DOI: 10.15514/ISPRAS-2022-34(4)-3

Инструмент динамического анализа IoT-систем ELF с поддержкой символьных вычислений

Р.Д. Коваленко, ORCID: 0000-0002-2225-5560 <r.kovalenko@ispras.ru>
А.Н. Макаров, ORCID: 0000-0003-2237-3396 <alex1118@ispras.ru>
Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. В результате работы по направлению анализа IoT-устройств авторами был создан инструмент ELF (embedded linux fuzz), который, в частности, предоставляет функционал для применения существующих средств динамического анализа в работе с различными IoT-устройствами. В статье рассматриваются вопросы применения символьного выполнения для анализа IoT-систем, построенных на базе ядер Linux, описывается способ интеграции одного из популярных фреймворков полносистемного символьного выполнения S2E в среду инструмента ELF, а также применимость полученной связки инструментов к реализации распределенного гибридного фаззинга IoT-устройств.

Ключевые слова: фаззинг; символьное выполнение; IoT-устройство; Linux

Для цитирования: Коваленко Р.Д., Макаров А.Н. Инструмент динамического анализа IoT-систем ELF с поддержкой символьных вычислений. Труды ИСП РАН, том 34, вып. 4, 2022 г., стр. 35-48. 10.15514/ISPRAS-2022-34(4)-3

ELF dynamic analysis tool for IoT systems with symbolic execution

R.D. Kovalenko, ORCID: 0000-0002-2225-5560 <r.kovalenko@ispras.ru>
A.N. Makarov, ORCID: 0000-0003-2237-3396 <alex1118@ispras.ru>
Ivannikov Institute for System Programming of the RAS,
25, Alexander Solzhenitsyn st., Moscow,109004, Russia

Abstract. As a result of background work on analysis in embedded Linux OS, the authors created the ELF (embedded linux fuzzing) tool that provides functionality for use in conventional dynamic analysis tools working with IoT devices. The article discusses the use of full-system symbolic execution for the analysis of IoT systems based on Linux kernels, describes how to integrate S2E full-system symbolic execution frameworks into the ELF tool environment, as well as the possibility of applicability of the resulting toolchain to the implementation of distributed hybrid IoT fuzzing.

Keywords: fuzzing; symbolic execution; IoT-device; Linux

For citation: Kovalenko R.D., Makarov A.N. ELF dynamic analysis tool for IoT systems with symbolic execution support. Trudy ISP RAN/Proc. ISP RAS, vol. 34, issue 4, 2022. pp. 35-48 (in Russian). DOI: 10.15514/ISPRAS-2022-34(4)-3

1. Введение

Процесс проведения анализа IoT-устройств зачастую обладает высокой трудоемкостью в силу ряда особенностей реализации и функционирования таких устройств. Архитектура аппаратных компонентов Интернета-вещей разнообразна и, как правило, различается у разных производителей, а также в рамках разных линеек одного производителя. Этот факт повышает трудоемкость анализа ПО данного класса. Другим фактором, осложняющим

анализ, является характерное для IoT-систем отсутствие исходных кодов. При этом, в отличие от десктопных систем, в IoT-устройствах бинарный код объектов анализа содержится внутри образа, формат которого, как правило, заранее не известен, и требуется предварительный анализ для их извлечения. Запуск же целевого бинарного модуля IoT-устройства вне предполагаемой производителем среды выполнения либо невозможен, либо затруднен, либо результат его работы может отличаться от результата в исходной среде запуска. Использование полносистемного эмулятора – один из подходов к решению указанной проблемы.

Для проведения динамического анализа бинарного кода IoT-устройства необходимо внесение дополнительных инструментов в среду выполнения (отладчики, профилировщики и т.д.), что не всегда возможно, а в ряде случаев довольно трудоемко в силу малого объёма памяти IoT-устройства, наличия в нем специфичных файловых систем (ФС) и т.д. При этом, как правило, требуется пересборка образа IoT-устройства, в процессе которой может потребоваться дополнительный анализ реализации загрузчиков, кода проверки целостности и т.п. При этом, даже в тех случаях, когда удалось поместить «навесной» инструмент (например, отладчик) внутрь IoT-устройства, его использование может быть затруднено по причинам ограниченности функциональности ядра.

В силу перечисленных ограничений и особенностей широкое распространение получил подход к анализу безопасности IoT-устройств, основанный на анализе отдельных библиотек с доступным исходным кодом. В этом случае, установив точную версию библиотеки, можно применять к ней средства анализа с учетом наличия исходных кодов (фаззинг, символьное выполнение, статический анализ по исходным кодам и т.д.). Развитие методов анализа по исходным кодам привело к появлению гибридного фаззинга, в котором совмещаются идеи символьных вычислений и фаззинга. Гибридный фаззинг библиотек с открытым исходным кодом показал хорошие результаты [1]. Однако и данный подход не лишен недостатков – ошибки, имеющиеся в коде библиотеки, могут не проявляться вне контекста IoT-устройства. В качестве примера такой ситуации можно привести уязвимость CVE-2018-0101 в Cisco ASA [2]. В уязвимых устройствах при обработке сетевых запросов, содержащих данные в формате xml, был задействован код библиотеки libexpat. Суть уязвимости заключается в том, что при обработке сетевых запросов VPN-сервис IoT-устройства переходит в некоторое состояние, в котором после отправки нескольких пакетов происходит аварийное завершение сервиса с возможностью выполнить произвольный код в его контексте. Примечательно, что уязвимый код, реализующий логику разбора xml, содержится не в самой библиотеке libexpat, а в callback-функциях, относящихся к IoT-устройству. Указатели на эти функции получает библиотека libexpat, и передает управление на них в процессе разбора xml. Следовательно, протестировать уязвимый код вне контекста IoT-устройства невозможно. Таким образом, обнаружение подобного рода ошибок возможно только в процессе проведения полносистемного анализа.

В данной работе рассматривается интеграция в инструмент ELF возможностей полносистемного символьного выполнения с целью поддержки гибридного фаззинга.

2. Обзор архитектуры ELF, общая схема функционирования

В предыдущей статье об инструменте ELF [3] были подробно описаны его базовая архитектура и назначение: инструмент представляет собой платформу на основе полносистемной эмуляции QEMU, обеспечивающую исследование различными инструментами (отладчиков, фаззеров, средств инструментации и т.д.) в штатной среде функционирования программного кода IoT-устройства. В качестве исходных данных для помещения IoT-устройства в среду ELF необходимо наличие образа Linux-подобного ядра, образа ФС, а также образа начальной корневой ФС (initrd). Указанные компоненты предварительно должны быть извлечены из IoT-устройства. При этом стоит обратить

внимание на то, что в случаях, когда удаётся восстановить версию ядра IoT-устройства, а также полностью или частично его конфигурацию, то ядро может быть заново собрано из исходных кодов с добавлением различных модулей, полезных для отладки и анализа.

В ходе проведенных исследований было установлено, что дополнительные инструментальные средства анализа, которые добавляются в начальную корневую ФС, как правило, не влияют на логику функционирования IoT-устройства. Таким образом вносить изменения в образ ФС IoT-устройств не требуется, поскольку весь дополнительный инструментарий помещается в образ initrd. В составе инструмента ELF содержится набор скриптов, которые обеспечивают не только запуск тестов, различных средств анализа, но и выполняют при необходимости пересборку всех компонент в зависимости от сценария тестирования. Тем самым исключаются ошибки в работе ELF, связанные с рассогласованностью отдельных компонентов и модулей.

Интерфейс запуска всех инструментов анализа в ELF унифицирован, а параметры определяются в едином конфигурационном файле. Инструмент ELF может запускаться в нескольких режимах:

- тестовый – в этом режиме IoT-устройство запускается в эмуляторе QEMU. При этом может использоваться любая «ванильная» QEMU версии 3.0 и выше, QEMU ИСП РАН или QEMU из проекта S2E. При запуске IoT-устройства пользователю предоставляется shell-консоль от имени пользователя root. В тестовом режиме возможно проведение фаззинга с помощью внешних сетевых фаззеров, а также снятие трассы.
- отладки – предоставляет те же возможности, что тестовый режим, но добавляется возможность отладки IoT-устройства через интерфейс GDB/QEMU;
- фаззинга – в данном режиме проводится фаззинг целевого ПО IoT-устройства с помощью AFL++;
- символьных вычислений – в данном режиме выполняются символьные вычисления для целевого ПО IoT-устройства.

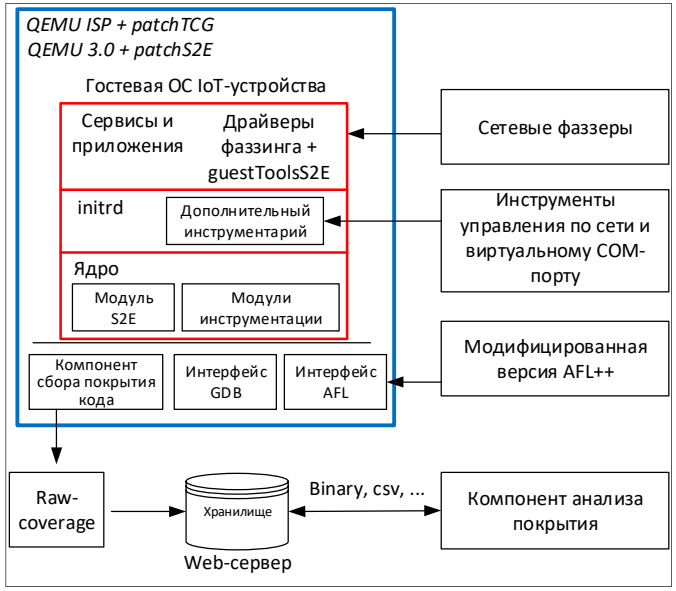


Рис. 1. Основные компоненты инструмента ELF
Fig. 1. The main components of the ELF tool

Базовая часть ELF – эмулятор QEMU. В текущей версии ELF для фаззинга и символьных вычислений поддерживаются две версии QEMU: 1) специализированная версия QEMU ИСП РАН [4], в которую внесены изменения для поддержки фаззинга (patchTCG) и 2) эмулятор на базе QEMU 3.0 из проекта S2E [5,6] с внесенными изменениями (patchS2E).

Инструмент ELF предназначен для исследования IoT-устройств с архитектурами x86_64 и ARM. На рис. 1 приведена схема функционирования ELF и его основные компоненты.

По сравнению с ELF предыдущей версии [3], в новой версии авторами обновлены и добавлены следующие возможности:

- поддержка последних версий AFL++ [7], Boofuzz [8] и Radamsa [9];
- инжектирование драйверов фаззинга в адресное пространство процессов;
- получения командной строки для управления IoT-устройством, посредством виртуальных COM-портов;
- интерфейс добавления различных утилит анализа (strace, tcpdump, gdbserver, busybox, far2l, различных сетевых инструментов и т.д);
- поддержка символьных вычислений (см. разд. 3);
- распределенный гибридный фаззинг с поддержкой docker-контейнеров (см. разд. 4).

Для использования режима отладки среда ELF запускается с GDB-сервером. Данный режим используется для работы с сетевыми фаззерами.

Механизм взаимодействия с AFL++ реализован в виде протокола взаимодействия между QEMU и фаззером, на основе вставки на уровне бинарного кода несуществующей машинной инструкции (рис. 2).

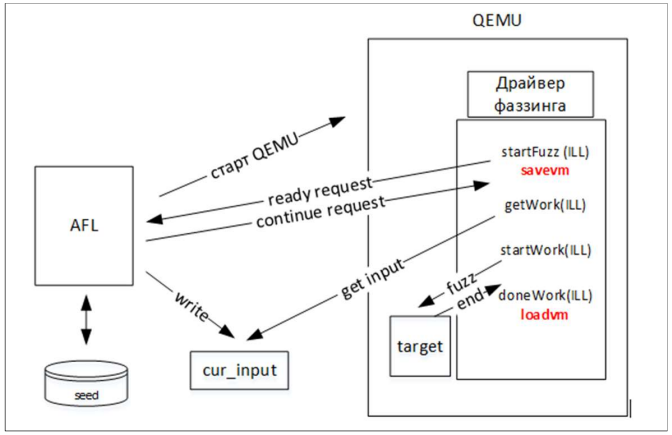


Рис. 2. Протокол взаимодействия AFL и QEMU
Fig. 2. AFL and QEMU interaction protocol

Добавлен дополнительный инструмент для загрузки драйверов фаззинга в адресное пространство IoT-устройства (утилита linuxinjector) – бинарный исполняемый модуль, который обладает функционалом полноценного инжектора разделяемых библиотек в адресные пространства других процессов и предназначен для создания контекста работы драйверов фаззинга в адресных пространствах произвольных процессов IoT-устройства.

Это позволило в новой версии ELF реализовать поддержку драйверов фаззинга в виде отдельных разделяемых библиотек, которые теперь можно внедрять в целевое приложение посредством инжектирования в его адресное пространство. Этот подход позволяет реализовывать логику сетевого фаззинга, вклиниваясь в то или иное место работы штатного

сценария работы исследуемого сетевого сервиса. Это помогает в тех ситуациях, когда, например, внутри IoT-устройства отсутствует возможность напрямую из дополнительного приложения отправлять сетевые пакеты на интересующий TCP-порт (используется dpdk, netmap и т.д.).

В состав дополнительного инструментария добавлена возможность получения командной строки IoT-устройства посредством механизма виртуального соединения COM-портов виртуальный машины и хоста. В процессе запуска QEMU на хосте открывается виртуальный COM-порт (/dev/pts/N), подключение к которому позволяет обеспечить доступ к внутренней командной оболочке (shell) IoT-устройства, или если она отсутствует (малофункциональна), то в IoT-устройство добавляется busybox, который используется в качестве shell-оболочки. Отметим, что данный функционал позволяет получить root-доступ к IoT-устройству даже в тех ситуациях, когда сеть не настроена или отключена.

При сборке компонентов ELF в зависимости от сценария тестирования в образ начальной ФС могут быть добавлены различные утилиты для упрощения изучения IoT-устройства «изнутри». Это могут быть инструменты отладки (strace, ltrace, gdbserver и т.д.), инструменты для работы сетью (tcpdump), shell-оболочки (busybox, far2l) с различными дополнительными утилитами и т.п. Для внедрения подобных утилит требуется их предварительно скомпилировать с помощью набора библиотек и компилятора, соответствующего исследуемому IoT-устройству.

3. Интеграция S2E в ELF

S2E [5, 6] – это платформа для написания инструментов анализа свойств и поведения программных систем. S2E представляет собой модульную библиотеку, которая предоставляет возможности символьного выполнения и анализа программ с использованием *полносистемной* эмуляции в QEMU. Инструмент S2E запускает *немодифицированные* программные стеки x86, x86-64 или ARM, включая программы, библиотеки, ядро и драйверы. Возможности работы с немодифицированными программными стеками в режиме полносистемной эмуляции полностью соответствуют идеологии инструмента ELF, что позволяет совместить работу этих двух инструментов. Инструмент S2E использует фрэймворк символьных вычислений KLEE [10] и решатель Z3 [11]. На рис. 3 приведена схема интеграция инструментов ELF и S2E с основными компонентами.

- Интерфейс управления ELF – унифицированный интерфейс для проведения динамического анализа IoT-устройств, в том числе с помощью символьных вычислений.
- Эмулятор QEMU из S2E, который используется при запуске в режиме символьных вычислений.
- Модифицированное ядро Linux. В S2E для исследования Linux используется исправленное ядро фиксированной четвертой версии, а также модуль ядра s2e.ko. В ELF нет привязки к конкретному ядру – модифицируется оригинальное ядро IoT-устройства, при этом исходные коды не обязательны.
- Образ файловой системы initrd, в который добавляются дополнительные компоненты для запуска S2E:
  - стартовые скрипты для запуска символьных вычислений;
  - библиотека GuestTools – библиотека для взаимодействия с модулями расширения S2E, а также указания символьных переменных;
  - драйвер символьных вычислений – играет роль, аналогичную GuestTools, но ориентирован на тестирование сетевых сервисов.
- Библиотека libs2e.so, которая содержит основной функционал S2E.

Компиляция дополнительных компонент для запуска S2E должна выполняться с помощью соответствующего набора библиотек и компилятора для выбранного IoT-устройства.

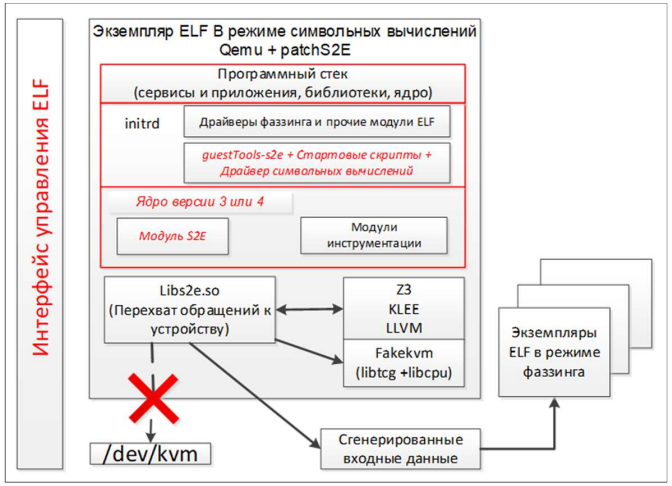


Рис. 3. Схема интеграции инструментов ELF и S2E
Fig. 3. Scheme of integration of ELF and S2E tools

Запуск QEMU в режиме символьных вычислений обладает рядом особенностей. Во-первых, требуется запускать QEMU с параметром LD_PRELOAD=libs2e.so. Данная библиотека содержит всю логику работы с символьными вычислениями, а также реализует логику работы эмулируемого процессора (вместо QEMU). Таким образом, отделяются задачи эмуляции процессора (libs2e.so) от эмуляции периферийных устройств (QEMU).

Во-вторых, QEMU запускается с ключом -enable-kvm, т.е. с точки зрения QEMU она запускается на реальном процессоре и эмулирует только аппаратные устройства посредством интерфейса KVM. Но реально до устройства /dev/kvm дело не доходит, поскольку библиотека libs2e.so перехватывает соответствующие системные вызовы к KVM. Таким образом, вторая значимая часть libs2e.so (отмечена на рис.3 как fakekvm) – это эмуляция процессора и перевод машинного кода в представление LLVM. Часто указывают, что для использования фрэймворка KLEE требуются исходные коды. Действительно, в примерах KLEE для выполнения символьных вычислений требуется использовать компилятор clang. Однако фактически для KLEE требуются не исходные коды, а представление кода в LLVM. Библиотека libs2e.so как раз это и выполняет – эмулирует процессор и получает LLVM представление, которое передаётся на вход KLEE.

В рамках проводимых исследований авторы не изменяли функционал libs2e.so для символьных вычислений, но внесли небольшие изменения в части касающейся эмуляции процессора (добавлена поддержка дополнительных наборов инструкций) для расширения номенклатуры исследуемых IoT-устройств.

4. Применение ELF для исследования сетевых сервисов с помощью символьных вычислений и распределенного фаззинга

В примерах от S2E для указания символьных данных в сетевых пакетах используется SystemTap – средство инструментации кода ядра Linux [12]. На рис. 4 представлен пример использования SystemTap из документации S2E.

Функция s2e_make_symbolic определяет, какие данные являются символьными. Следует обратить внимание на то, что сам код функции s2e_make_symbolic выполняет только вставку машинной «инструкции» с кодом «0x0F 0x3F». Этот машинный код некорректен, т.е. на самом деле вставляется несуществующая машинная инструкция, которая обрабатывается

библиотекой libs2e.so в контексте процесса QEMU, которая затем транслируется в вызов klee_make_symbolic, где и определяются символьные переменные. Такой приём полностью соответствуют принятому в ELF протоколу взаимодействия между фаззером (AFL++) и QEMU, который упомянут раннее в разд. 2.

```
static inline void s2e_make_symbolic( void *buf, int size, const char *name) {
  __asm__ __volatile__(
    ".byte 0x0f, 0x3f\n"
    ".byte 0x00, 0x03, 0x00, 0x00\n"
    ".byte 0x00, 0x00, 0x00, 0x00\n"
    : : "a" (buf), "b" (size), "c" (name) );
} %}
# Take a pointer to the IP header, and make the options length field symbolic
function s2e_inject_symbolic_ip_optionlength (ipheader: long) %{
    uint8_t *data = (uint8_t*)((uintptr_t)(THIS->ipheader + 0));
    uint8_t len;
    s2e_make_symbolic(&len, 1, "ip_headerlength");
    *data = *data & 0xF0;
    *data = *data | ((len) & 0xF);
%}
# Instruct SystemTap to intercept the netif_receive_skb kernel function
probe kernel.function("netif_receive_skb"){
    msg = sprintf("%s: len=%d datalen=%d\n", probefunc(), $skb->len,
                                             $skb->data_len)
    s2e_message(msg)
    s2e_inject_symbolic_ip_optionlength($skb->data)
}
```

Рис. 4. Пример скрипта SystemTap для указания символьных данных
Fig. 4. An example of a SystemTap script for specifying character data

Подход с применением SystemTap целесообразен при наличии исходных кодов ядра, однако практика использования ELF предполагает, что полностью исходный код ядра и конфигурационный файл не доступны, и поэтому анализ проводится на аутентичном бинарном образе ядра (но оно может модифицироваться). Поэтому авторами ELF были разработаны драйверы символьных вычислений (по аналогии с драйверами фаззинга).

- s2esender – сетевой клиент в составе ELF (расширение guestTools S2E) для задания символьных переменных в сетевых протоколах. В качестве символьных переменных может выступать как длина пакетов, так и их содержимое. Также есть возможность работать с цепочками пакетов, т.е. потенциально «пробиваться» через состояния сетевых сервисов.
- s2einjector – инжектор в процессы для указания символьных данных. Выполняет те же действия, что и s2esender, только не запускает новый процесс, а инжектируется в уже существующий процесс. Это удобно для анализа сервисов, которые стартуют при загрузке ОС в IoT-устройстве.

Драйверы символьных вычислений выделяют в сетевом пакете символьные данные (с помощью вызова s2e_make_symbolic) и отправляют его исследуемому сетевому сервису, либо отслеживают функции приема пакетов в сети и в полученных пакетах отмечают символьные данные.

Результатом работы символьных вычислений может быть как выявленная ошибка и вызывающие ее данные, так и набор входных данных для распределенного фаззинга. В последнем случае выходные данные работы ELF в режиме символьных вычислений автоматически передаются экземплярам ELF, запущенным в режиме фаззинга. Для этого авторами ELF был внесен соответствующий программный код в библиотеку libs2e.so.

Очевидно, что в режиме фаззинга ELF должен запускаться во многих экземплярах, их работа координироваться, а результаты – централизованно обрабатываться и храниться. Инструмент ELF может запускаться во многих экземплярах как на хосте, так и в docker-контейнерах. Для управления процессом распределенного фаззинга в настоящее время разрабатывается

система оркестрации на основе Web-приложения. Пользователю инструмента ELF через браузер доступны операции как над группами экземпляров ELF, так и для каждого экземпляра ELF в отдельности. В настоящее время результаты работы ELF хранятся в файловых директориях, но предполагается их хранение в единой базе данных. На рис. 5 приведены снимки экрана со списком задач (левая часть рисунка) и окном для выбора просмотра/управления хостом, docker-контейнером или отображением статистики по текущему фаззингу (правая часть рисунка).

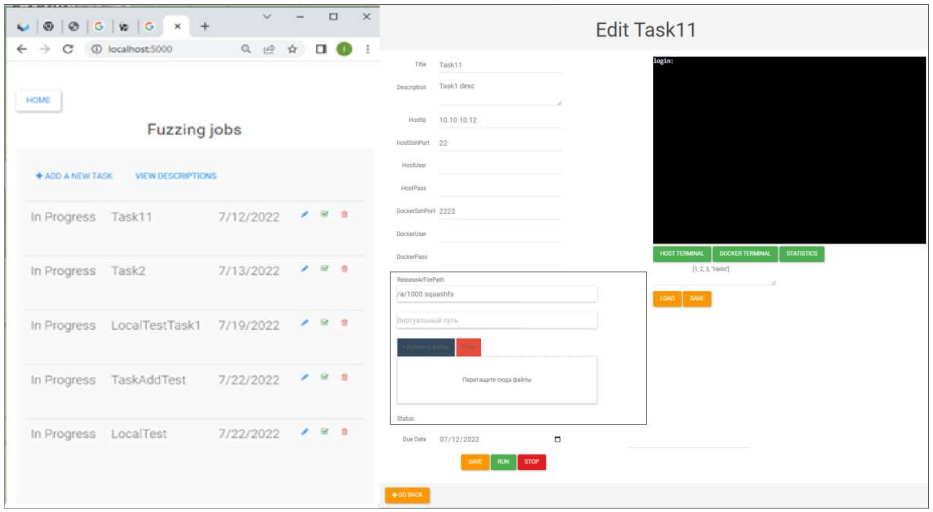


Рис. 5. Снимки экранов системы оркестрации для управления фаззингом
Fig. 5. Screenshots of the orchestration system for managing fuzzing

5. Синтетические тесты

В данном разделе приводятся результаты работы инструмента ELF на различных синтетических тестах. Инструмент ELF для проверки и контроля работоспособности символьных вычислений содержит следующие синтетические тесты:

- набор тестов от s2e;
- консольное приложение;
- сетевой сервер vulnserver;
- набор тестов для ряда известных уязвимостей;
- тест для штатного HTTP-сервера IoT-устройства.

Функционал анализируемого консольного приложения – отображение текущей версии устройства при запуске с параметром «-v». При этом добавлен ключ «-vv», который приводит к аварийному завершению приложения. В результате работы ELF восстановлены все поддерживаемые приложением аргументы, а также зафиксировано аварийное завершение работы приложения при запуске с параметром «-vv» (рис. 6). При этом логируются входные данные, вызвавшие ошибку.

На тестовом примере vulnserver показана возможность восстановления команд простого тестового FTP-подобного сетевого сервера, воспринимающего ограниченный набор простых команд. Инструмент ELF в режиме символьного выполнения успешно восстанавливает все команды в течение нескольких минут (рис. 7).

```
169 [State 2] TestCaseGenerator: generating test case at address 0x80489dd
169 [State 2] TestCaseGenerator:            v0_arg1_0 = {0x2d, 0x76, 0x2d}; (string) "-v-"
169 [State 2] Switching from state 2 to state 4
169 [State 4] LinuxMonitor: Removing task (pid=0x11e, cr3=0x177f000, exitCode=0).
169 [State 4] ProcessExecutionDetector: Unloading process 0x11e
169 [State 4] LinuxMonitor: Removing task (pid=0x11d, cr3=0x177f000, exitCode=0).
169 [State 4] LinuxMonitor: Process ./s2ecmd loaded pid=0x11f
169 [State 4] LinuxMonitor: Process ./s2ecmd loaded pid=0x11f
169 [State 4] LinuxMonitor: ModuleDescriptor Name=s2ecmd Path=./s2ecmd Size=0x6d1ac Address
169 [State 4] ModuleExecutionDetector: module loaded: ModuleDescriptor Name=s2ecmd Path=./s
169 [State 4] BaseInstructions: Killing state 4
169 [State 4] Terminating state: State was terminated by opcode
           message: "bootstrap terminated"
           status: 0x0
169 [State 4] TestCaseGenerator: generating test case at address 0x80489dd
169 [State 4] TestCaseGenerator:            v0_arg1_0 = {0x2d, 0x76, 0x76}; (string) "-vv"
169 [State 4] Switching from state 4 to state 3
169 [State 3] LinuxMonitor: Received segfault type=0 pagedir=0xdc30000 pid=0x11e pc=0x8048
169 [State 3] LinuxMonitor: Blocking searcher until state is terminated
169 [State 3] LinuxMonitor: 0x89 0x7 0x90 0x90 0x89 0x34 0x24 0xeb 0x40 0x57 0x57 0x68 0x5
0x65 0xfe 0xff 0xff 0x89 0x85 0x54 0xff 0xff 0xff 0x83 0xc4 0x14 0x53 0xe8 0xc6 0xfd 0xff
0x8048967:  mov     dword ptr [rdi], eax
0x8048969:  nop
0x804896a:  nop
0x804896b:  mov     dword ptr [rsp], esi
0x804896e:  jmp     0x80489b0
0x8048970:  push    rdi
0x8048971:  push    rdi
0x8048972:  push    0x8048d53
0x8048977:  push    rsi
0x8048978:  call    0x80487a0
```

Рис. 6. Результат работы ELF в режиме символьного выполнения на тестовом консольном приложении

Fig. 6. The result of running ELF in symbolic execution mode on a test console application

```
  64 [State 51] TestCaseGenerator: generating test case at address 0x80489dd
  64 [State 51] TestCaseGenerator:      v0_netpackage_0 =
{0x48, 0x45, 0x4c, 0x50, 0x48, 0x48, 0x48, 0x48, 0x48, 0x48, 0x48, 0x48, 0x48, 0x48}; (string) "HELPHHHHHH
     v1_lenpackage_1 = {0x10, 0x0, 0x0, 0x0}; (int32_t) 16, (string) "...."
  64 [State 51] Switching from state 51 to state 42

          message: "bootstrap terminated"
          status: 0x0
1430 [State 291] TestCaseGenerator: generating test case at address 0x80489dd
1430 [State 291] TestCaseGenerator:      v0_netpackage_0 =
{0x47, 0x4d, 0x4f, 0x4e, 0x47, 0x47, 0x47, 0x47, 0x47, 0x47, 0x47, 0x47, 0x47, 0xx47}; (string) "GMONGGGGGGG
     v1_lenpackage_1 = {0x19, 0x0, 0x0, 0x0}; (int32_t) 25, (string) "...."

          message: "bootstrap terminated"
          status: 0x0
2106 [State 197] TestCaseGenerator: generating test case at address 0x80489dd
2106 [State 197] TestCaseGenerator:      v0_netpackage_0 =
{0x47, 0x44, 0x4f, 0x47, 0x47, 0x47, 0x47, 0x47, 0x47, 0x47, 0x47, 0x47, 0x47, 0x47}; (string) "GDOGGGGGGGGG
     v1_lenpackage_1 = {0x19, 0x0, 0x0, 0x0}; (int32_t) 25, (string) "...."
2106 [State 197] Switching from state 197 to state 237

           message: "bootstrap terminated"
           status: 0x0
 2820 [State 83] TestCaseGenerator: generating test case at address 0x80489dd
 2820 [State 83] TestCaseGenerator:      v0_netpackage_0 =
{0x4b, 0x53, 0x54, 0x45, 0x54, 0x20, 0x54, 0x54, 0x54, 0x54, 0x54, 0x54, 0x54, 0x54}; (string) "KSTET TTTTTT
     v1_lenpackage_1 = {0x11, 0x0, 0x0, 0x0}; (int32_t) 17, (string) "...."
 2820 [State 83] Switching from state 83 to state 441

          message: "bootstrap terminated"
          status: 0x0
2847 [State 144] TestCaseGenerator: generating test case at address 0x80489dd
2847 [State 144] TestCaseGenerator:      v0_netpackage_0 =
{0x4b, 0x53, 0x54, 0x41, 0x4e, 0x20, 0x4b, 0x4b, 0x4b, 0x4b, 0x4b, 0x4b, 0x4b, 0x4b}; (string) "KSTAN KKKKKKK
     v1_lenpackage_1 = {0x11, 0x0, 0x0, 0x0}; (int32_t) 17, (string) "...."
2847 [State 144] Switching from state 144 to state 116
```

Рис. 7. Результат работы ELF с поддержкой полносистемного символьного выполнения на модельном сетевом сервисе

Fig. 7. The result of ELF operation with support for full-system symbolic execution on a model network service

Набор тестов содержит примеры обнаружения известных уязвимостей при запуске ELF в режиме символьного выполнения. Один из таких примеров – нахождение уязвимости CVE-2018-7445 (в сетевом сервисе SMB). В тесте всё содержимое сетевого пакета, а также его длина объявляются символьными переменными. Сформированный сетевой пакет отправляется на TCP-порт SMB с номером 445 (см. рис. 8). В результате уже при проверке четвертого состояния выдаются данные сетевого пакета, приводящие к ошибке.

```
84 [State 41] TestCaseGenerator: generating test case at address 0x80489dd
84 [State 41] TestCaseGenerator:      v0_netpackage_0 =
{0x0, 0x0, 0x0, 0x5, 0xfe, 0x53, 0x4d, 0x42, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}; (string) ".....SMB....
     v1_lenpackage_1 = {0x10, 0x0, 0x0, 0x0}; (int32_t) 16, (string) "...."
84 [State 41] Switching from state 41 to state 12
85 [State 12] LinuxMonitor: Received segfault type=0 pagedir=0x94fe000 pid=0x128 pc=0x80566ed addr=0x0
85 [State 12] LinuxMonitor: Blocking searcher until state is terminated
85 [State 12] LinuxMonitor: 0xf3 0xa4 0x5b 0x5e 0x5f 0x5d 0xc3 0x5b 0x5e 0x5f 0x5d 0xe9 0xaf 0xff 0xff 0xff 0x55
0x83 0xec 0xc 0x8b 0x10 0x50 0xff 0x52 0xc 0x83 0xc4 0x10 0xc7 0x3 0x00 0x00 0x00 0xc7 0x43 0x4 0x00 0x00 0x00 0x8b 0x5d
0x80566ed:  rep movsb byte ptr [rdi], byte ptr [rsi]
0x80566ef:  pop     rbx
0x80566f0:  pop     rsi
0x80566f1:  pop     rdi
```

Рис. 8. Найденная уязвимость CVE-2018-7445 в результате работы ELF в режиме символьного выполнения

Fig. 8. Found vulnerability CVE-2018-7445 as a result of ELF running in symbolic execution mode

Большинство IoT-устройств содержит штатный HTTP-сервер, поэтому набор тестов в ELF содержит универсальный сетевой клиент для отправки символьных данных на TCP-порт с номером 80. На рис. 9 отображены состояния, в которых восстановлены поддерживаемые методы протокола HTTP в исследуемом IoT-устройстве до прохождения пользователем процедуры аутентификации.

```
  266 [State 1470] TestCaseGenerator: generating test case at address 0x80489dd
  266 [State 1470] TestCaseGenerator:      v0_netpackage_0 =
{0x47, 0x45, 0x54, 0x10, 0x9, 0xd, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}; (string) "GET........
     v1_lenpackage_1 = {0x14, 0x0, 0x0, 0x0}; (int32_t) 20, (string) "...."
  266 [State 1470] Switching from state 1470 to state 1479
19680 [State 6371] Terminating state: State was terminated by opcode
          message: "bootstrap terminated"
          status: 0x0
19680 [State 6371] TestCaseGenerator: generating test case at address 0x80489dd
19680 [State 6371] TestCaseGenerator:      v0_netpackage_0 =
{0x50, 0x4f, 0x53, 0x54, 0x80, 0x80, 0x9, 0x2, 0x40, 0xd, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}; (string) "POST....@....
     v1_lenpackage_1 = {0x14, 0x0, 0x0, 0x0}; (int32_t) 20, (string) "...."
19680 [State 6371] Switching from state 6371 to state 6374

          message: "bootstrap terminated"
          status: 0x0
19646 [State 1697] TestCaseGenerator: generating test case at address 0x80489dd
19646 [State 1697] TestCaseGenerator:      v0_netpackage_0 =
{0x50, 0x55, 0x54, 0x1, 0x1, 0x1, 0x9, 0x1, 0x1, 0xd, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}; (string) "PUT............
     v1_lenpackage_1 = {0x14, 0x0, 0x0, 0x0}; (int32_t) 20, (string) "...."
19646 [State 1697] Switching from state 1697 to state 1694
19647 [State 1694] Forking state 1694 at pc = 0x77567c0f at pagedir = 0xdf9f000
    state 1694

           message: "bootstrap terminated"
           status: 0x0
 4338 [State 3132] TestCaseGenerator: generating test case at address 0x80489dd
 4338 [State 3132] TestCaseGenerator:      v0_netpackage_0 =
{0x48, 0x45, 0x41, 0x44, 0x9, 0x8, 0x4, 0x80, 0x1, 0xd, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}; (string) "HEAD........
     v1_lenpackage_1 = {0x14, 0x0, 0x0, 0x0}; (int32_t) 20, (string) "...."
 4338 [State 3132] Switching from state 3132 to state 3134
```

Рис. 9. Восстановление ключевых команд HTTP-сервера IoT-устройства в среде ELF с поддержкой полносистемного символьного выполнения

Fig. 9. Recovery of key commands of the HTTP server of the IoT device in the ELF environment with support for full-system symbolic execution

Одним из показателей качества фаззинга является величина покрытия. Целью применения символьных вычислений при генерации входных данных для фаззеров является увеличение величины покрытия. Авторы ELF провели сравнение величин покрытия при фаззинге AFL++ и фаззинге совместно с символьными вычислениями. На всех проводимых тестах наблюдалось увеличение покрытия. Инструмент ELF позволяет вычислять покрытие на уровне QEMU с применением инструментов интроспекции. Получить покрытие для системного и прикладного ПО IoT-устройства в привычных для многих программистов форматах не всегда возможно, т.к. исходные коды отсутствуют. Средства бинарной инструментации (dynamorio, pin tools и т.п.) не всегда могут быть использованы непосредственно в среде IoT-устройства. Чтобы сравнить величины покрытия, полученные от разных фаззеров, сохранялись все сгенерированные ими входные данные, которые впоследствии передавались на эталонный экземпляр тестового ПО.

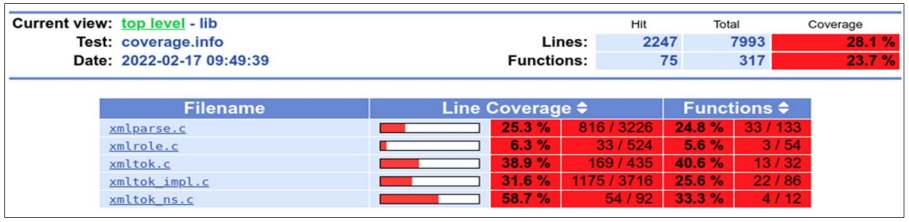
Далее приводится результат только одного теста, который является весьма показательным.

В этом примере использовалось одно из IoT-устройств, содержащее библиотеку libexpat версии 2.1.0. Для проверки инструмент ELF запускался с тремя сценариями фаззинга: Radamsa версии 0.6, AFL++ версии 4.01a и S2E совместно с AFL++. Для теста было создано небольшое приложение (fuzz_test), которое вызывало функцию разбора XML_Parse (точка входа в разбор xml в библиотеки libexpat) для входного файла seed.xml. Приложение fuzz_test запускалось на IoT-устройстве и являлось целевым приложением для фаззинга. Начальный корпус для фаззеров Radamsa и AFL++ состоял из одного файла seed.xml, содержимое которого подготовленного по результатам «ручного» анализа бинарного кода IoT-устройства. В файл seed.xml были добавлены все теги и их параметры, которые были выявлены, чтобы обеспечить фаззерам максимальное покрытие. Для сценария с символьными вычислениями файл seed.xml не использовался, а лишь указывалось, что входной файл является символьным файлом размером 128 байт.

Чтобы максимально точно выполнить замеры величин покрытия, было скомпилировано контрольно-измерительное приложение fuzz_test_control из исходных кодов fuzz_test и libexpat версии 2.1.0 с опциями компилятора для получения покрытия в формате gcov.

Контрольно-измерительное приложение fuzz_test_control использовалось следующим образом:

- в процессе фаззинга сохранялся корпус из всех сгенерированных входных данных или из тех данных, которые сам фаззер определяет как минимальный корпус для достижения заданного покрытия;
- независимо от того, вычисляет ли фаззер самостоятельно покрытие или нет, для измерения покрытия используется fuzz_test_control;
- измерение покрытия выполняется следующим образом: для каждого из трех сценариев все сохраненные входные данные последовательно подаются на вход fuzz_test_control – в результате для каждого сценария получается покрытие в формате gcda.

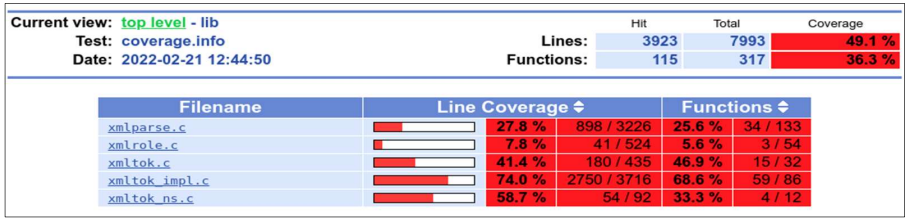

Current view: top level - lib
Test: coverage.info
Date: 2022-02-17 09:49:39

	Hit	Total	Coverage
Lines:	2247	7993	28.1 %
Functions:	75	317	23.7 %

Filename	Line Coverage		Functions	
xmlparse.c	25.3 %	816 / 3226	24.8 %	33 / 133
xmlrole.c	6.3 %	33 / 524	5.6 %	3 / 54
xmltok.c	38.9 %	169 / 435	40.6 %	13 / 32
xmltok_impl.c	31.6 %	1175 / 3716	25.6 %	22 / 86
xmltok_ns.c	58.7 %	54 / 92	33.3 %	4 / 12

Рис. 10. Покрытие, полученное фаззером Radamsa версии 0.6
Fig. 10. Coverage obtained by Radamsa fuzzer version 0.6

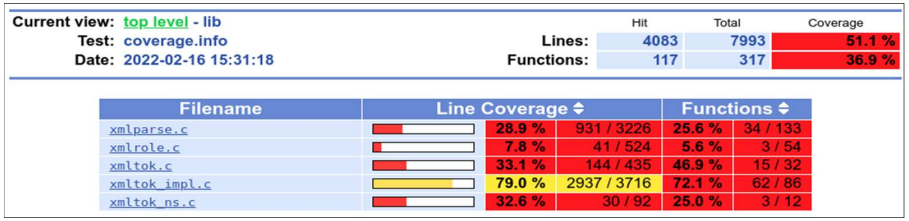
На рис. 10, 11 и 12 показаны результаты измерений величин покрытия, полученные фаззерами Radamsa, AFL++ и гибридным фаззингом. В табл. 1 сведены результаты измерений для указанных фаззеров.


Current view: top level - lib
Test: coverage.info
Date: 2022-02-21 12:44:50

	Hit	Total	Coverage
Lines:	3923	7993	49.1 %
Functions:	115	317	36.3 %

Filename	Line Coverage		Functions	
xmlparse.c	27.8 %	898 / 3226	25.6 %	34 / 133
xmlrole.c	7.8 %	41 / 524	5.6 %	3 / 54
xmltok.c	41.4 %	180 / 435	46.9 %	15 / 32
xmltok_impl.c	74.0 %	2750 / 3716	68.6 %	59 / 86
xmltok_ns.c	58.7 %	54 / 92	33.3 %	4 / 12

Рис. 11. Покрытие, полученное фаззером AFL++ версии 4.01a
Fig. 11. Coverage obtained by fuzzer AFL++ version 4.01a


Current view: top level - lib
Test: coverage.info
Date: 2022-02-16 15:31:18

	Hit	Total	Coverage
Lines:	4083	7993	51.1 %
Functions:	117	317	36.9 %

Filename	Line Coverage		Functions	
xmlparse.c	28.9 %	931 / 3226	25.6 %	34 / 133
xmlrole.c	7.8 %	41 / 524	5.6 %	3 / 54
xmltok.c	33.1 %	144 / 435	46.9 %	15 / 32
xmltok_impl.c	79.0 %	2937 / 3716	72.1 %	62 / 86
xmltok_ns.c	32.6 %	30 / 92	25.0 %	3 / 12

Рис. 12. Покрытие, полученное в результате гибридного фаззинга
Fig. 12. Coverage resulting from hybrid fuzzing

Табл. 1. Сравнение покрытий, полученных от различных фаззеров
Table 1. Comparison of coverages obtained from different fuzzers

фаззер	**Количество строк (всего 7993), %**	**Количество функций (всего 317), %**	**Начальный seed**	**Время (ч.)**
Radamsa 0.6	2247 28,1%	75 23,7%	seed.xml	96
AFL++ 4.01a	3923 49,1%	115 36,3%	seed.xml	96
Гибридный фаззинг: S2E+ AFL++ 4.01a	4083 51,1%	117 36,9%	отсутствует	24 часа S2E 24 часа AFL++

Как видно из табл. 1, разница между покрытием, полученным от AFL++ и гибридным фаззингом, незначительная – всего 2 %. Однако, гибридный фаззинг достиг указанной величины покрытия в два раза быстрее – суммарно за 48 часов. На практике ELF в режиме символьных вычислений запускается параллельно с экземплярами ELF в режиме фазинга и по мере генерации входных данных передает их фаззерам AFL. Важно отметить, что для того чтобы AFL++ достиг покрытия 49%, потребовался ручной анализ кода IoT-устройства для определения тегов, которые были указаны в файле seed.xml. В сценарии же с гибридным фаззером предварительного анализа не требовалось. Входной файл, как таковой, отсутствовал, была указана его длина (128 байт), а все его содержимое было объявлено символьной переменной.

5. Заключение

Версия ELF с поддержкой символьных вычислений и гибридного фаззинга успешно продемонстрировала возможность полносистемного анализа IoT-устройств с применением

символьных вычислений. Тот факт, что для некоторых известных CVE инструмент ELF в символьном режиме находит входные данные, приводящие к аварийному завершению, позволяет рассматривать символьные вычисления не только как способ генерации входных данных, но и инструмент поиска ошибок в контексте IoT-устройствах. При этом, следует отметить, что тестовые CVE подобраны для демонстрации таких ситуаций, в которых поиск ошибок в уязвимых библиотеках вне контекста их использования в IoT-устройстве не даёт результата.

В качестве развития среды ELF с полносистемным символьным выполнением можно выделить следующие направления:

- автоматизированное внесение функционала S2E в бинарный образ ядра;
- переход на новую версию KLEE;
- выбор оптимального решателя;
- улучшение функционирования моделей функций;
- отображение покрытия в формате cachegrind в символьном режиме;
- переход на более новую версию QEMU в символьном режиме работы S2E;
- создание единой базы данных для хранения и обработки результатов распределенного гибридного фаззинга.

Список литературы / References

[1]. OSS-Sydr-Fuzz: Hybrid Fuzzing for Open Source Software. Available at: https://github.com/ispras/oss-sydr-fuzz.git, accessed 16.08.2022.
[2]. C. Halbronn. The Return of Robin Hood vs Cisco ASA Available at: https://www.nccgroup.trust/globalassets/newsroom/uk/events/offensivecon2018-the-return-of-robin-hood-vs-cisco-asa.pdf, accessed: 16.08.2022.
[3]. Коваленко Р.Д., Макаров А.Н. Динамический анализ IoT-систем на основе полносистемной эмуляции в QEMU. Труды ИСП РАН, том 33, вып. 5, 2021 г., стр. 155-166. DOI: 10.15514/ISPRAS–2021–33(5)–9 / Kovalenko R.D., Makarov A.N. Dynamic analysis of IoT systems based on full-system emulation in QEMU. Trudy ISP RAN/Proc. ISP RAS, vol. 33, issue 5, 2021, pp. 155-166 (in Russian).
[4]. P. Dovgalyuk. Deterministic Replay of System's Execution with Multi-target QEMU Simulator for Dynamic Analysis and Reverse Debugging. In Proc. of the 16th European Conference on Software Maintenance and Reengineerin, 2012, pp. 553-556.
[5]. V. Chipounov, V. Kuznetsov, G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. ACM SIGPLAN Notices, vol. 46, issue 3, 2011, pp 265-278.
[6]. S2E: The Selective Symbolic Execution Platform. Available at: http://s2e.systems/docs/, accessed 16.08.2022.
[7]. The fuzzer afl++. Available at: https://github.com/AFLplusplus/AFLplusplus, accessed 16.08.2022.
[8]. J. Pereyda. Boofuzz Documentation, Release 0.4.1. Available at: https://buildmedia.readthedocs.org/media/pdf/boofuzz/latest/boofuzz.pdf, accessed 16.08.2022.
[9]. radamsa: a general-purpose fuzzer. Available at: https://gitlab.com/akihe/radamsa, accessed 16.08.2022.
[10]. KLEE Symbolic Execution Engine. c https://klee.github.io/, accessed 16.08.2022.
[11]. The Z3 Theorem Prover. The Z3 Theorem Prover. Available at: https://github.com/Z3Prover/z3, accessed 16.08.2022.
[12]. Using SystemTap with S2E. Available at: http://s2e.systems/docs/Tutorials/SystemTap/index.html, accessed 16.08.2022.

Информация об авторах / Information about authors

Алексей Николаевич МАКАРОВ – научный сотрудник отдела компиляторных технологий. Сфера научных интересов: информационные технологии, безопасность ПО, анализ бинарного кода, системное программирование, фаззинг, исследования программ.

Aleksey Nikolaevich MAKAROV – Researcher at the Department of Compiler Technologies. Research interests: information technology, software security, binary code analysis, system programming, fuzzing, software research.

Роман Дмитриевич КОВАЛЕНКО – научный сотрудник отдела компиляторных технологий. Сфера научных интересов: информационные технологии, безопасность ПО, анализ бинарного кода, системное программирование, фаззинг, исследования программ.

Roman Dmitrievich KOVALENKO – Researcher, Compiler Technology Department. Research interests: information technology, software security, binary code analysis, system programming, fuzzing, software research.