УДК 004.056

П.А. Теплюк, А.Г. Якунин

Методика фаззинга системных вызовов ядра Linux с применением больших языковых моделей

Актуальной проблемой организации фаззинг-тестирования ядра Linux является создание спецификаций системных вызовов – специальных декларативных описаний, которые впоследствии используются фаззером для генерации последовательностей системных вызовов. По большей части это является ручной работой, требующей глубоких знаний, занимающей много времени и не исключающей фактор ошибки. В настоящее время проводятся исследования, направленные на автоматизацию процесса создания таких спецификаций. В работе рассмотрены подходы к генерации спецификаций системных вызовов KSG, SyzDescribe и KernelGPT, которые зарекомендовали себя при обнаружении уникальных сбоев ядра в процессе фаззинг-тестирования. Предложена методика организации фаззинг-тестирования ядра Linux, содержащая в том числе этап автоматической генерации спецификаций системных вызовов на основе больших языковых моделей (Large Language Model – LLM).

Ключевые слова: операционная система, ядро Linux, фаззинг, Syzkaller, спецификации системных вызовов, LLM.

DOI: 10.21293/1818-0442-2024-27-3-85-91

В настоящее время вместе со стремительным развитием компьютерных технологий все более актуальной становится задача обеспечения безопасности операционных систем, сетевых протоколов и программного обеспечения. Операционные системы на базе ядра Linux как основа системного программного обеспечения используются в информационных системах, в том числе являющихся объектами критической информационной инфраструктуры [1].

В соответствии с приказом ФСТЭК России № 239 от 25 декабря 2017 г. «Об утверждении требований по обеспечению безопасности значимых объектов критической информационной инфраструктуры Российской Федерации» [2], в число требований к испытаниям по выявлению уязвимостей в программном обеспечении входит проведение фаззинг-тестирования программы.

При организации фаззинг-тестирования ядра Linux необходимо решить следующие задачи:

– выбор способа запуска ядра системы;

– определение входных данных для проведения фаззинга;

– выбор способа передачи входных данных ядру системы;

– разработка технологии автоматической генерации входных данных;

– определение способов выявления сбоев или уязвимостей системы в процессе ее тестирования;

– разработка системы автоматизации процесса фаззинга.

В предыдущей работе [3] исследованы подходы к определению поверхности атаки ядра Linux в контексте подготовки к фаззинг-тестированию. Определение поверхности атаки позволяет выявить наиболее приоритетные подсистемы, которые необходимо протестировать в первую очередь ввиду достаточно объемной кодовой базы ядра Linux.

В другой работе [4] был проведен анализ существующих подходов и инструментов фаззинга системных вызовов ядра Linux, а также выполнено экс-периментальное фаззинг-тестирование с применением инструмента с открытым исходным кодом Syzkaller [5]. В рамках исследования были выявлены сбои и уязвимости в актуальных версиях ядра, одна из которых – use-after-free (использование памяти после освобождения).

Возвращаясь к вопросу решения проблем при организации фаззинг-тестирования ядра Linux, входными данными для фаззера уровня ядра являются системные вызовы, аргументы к которым необходимо передавать через запуск исполняемого файла.

Инструмент Syzkaller в процессе своей работы генерирует случайные программы на основе спецификаций системных вызовов, запускает их на исследуемой ОС, получает обратную связь в виде достигнутого покрытия кода по ядру и выполняет мониторинг журнала ядра на предмет записей об ошибках. Случайные программы конструируются таким образом, чтобы увеличить размер покрытия.

Актуальной проблемой организации фаззинга ядра с помощью Syzkaller является процесс создания спецификаций системных вызовов – в настоящее время это остается по большей части ручной работой, требующей глубоких знаний ядра и больших затрат времени. Вследствие этого достаточное количество системных вызовов еще не охвачено. Не исключен также человеческий фактор при создании спецификаций, что может привести к ошибкам.

Создание спецификаций системных вызовов в Syzkaller

Фаззер Syzkaller для генерации системных вызовов использует спецификации – сконструированные определенным образом описания интерфейсов ядра, чтобы знать, какие имеются системные вызовы и какие типы аргументов они принимают. Описания составляются вручную разработчиками фаззера.

В процессе своей работы Syzkaller исполняет программы, которые состоят из последовательности системных вызовов. Программы генерируются слу-

чайным образом на основе описаний. Цепочки системных вызовов обычно связаны друг с другом. Например, за системным вызовом open() следует write(), который пишет данные в файл.

Syzkaller составляет список наиболее «интересных» с точки зрения эффективности фаззинга программ, который называется корпусом.

Помимо генерации новых программ с нуля, Syzkaller может взять существующую программу из корпуса и изменить ее. Мутация включает в себя вставку и удаление системных вызовов, а также изменение их аргументов. Измененные программы, которые фаззер пометил в качестве «интересных», также могут включаться в корпус.

Описание грамматики системных вызовов осуществляется с помощью специального декларативного языка Syzlang [6]. Пример грамматики представлен на рис. 1.

```
syscallname "(" [arg ["," arg]*] ")" [type] ["(" attribute* ")"]
arg = argname type
argname = identifier
type = typename [ "[" type-options "]" ]
typename = "const" | "intN" | "intptr" | "flags" | "array" | "ptr" |
           "string" | "strconst" | "filename" | "glob" | "len" |
           "bytesize" | "bytesizeN" | "bitsize" | "vma" | "proc" |
           "compressed_image"
type-options = [type-opt ["," type-opt]]
```

Рис. 1. Пример грамматики описания системного вызова

Пример описаний системных вызовов для MIDI-интерфейса [7] представлен на рис. 2.

```
write$midi(
    fd fd_midi,
    data ptr[in, array[int8]],
    len bytesize[data]
)

read$midi(
    fd fd_midi,
    data ptr[out, array[int8]],
    len bytesize[data]
)

ioctl$SNDRV_RAWMIDI_IOCTL_PVERSION(
    fd fd_midi,
    cmd const[SNDRV_RAWMIDI_IOCTL_PVERSION],
    arg ptr[out, int32]
)

ioctl$SNDRV_RAWMIDI_IOCTL_INFO(
    fd fd_midi,
    cmd const[SNDRV_RAWMIDI_IOCTL_INFO],
    arg ptr[out, snd_rawmidi_info]
)
```

Рис. 2. Спецификация системных вызовов для MIDI-интерфейса

Такие описания уже дальше используются фаззером для генерации, мутации, сериализации, десериализации программ. Программой, как уже ранее было сказано, является в данном случае последовательность системных вызовов с конкретным набором аргументов. Пример программы представлен на рис. 3.

```
r0 = open(&(0x7f0000000000)="./file0", 0x3, 0x9)
read(r0, &(0x7f0000000000), 42)
close(r0)
```

Рис. 3. Пример программы Syzkaller

Подходы к автоматической генерации спецификаций системных вызовов

Как уже было отмечено, актуальным вопросом при организации фаззинг-тестирования ядра Linux с помощью Syzkaller является создание спецификаций системных вызовов – по большей части описания интерфейсов составляются вручную разработчиками, что занимает определенное время и человеческие ресурсы. В последнее время проводятся исследования, направленные на автоматизацию данного процесса.

Авторы работы [8] предлагают подход к генерации спецификаций системных вызовов KSG (Kernel Specification Generation) для фаззеров уровня ядра. Вследствие сложности исходного кода ядра Linux возникают следующие проблемы:

1. Извлечение точек входа – операций подсистем ядра, которые выполняются системными вызовами. Некоторые такие операции могут быть зарегистрированы динамически во время инициализации ядра и загрузки модуля.

2. Типы входных данных для точек входа могут различаться в разных путях выполнения кода, что усложняет их идентификацию.

3. Для генерации спецификаций с использованием языка, применяемого конкретным фаззером, необходимо выполнить сопоставление синтаксиса и семантическое кодирование на основе собранной информации.

Автоматическая генерация спецификаций системных вызовов в KSG выполняется в 3 шага:

1. Сначала извлекается информация о точках входа без привязки к деталям их реализации.

2. На основе полученных точек входа KSG выполняется чувствительный к путям выполнения кода анализ для сбора точных типов входных данных и ограничений диапазона.

3. На основе собранной информации генерируются спецификации системных вызовов на языке Syzlang.

Процесс генерации представлен на рис. 4.

В исследовании [9] предлагается решение под названием SyzDescribe для создания описаний системных вызовов драйверов ядра Linux. Процесс генерации состоит из двух основных этапов:

1. Анализ модулей ядра. SyzDescribe определяет модули ядра по функциям инициализации и связывает их с порядком выполнения во время загрузки ядра. Затем SyzDescribe распознает наличие драйве-

ра ядра, который охватывает более одного модуля, и восстанавливает основные интерфейсы, созданные и предоставленные пользовательскому пространству, т.е. поддерживаемые системные вызовы и соответствующие обработчики, а также имя файла устройства.

2. Анализ обработчиков системных вызовов. Для каждого обнаруженного обработчика системных вызовов восстанавливаются дополнительные сведения об этих интерфейсах: значения команд и типы аргументов, поддерживаемые системным вызовом ioctl(). В конечном итоге SyzDescribe может преобразовать полученную информацию в формат описаний системных вызовов, поддерживаемый фаззером Syzkaller.

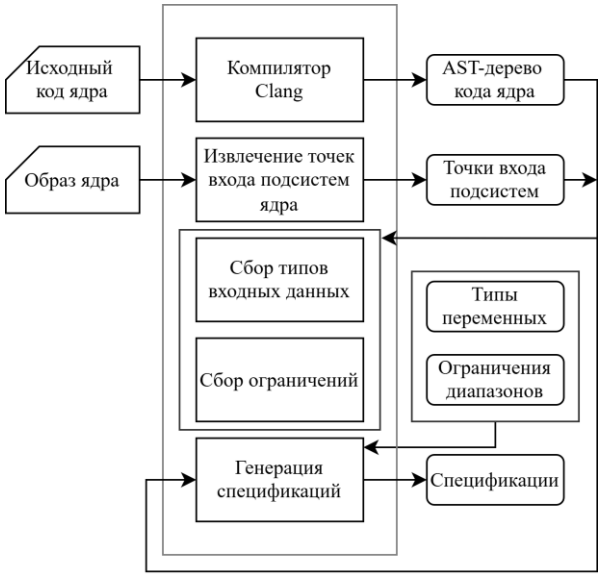

Рис. 4. Процесс генерации спецификаций системных вызовов в KSG

По результатам экспериментов было обнаружено, что разработанные вручную спецификации системных вызовов в Syzkaller составляют менее половины от общего числа спецификаций, генерируемых SyzDescribe. Кроме того, решение было применено к ядру ОС Android смартфона Google Pixel 6, для которого еще не было существующих спецификаций, и обнаружено 18 уникальных сбоев ядра.

В работе [10] предлагается подход, именуемый KernelGPT, в основе которого лежит использование больших языковых моделей (Large Language Model – LLM) для генерации спецификаций системных вызовов. Применение LLM обусловлено, в частности, тем, что именования системных вызовов и аргументов, как правило, выполняются на естественном языке.

Большая языковая модель – это нейронная лингвистическая сеть, обученная на огромных корпусах данных для понимания и обработки текста. Следующие ключевые особенности LLM позволяют применить ее для автоматического создания спецификаций системных вызовов ядра:

– глубокое понимание контекста именования системных вызовов и их аргументов;

– способность генерации описаний с использованием именований системных вызовов на естественном языке.

LLM в последнее время активно внедряются для решения задач в области информационной безопасности [11–14].

В рамках реализации подхода KernelGPT используется модель GPT4 от компании OpenAI [15]. Модель обучается на основе исходного кода ядра, документации и вариантов использования системных вызовов.

Сначала KernelGPT идентифицирует драйверы с помощью LLM, чтобы вывести имена устройств и спецификации их инициализации, используя сведения об обработчиках операций. Затем KernelGPT определяет значения команд, типы аргументов и определения типов для описания обработчиков ioctl() устройств. При этом используется соответствующий исходный код ядра. KernelGPT использует итеративный подход для автоматического включения всех компонентов спецификации.

На рис. 5 представлена архитектура KernelGPT с примером генерации спецификации для системного вызова ioctl(), предназначенного для операций ввода-вывода, специфичных для устройства.

В рамках экспериментальных исследований с помощью модуля извлечения кода было выявлено 132 обработчика ioctl() в конфигурации ядра Syzbot, не считая обработчиков USB и сетевых драйверов. У 50 из них не были обнаружены спецификации в Syzkaller. KernelGPT сгенерировал 39 описаний. Оставшиеся 11 спецификаций не были сгенерированы по следующим причинам:

1. Сложная логика кода затрудняет его понимание со стороны LLM.

2. Анализируемый код может превышать ограничение контекстного окна LLM.

Используя KernelGPT, авторам удалось обнаружить 7 ранее неизвестных сбоев ядра в драйверах, спецификации к которым были сгенерированы автоматически.

Описание методики организации фаззинга системных вызовов

По результатам анализа подходов к автоматической генерации спецификаций системных вызовов была предложена методика фаззинг-тестирования ядра Linux, включающая в себя 5 этапов.

Этап 1. Выбор и развертывание актуальной версии ядра Linux.

Необходимо в основной операционной системе либо в виртуальной машине выполнить сборку фаззера. Также нужно определиться с актуальной версией ядра Linux. Его необходимо загрузить, скомпилировать и создать образ, который далее будет использоваться фаззером для создания тестовых виртуальных машин.

Этап 2. Определение поверхности атаки.

Предварительным этапом организации фаззинг-тестирования, как уже упоминалось ранее, является определение поверхности атаки, т.е. выявление всех неконтролируемых входов в систему, которые не подконтрольны легитимному пользователю, но могут быть потенциально подконтрольны злоумышленнику.

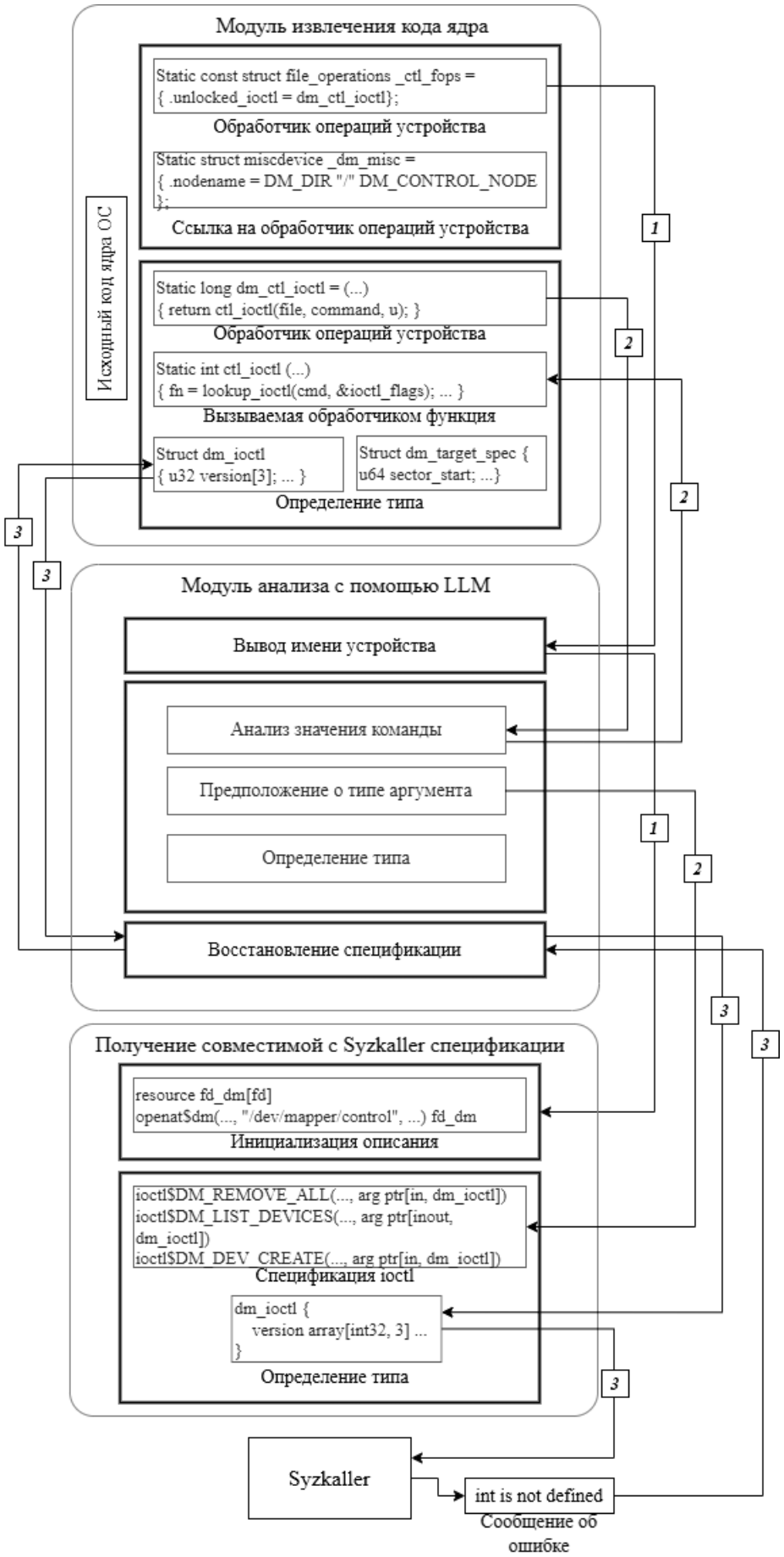

Рис. 5. Архитектура KernelGPT

В предыдущей работе [3] был предложен гибридный подход к определению поверхности атаки ядра Linux, который включает в себя следующие методы:

– измерение метрик сложности кода [16];

– применение интроспекции виртуальных машин;

– динамический анализ помеченных данных [17].

Применение предложенного подхода, в котором сочетаются статические и динамические методы анализа, позволяет более точно построить поверхность атаки, нежели использование методов по отдельности.

Этап 3. Генерирование спецификаций системных вызовов на основе больших языковых моделей.

Фаззеры уровня ядра требуют определенного формата описания системных вызовов. Как правило, такие спецификации составляются вручную разработчиками на декларативном языке Syzlang. Однако были исследованы новые подходы, которые предлагают автоматическую генерацию этих описаний и, судя по результатам экспериментов, показали свою эффективность.

В частности, на данном этапе методики предлагается использовать описанный выше перспективный подход KernelGPT на базе LLM для автоматизации создания спецификаций.

Этап 4. Запуск фаззинг-тестирования.

Фаззинг осуществляется путем запуска системных вызовов с аргументами в виде случайных данных. При фаззинге системных вызовов необходимо решить 2 проблемы:

– корректно сформировать аргументы системных вызовов для прохождения валидации;

– корректно зафиксировать факт того, что ядро ведет себя неожиданным способом.

Многие системные вызовы осуществляют валидацию переданных аргументов. Поэтому необходимо, чтобы фаззер создавал необходимые условия для корректного запуска системных вызовов. Этот подход существенно повышает эффективность тестирования по сравнению с более грубым подходом, когда системные вызовы выполняются с абсолютно случайными аргументами.

Существуют следующие методы генерации входных данных для фаззинга:

1. Модификация существующих данных. Новые данные генерируются путем незначительных изменений имеющихся данных.

2. Генерация новых данных. Данные подготавливаются заранее на основе протоколов или в соответствии с заданными правилами [18].

Для того чтобы понять, что программа ведет себя недокументированным способом, необходимо анализировать возвращаемые значения. Поэтому любой фаззер системных вызовов должен вести журнал всех попыток.

Этап 5. Анализ результатов фаззинг-тестирования.

На данном этапе необходимо проанализировать следующие результаты:

– информацию о найденных сбоях ядра;

– входные данные, приводящие к сбою;

– воспроизводимость найденных сбоев;

– графы вызовов функций;

– взаимосвязь линейных блоков функций.

В рамках этого этапа необходимо также определить, приводит ли сбой к уязвимости.

На рис. 6 представлены этапы фаззинг-тестирования ядра Linux согласно предложенной методике.

Рис. 6. Этапы методики фаззинг-тестирования ядра Linux

Заключение

В работе были проанализированы следующие подходы к автоматическому созданию спецификаций системных вызовов для фаззинга ядра Linux: KSG, SyzDescribe и KernelGPT. Как правило, в своей основе эти подходы предполагают статический анализ кода ядра и дальнейшую обработку полученных системных вызовов, а также типов их аргументов.

Интересным с точки зрения исследования является KernelGPT. KernelGPT – первый подход к автоматическому созданию спецификаций системных вызовов с помощью LLM. Перспективными направлениями исследований являются генерация тестовых наборов данных с помощью KernelGPT, мутации данных для фаззинга и непосредственной генерации последовательностей системных вызовов для программ Syzkaller.

Предложена методика проведения фаззинг-тестирования с использованием LLM. Она включает в себя следующие этапы:

1. Выбор и развертывание актуальной версии ядра Linux и фаззера.

2. Определение поверхности атаки.

3. Генерирование спецификаций системных вызовов с использованием LLM.

4. Запуск фаззинг-тестирования.

5. Анализ результатов.

В дальнейшем планируется экспериментальное тестирование предложенной методики с использованием фаззера Syzkaller.

Исследование выполнено при финансовой поддержке Минцифры России (грант ИБ). Проект № 26/23-К.

Литература

1. Хорошилов А. Международный проект по разработке ядра Linux // Системный администратор. – 2022. – № 3 (232). – С. 32–37.

2. Информационное сообщение ФСТЭК России от 10 февраля 2021 г. [Электронный ресурс]. – Режим доступа: https://fstec.ru/dokumenty/vse-dokumenty/informatsionnye-i-analiticheskie-materialy/informationnoe-soobshchenie-fstek-rossii-ot-10-fevralya-2021-g-n-240-24-647, свободный (дата обращения: 28.08.2023).

3. Теплюк П.А. Модели и подходы к анализу поверхности атаки для фаззинг-тестирования ядра Linux / П.А. Теплюк, А.Г. Якунин // Безопасность информационных технологий. – 2024. – Т. 31, № 1. – С. 135–145. DOI: 10.26583/bit.2024.1.08.

4. Теплюк П.А. Выявление недостатков безопасности ядра Linux с применением фаззинга системных вызовов / П.А. Теплюк, А.Г. Якунин // Проблемы информационной безопасности. Компьютерные системы. – 2024. – № 2 (59). – С. 138–151. DOI: 10.48612/jisp/pdp9-d25r-g6eu.

5. Syzkaller – kernel fuzzer [Электронный ресурс]. – Режим доступа: https://github.com/google/syzkaller, свободный (дата обращения: 07.09.2024).

6. Syscall description language [Электронный ресурс]. – Режим доступа: https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions_syntax.md, свободный (дата обращения: 07.09.2024).

7. Спецификации MIDI-интерфейса [Электронный ресурс]. – Режим доступа: https://github.com/google/syzkaller/blob/master/sys/linux/dev_snd_midi.txt, свободный (дата обращения: 07.09.2024).

8. Sun H. KSG: Augmenting Kernel Fuzzing with System Call Specification Generation / H. Sun, Y. Shen, J. Liu, Y. Xu, Y. Jiang // Proc. of the 2022 USENIX Annual Technical Conference (USENIX ATC 22). – 2022. – P. 351–365.

9. Hao Y. SyzDescribe: Principled, Automated, Static Generation of Syscall Descriptions for Kernel Drivers / Y. Hao, G. Li, X. Zou, W. Chen, S. Zhu, Z. Qian, A. Sani // Proc. of 2023 IEEE Symposium on Security and Privacy (SP). – 2023. – P. 3262–3278. DOI: 10.1109/SP46215.2023.10179298.

10. Yang C. KernelGPT: Enhanced Kernel Fuzzing via Large Language Models / C. Yang, Z. Zhao, L. Zhang // arXiv:2401.00563 [cs.CR]. – 2023. – P. 13. DOI: 10.48550/arXiv.2401.00563.

11. Patsakis C. Assessing LLMs in malicious code deobfuscation of real-world malware campaigns / C. Patsakis, F. Casino, N. Lykousas // Expert Systems with Applications. – 2024. – Vol. 256. – P. 13. DOI: 10.1016/j.eswa.2024.124912.

12. Ye J. Detecting command injection vulnerabilities in Linux-based embedded firmware with LLM-based taint analysis of library functions / J. Ye, X. Fei, X. de Carne de Carnavalet, L. Zhao // Computers & Security. – 2024. – Vol. 144. DOI: 10.1016/j.cose.2024.103971.

13. Lu G. GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning / G. Lu, X. Ju, X. Chen, W. Pei, Z. Cai // Journal of Systems and Software. – 2024. – Vol. 212. DOI: 10.1016/j.jss.2024.112031.

14. Sufi F. An innovative GPT-based open-source intelligence using historical cyber incident reports / F. Sufi // Natural Language Processing Journal. – 2024. – Vol. 7. – P. 17. DOI: 10.1016/j.nlp.2024.100074.

15. GPT4 | OpenAI [Электронный ресурс]. – Режим доступа: https://openai.com/index/gpt-4/, свободный (дата обращения: 09.09.2024).

16. Bavendiek S. Attack surface analysis of the Linux kernel based on complexity metrics. // Master's Thesis in the study course «Applied Informatics / Software Engineering». – 2021. – P. 88. DOI: 10.13140/RG.2.2.29943.70561.

17. Natch: Определение поверхности атаки программ с помощью отслеживания помеченных данных и интроспекции виртуальных машин / П.М. Довгалюк, М.А. Климушенкова, Н.И. Фурсова, В.М. Степанов, И.А. Васильев, А.А. Иванов, А.В. Иванов, М.Г. Бакулин, Д.И. Егоров // Труды ИСП РАН. – 2022. – Т. 34, № 5. – С. 89–110.

18. Томилов И.О. Разработка методики применения фаззинга для анализа уязвимостей программного обеспечения / И.О. Томилов, И.Н. Карманов, П.А. Звягинцева, Е.В. Грицкевич // Системы управления, связи и безопасности. – 2018. – № 4. – С. 48–63.

Теплюк Павел Андреевич
Ст. преп. каф. информатики, вычислительной техники и информационной безопасности
Алтайского государственного технического университета им. И. И. Ползунова
Ленина пр-т, 46, г. Барнаул, Россия, 656038
Тел.: +7-909-506-53-05
Эл. почта: my@teplyukpavel.ru

Якунин Алексей Григорьевич
Д-р техн. наук, проф. каф. информатики, вычислительной техники и информационной безопасности
Алтайского государственного технического университета им. И. И. Ползунова
Ленина пр-т, 46, г. Барнаул, Россия, 656038
Тел.: +7 (385-2) 29-07-86
Эл. почта: almpas@list.ru

Teplyuk P.A., Yakunin A.G.

Methodology for fuzzing Linux kernel system calls using large language models

A pressing issue in organizing Linux kernel fuzzing testing is creating system call specifications – special declarative descriptions that are subsequently used by a fuzzer to generate system call sequences. This is mostly manual work that requires deep knowledge, takes a lot of time, and does not exclude the error factor. Research is currently underway to automate the process of creating such specifications. The paper considers approaches to generate system call specifications KSG, SyzDescribe, and KernelGPT that have proven themselves in detecting unique kernel crashes during fuzz testing. A methodology to organize Linux kernel fuzzing testing is proposed, that includes a stage of automatic generation of system call specifications based on large language models (Large Language Model – LLM).

Keywords: operating system, Linux kernel, fuzzing, Syzkaller, system call specification, LLM.
DOI: 10.21293/1818-0442-2024-27-3-85-91

References

1. Khoroshilov A. [International Linux Kernel Development Project]. *Sistemnyj Administrator*, 2022, no. 3 (232), pp. 32–37 (in Russ.).

2. Informacionnoe soobshchenie FSTEK Rossii ot 10 fevralya 2021 g. [Information message from FSTEC of Russia dated February 10, 2021] (in Russ.). Available at: https://fstec.ru/dokumenty/vse-dokumenty/informatsionnye-i-analiticheskie-materialy/informationnoe-soobshchenie-fstek-rossii-ot-10-fevralya-2021-g-n-240-24-647, free (Accessed: August 28, 2023).

3. Teplyuk P.A., Yakunin A.G. Models and approaches to attack surface analysis for fuzz testing of the Linux kernel. It Security (Russia), 2024, vol. 31, no. 1, pp. 135–145. DOI: 10.26583/bit.2024.1.08.

4. Teplyuk P.A., Yakunin A.G. Identifying security flaws in the Linux Kernel using system call fuzzing. Information Security Problems. Computer Systems, 2024, no. 2 (59), pp. 138–151. DOI: 10.48612/jisp/pdp9-d25r-g6eu.

5. Syzkaller – kernel fuzzer. Available at: https://github.com /google/syzkaller, free (Accessed: September 7, 2024).

6. Syscall description language. Available at: https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions_syntax.md, free (Accessed: September 07, 2024).

7. MIDI system calls specifications. Available at: https://github.com/google/syzkaller/blob/master/sys/linux/dev_snd_midi.txt, free (Accessed: September 07, 2024).

8. Sun H., Shen Y., Liu J., Xu Y., Jiang Y. KSG: Augmenting Kernel Fuzzing with System Call Specification Generation. *Proceedingsof the 2022 USENIX Annual Technical Conference* (USENIX ATC 22), 2022, pp. 351–365.

9. Hao Y., Li G., Zou X., Chen W., Zhi S., Qian Z., Sani A. SyzDescribe: Principled, Automated, Static Generation of Syscall Descriptions for Kernel Drivers. *Proceedings of 2023 IEEE Symposium on Security and Privacy* (SP), 2023, pp. 3262–3278. DOI: 10.1109/SP46215.2023.10179298.

10. Yang C., Zhao Z., Zhang L. KernelGPT: Enhanced Kernel Fuzzing via Large Language Models. arXiv:2401.00563 [cs.CR], 2023, 13 p. DOI: 10.48550/arXiv.2401.00563.

11. Patsakis C., Casino F., Lykousas N. Assessing LLMs in malicious code deobfuscation of real-world malware campaigns. *Expert Systems with Applications*, 2024, vol. 256, 13 p. DOI: 10.1016/j.eswa.2024.124912.

12. Ye J., Fei X., de Carne de Carnavalet X., Zhao L. Detecting command injection vulnerabilities in Linux-based embedded firmware with LLM-based taint analysis of library functions. *Computers & Security*, 2024, vol. 144. DOI: 10.1016/j.cose.2024.103971.

13. Lu G., Ju X., Chen X., Pei W., Cai Z. GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning. *Journal of Systems and Software*, 2024, vol. 212. DOI: 10.1016/j.jss.2024.112031

14. Sufi F. An innovative GPT-based open-source intelligence using historical cyber incident reports. *Natural Language Processing Journal*, 2024, vol. 7, 17 p. DOI: 10.1016/j.nlp. 2024.100074.

15. GPT4 / OpenAI. Available at: https://openai.com/index/gpt-4/, free (Accessed: September 09, 2024).

16. Bavendiek S. Attack surface analysis of the Linux kernel based on complexity metrics. Master's Thesis in the study course «Applied Informatics / Software Engineering», 2021, 88 p. DOI: 10.13140/RG.2.2.29943.70561.

17. Dovgalyuk P.M., Klimushenkova M.A., Fursova N.I., Stepanov V.M., Vasiliev I.A., Ivanov A.A., Ivanov A.V., Bakulin M.G., Egorov D.I. [Natch: using virtual machine introspection and taint analysis for detection attack surface of the software]. Trudy ISP RAN [Proceedings ofISP RAS], 2022, vol. 34, no. 5, pp. 89–110 (in Russ.)

18. Tomilov I.O., Karmanov I.N., Zvyagintseva P.A., Gritskevich E.V. Development of fuzzing application technique for software vulnerabilities analysis. *Systems of Control, Communication and Security*, 2018, no. 4, pp. 48–63.

Pavel A. Teplyuk
Senior Lecturer, Department of Informatics,
Computer Engineering and Information Security,
Polzunov Altai State Technical University
46, Lenin pr., Barnaul, Russia, 656038
Phone: +7-909-506-53-05
Email: my@teplyukpavel.ru

Alexey G. Yakunin
Doctor of Science in Engineering, Professor,
Department of Informatics, Computer Engineering
and Information Security,
Polzunov Altai State Technical University
46, Lenin pr., Barnaul, Russia, 656038
Phone: +7 (385-2) 29-07-86
Email: almpas@list.ru