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-

туальная машина JavaScript требует на вход высокоструктурированный вход — программы, правильные синтаксически и семантически. Большинство текстов программ JavaScript, сгенерированных фаззером, не проходят дальше проверки синтаксиса. Из-за этого увеличивается количество запусков виртуальной машины JavaScript, которые не дают новых результатов.

В работе предложено расширение метода фаззинга с генерацией абстрактного синтаксического дерева. В результате при проведении экспериментов:

1) значительно увеличена скорость нахождения новых путей в целевой программе;
2) увеличено общее количество состояний программы, которые проходит очередь фаззера.

1. Фаззинг с мутацией абстрактного синтаксического дерева

Рассмотрим принцип работы фаззера с мутацией абстрактного синтаксического дерева (рис. 1).

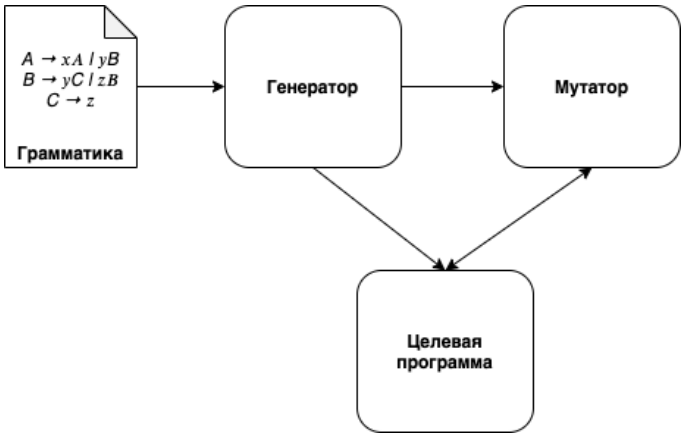

Рис. 1. Схема работы фаззера с мутацией абстрактного синтаксического дерева

Фаззер с мутацией абстрактного синтаксического дерева является комбинированным, то есть это фаззер, где используются и генерационные, и мутационные операции над входными данными программы. Объекты, которые отвечают за генерационные и мутационные операции, называются генератором и мутатором соответственно.

На вход генератору задаются правила грамматики, по которым он генерирует деревья, применяя эти правила в случайном порядке. Примеры фаззеров, которые используют данный подход: Domato [1], Dharma [2], Nautilus [3].

Такой подход генерирует синтаксически правильные тесты, однако их семантическая корректность не гарантируется. Такие тесты обычно не проходят дальше первых компонент виртуальной машины JavaScript — парсера или байт-кода компилятора.

Под семантической неправильностью подразумеваются следующие случаи:

1) При генерации дерева не контролируется тип контекста, например ключевое слово break может появляться только в конструкции switch-case или циклах (листинг 1).

```
1 function test() {
2     var d = 10;
3     break;
4 }
```

Листинг 1. Семантическая неправильность. Пример 1

2) При генерации дерева не контролируются области видимости объявленных переменных. В примере (листинг 2) переменная d не видна вне функции test.

```
1 function test() {
2     var d = 10;
3 }
4 d++;
```

Листинг 2. Семантическая неправильность. Пример 2

3) При генерации дерева не контролируются типы объявленных переменных; в третьем примере (листинг 3) происходит обращение к переменной d, как к массиву, в то время как она является числовой переменной.

```
1 function test() {
2     var d = 10;
3     d[i] = 42;
4 }
```

Листинг 3. Семантическая неправильность. Пример 3

Во всех рассмотренных случаях выполнение кода остановится либо на этапе компиляции байт-кода, либо на этапе интерпретации байт-кода программы JavaScript. Где именно произойдет остановка, зависит от реализации виртуальной машины JavaScript.

2. Расширение метода фаззинга с мутацией абстрактного синтаксического дерева

2.1. Семантический анализ

Предлагается добавить семантический анализ при генерации и мутации абстрактных синтаксических деревьев. Семантический анализ включает в себя:

1) Контроль объявленных переменных. При генерации мутируемого дерева предлагается добавить контроль контекста.
   Контекст — множество названий переменных, классов, функций, которые объявлены и доступны для текущего узла.
2) Контроль типа контекста генерируемого дерева. Это значит, что мутация узла, которая содержит неприменимое в текущем контексте ключевое слово, невозможна.

2.2. Способ задания грамматики JavaScript

Предлагается задавать грамматику с помощью *базы тестов* — набора файлов, содержащих исходный код программ на языке JavaScript. Этими программами могут быть:

1) тесты функциональностей;
2) тесты найденных ошибок;
3) демонстрации найденных уязвимостей.

Так как любую программу на JavaScript можно представить абстрактным синтаксическим деревом, можно считать, что база тестов — это множество абстрактных синтаксических деревьев.

Будем говорить, что грамматика A меньше или равна грамматике B, если мощность языка, порождаемого грамматикой A, меньше или равна мощности языка, порождаемого грамматикой B.

Из теории формальных грамматик [4] известно, что для всякого множества абстрактных синтаксических деревьев существует такая грамматика, что язык, порождённый ею, включает множество этих деревьев. При этом грамматика, образованная множеством деревьев, меньше или равна грамматике всего языка. Другими словами, используя базу тестов, фаззер использует грамматику, построенную по множеству тестов; эта грамматика меньше или равна грамматике языка JavaScript.

С теоретической точки зрения база тестов имеет явные недостатки, потому что не всегда порождает грамматику, равную грамматике всего языка JavaScript. С практической точки зрения задание грамматики базой тестов имеет следующие преимущества:

1) Широта грамматики. В силу специфики реализаций виртуальных машин JavaScript, анализ применимости входной программы к грамматике языка происходит на поздних этапах исполнения, где грамматика задана неявно. Поэтому пользователям других фаззеров приходится вручную анализировать стандарты и примеры работы со стандартными модулями, чтобы включить их в свою грамматику. Так как правила грамматики задаются человеком, велика вероятность, что какие-то конструкции не будут генерироваться этими правилами.
2) Удобство задания грамматики. Тексты программ JavaScript сами по себе задают грамматику, в то время как с ручным заданием грамматики нужно изучать стандарт JavaScript и вручную прописывать правила. Например, при добавлении нового модуля JavaScript в грамматику не нужно анализировать работу с этим модулем, а достаточно добавить набор тестов для него.

3. Алгоритм

Замена узла новым тестом из базы тестов

Метод замены узла в мутируемом дереве происходит по следующим шагам:

1. Запрос из базы тестов узла с таким же типом. В дереве-источнике находится узел, который имеет такой же тип, как заменяемый узел.
2. Анализ применимости узла в текущем контексте. Новый узел применим только тогда, когда он не содержит ключевых слов, которые неприменимы в текущем контексте.
3. Подготовка вставки узла в дерево:
   а) замена идентификаторов переменных, если они:
      – не объявлены в текущем контексте;
      – не объявлены в новой ветке;
      – не являются объектом runtime виртуальной машины JavaScript;
   б) включение веток-функций, веток-классов в глобальный контекст из дерева-источника, если новая ветка вызывает функцию или создаёт класс, при этом они:
      – не объявлены в мутируемом дереве;
      – не являются объектом runtime виртуальной машины JavaScript.

   Новые ветки-функции и ветки-классы также должны пройти подготовку вставки.

Замена узла с обратной связью

Данная мутация является вариацией описанной — замены узла новым тестом из базы тестов, где вместо базы тестов используется очередь фаззера. Для того чтобы определить, что такое очередь фаззера, разберёмся, что такое путь.

Любую программу можно представить ориентированным графом, вершины которого соответствуют прямолинейному участку кода, а дуги — инструкциям перехода.

Такое представление программы называют *графом потока управления* — это множество всех возможных путей исполнения программы.

Путь — это множество вершин графа потока управления, которые были выполнены при запуске программы. Если тест выполнил новый путь, то он помещается в очередь фаззера.

Таким образом, *очередь фаззера* — это множество тестов, которые выполняют разные пути в программе.

4. Экспериментальная часть

Для сравнения эффективности работы фаззеров обычно говорят о метриках покрытия кода. Метрика покрытия кода может задаваться по-разному: это может быть количество выполненных строк кода или количество найденных путей в целевой программе.

В данной работе для сравнения эффективности используется скорость нахождения новых путей, определяемая как количество путей в отношении к количеству запусков. Эта метрика показывает, как часто достигаются новые состояния программы в процессе фаззинга.

Задача фаззинга — это нахождение аварийных завершений программы, некоторые из которых могут сигнализировать о наличии в ней уязвимости. Уязвимости программы содержатся не только в строках кода, но и в состояниях программы, которые могут привести к этой уязвимости. Поэтому используем ещё одну метрику — количество путей, сгенерированных фаззером.

Для сравнения эффективности разработанного фаззера был выбран фаззер Nautilus, который является фаззером с классическим подходом с мутацией абстрактного синтаксического дерева. Идея мутации с обратной связью была взята именно из этого фаззера, поэтому при сравнении с ним можно оценить, как предложенные улучшения алгоритма мутаций повлияли на эффективность подхода.

4.1. Фаззинг виртуальной машины SpiderMonkey

На рис. 2 представлен график скорости генерации новых путей при фаззинге виртуальной машины SpiderMonkey [5]. Можно заметить, что с количеством запусков скорость генерации новых путей Nautilus заметно ниже, чем у DFuzzer, реализующего предложенное расширение. При этом количество путей, сгенерированных за равное количество запусков, у разработанного фаззера больше.

Если посмотреть на график в приближении, то можно видеть, что в начале работы скорость генерации новых путей у фаззера Nautilus выше. Это обусловлено тем, что Nautilus использует генерационные методы для модифицирования входных данных целевой программы. Фаззер DFuzzer является мутационным, это значит, что на его входе задано начальное дерево, с которого он начинает мутации, в то время как генерационный фаззер сразу генерирует все возможные деревья по заданной грамматике.

В ходе экспериментов аварийных завершений виртуальной машины JavaScript SpiderMonkey не выявлено.

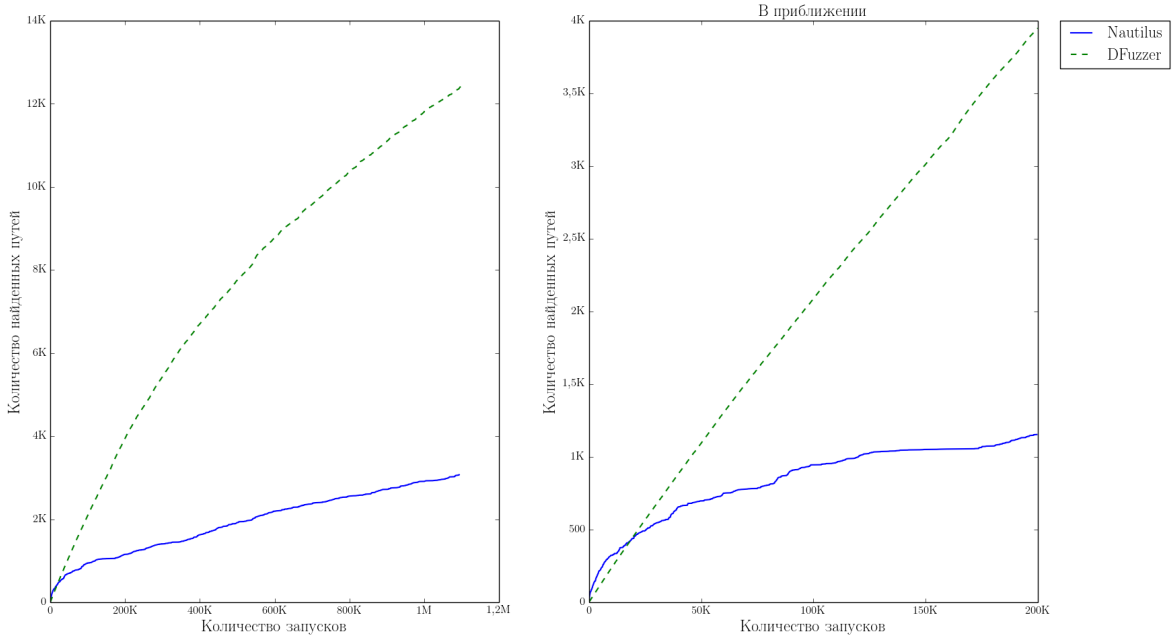

Рис. 2. Скорость генерации новых путей при фаззинге SpiderMonkey

4.2. Фаззинг виртуальной машины ChakraCore

При фаззинге виртуальной машины ChakraCore [6] (рис. 3) можно наблюдать схожую картину, что и при фаззинге виртуальной машны SpiderMonkey.

В ходе тестирования виртуальной машины ChakraCore выявлены аварийные завершения программы. Они будут проанализированы на наличие уязвимостей, и информация о них будет передана производителю.

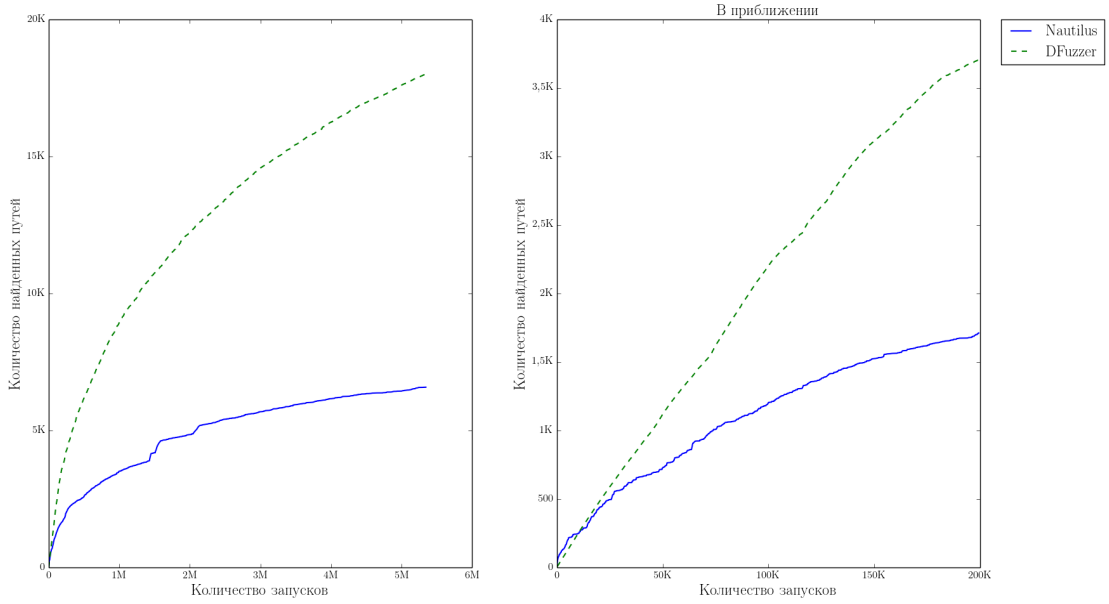

Рис. 3. Скорость генерации новых путей при фаззинге ChakraCore

Заключение

Предложенное расширение является развитием современных подходов фаззинга виртуальных машин JavaScript. В результате удалось существенно повысить эффективность тестирования на уязвимости, расширив метод фаззинга, использующего мутации абстрактных синтаксических деревьев, и значительно увеличить скорость обнаружения новых путей исполнения в тестируемой программе. Исходный код реализованного фаззера доступен для изучения и использования на портале github [7].

ЛИТЕРАТУРА

1. Domato Fuzzer. `https://github.com/googleprojectzero/domato/blob/master/README.md`. 2021.
2. Dharma Fuzzer. `https://github.com/MozillaSecurity/dharma/blob/master/README.md`. 2021.
3. *Aschermann C.* NAUTILUS: Fishing for Deep Bugs with Grammars. `https://www.syssec.ruhr-uni-bochum.de/media/emma/veroeffentlichungen/2018/12/17/NDSS19-Nautilus.pdf`. 2021.
4. *Aho A. V., Lam M. S., Sethi R., and Ullman J. D.* Compilers: Principles, Techniques, and Tools. Addison Wesley, 2007.
5. Виртуальная машина JavaScript SpiderMonkey. `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey`. 2021.
6. Виртуальная машина JavaScript ChakraCore. `https://github.com/chakra-core/ChakraCore`. 2021.
7. `https://github.com/MashaSamoylova/DFuzzer/blob/master/README.md`. 2021.

УДК 004.056 DOI 10.17223/2226308X/14/32

РАСШИРЕНИЕ И ИССЛЕДОВАНИЕ МЕТОДА СОКРЫТИЯ ИНФОРМАЦИИ DEEP STEGANOGRAPHY

А. А. Николаев

Программно реализован метод сокрытия информации с помощью нейронных сетей Deep Steganography. Предложено и реализовано расширение метода в виде добавления дополнительных скрытых слоёв в закодированное изображение для возможности передачи большего объёма информации. Исследованы качества и свойства предложенного метода сокрытия информации.

Ключевые слова: *сокрытие информации, скрытые каналы, нейронные сети.*

1. Сокрытие информации с помощью нейронных сетей

В настоящее время известны два основных метода сокрытия информации с помощью нейронных сетей: Deep Steganography [1] и HiDDeN [2]. Основным их различием является возможный тип передаваемых данных: в случае Deep Steganography в качестве секретного сообщения выступает изображение; в случае HiDDeN — любая битовая последовательность.

В работе исследуется расширение метода сокрытия информации с помощью нейронных сетей Deep Steganography.

Метод Deep Steganography впервые описан в работе [1]. Схема сокрытия и раскрытия секретного изображения состоит из трёх нейронных сетей, которые обучаются одновременно как единая сеть, но работают независимо друг от друга и могут быть