbolic execution attacks by low cost obfuscation // 20th IEEE/ACIS Intern. Conf. Software Eng., Artif. Intelligence, Networking and Parallel/Distributed Comput. (SNPD). 2019. P. 495–500
4. *Лебедев Р. К.* Автоматическая генерация хэш-функций для обфускации программного кода // Прикладная дискретная математика. 2020. № 50. С. 102–117
5. `https://intelxed.github.io/` — Intel XED. 2019.
6. `https://software.intel.com/content/www/us/en/develop/articles/intel-advanced-encryption-standard-instructions-aes-ni.html` — Intel® Advanced Encryption Standard Instructions (AES-NI). 2012.
7. *Lattner C. and Adve V.* LLVM: A compilation framework for lifelong program analysis & transformation // Intern. Symp. Code Generation and Optimization. 2004. P. 75–86
8. `https://www.hex-rays.com/ida-pro/` — IDA Pro. 2021.
9. `https://github.com/NationalSecurityAgency/ghidra` — Ghidra Software Reverse Engineering Framework. 2021.
10. *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 004.052 DOI 10.17223/2226308X/14/31

ПОВЫШЕНИЕ ЭФФЕКТИВНОСТИ ПОИСКА УЯЗВИМОСТЕЙ С ИСПОЛЬЗОВАНИЕМ ТЕХНОЛОГИИ ФАЗЗИНГА В ВИРТУАЛЬНЫХ МАШИНАХ JAVASCRIPT

М. С. Недяк

Описано расширение метода фаззинга виртуальных машин JavaScript, использующего мутации абстрактного синтаксического дерева. Рассматриваются результаты работы алгоритма, реализующего предложенное расширение.

Ключевые слова: *фаззинг, JavaScript, автоматизированный поиск уязвимостей.*

Введение

Существуют различные подходы к фаззингу виртуальных машин JavaScript. Каждый из них имеет некоторые недостатки: малое покрытие кода, малое количество состояний программы, затронутых в течение фаззинга. Кроме того, сам процесс фаззинга виртуальных машин требует большое количество вычислительных ресурсов и процессорного времени. Основная причина этих недостатков заключается в том, что вир-