tarantool

Делаем фаззер для Lua на основе libFuzzer и AFL

Сергей Бронников

Обо мне

Делаю мир лучше в команде Tarantool

Люблю рандомизированное тестирование

Пишу в t.me/sqaunderhood, подпишитесь

Работал над разными интересными проектами (вставить логотипы компаний)

Продам книгу Майерса “Искусство тестирования программ”

Делал слайды в последний момент

Tarantool

- Платформа для in-memory вычислений
- Описание бизнес-логики на Lua
- Используем много кода на Lua в разработке:
  - Продукты и модули на “чистом” Lua
  - Модули на Lua, встроенные в C/C++ (18 KLOC)
  - Модули на Lua, использующие C/C++ библиотеки (19 KLOC)

Тестирование Tarantool

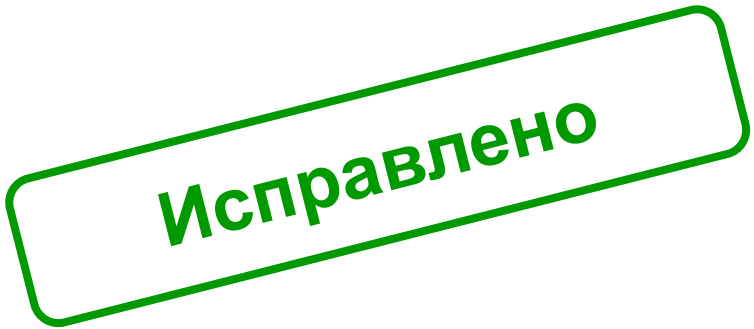
- Юнит-тестирование для кода на Си
- Системные тесты на Lua
- Регрессионное тестирование для каждого PR
- Патчи с исправлениями всегда включают тесты
- 90% тестов используют публичный Lua API
- Покрытие кода:
  - 85% по строкам
  - 54% по веткам

Баг в Tarantool #4773: три байта «смерти»

```
tarantool> '\x36\x00\x80'

$ echo $?       # процесс тихо закрывается

0
```



Баг в Tarantool #6781: один байт «смерти»

```
tarantool> box.cfg{ listen=3303 }

tarantool> require('net.box').connect('3303'):call('\x8a')

$ echo $?       # процесс тихо закрывается

0
```

Тестирования всё ещё недостаточно!

Принципиальные подходы к тестированию

- Тестирование с помощью примеров (example-based)
  - Тестирование “известного”, данные заранее известны
  - Паттерн: Arrange - Act - Assert
- Рандомизированное тестирование
  - Тестирование “неизвестного”, данные всегда случайны
  - Фаззинг - с обратной связью (coverage-guided)
  - Тестирование с помощью свойств (property-based testing)
  - Эффективность фаззинга подтверждена эмпирически (найдено ~40k багов в 650 проектах)

Примеры **уязвимостей**, найденных в OSS Fuzz

- iPhone 6s
  - libxml2: CVE-2019-8749, CVE-2019-8756
  - WebKit: CVE-2019-8734
- iPhone 6s+, iPad Air 2+, iPad mini 4+
  - libxslt: CVE-2019-8750
  - WebKit: CVE-2019-8710, CVE-2019-8766, CVE-2019-8773
- iPhone 5+, iPad (4-го поколения), iPod touch (6-го поколения).
  - SQLite: CVE-2017-2518, CVE-2017-2520, CVE-2017-2513

Протестируем функцию сложения

с использованием примеров, PBT и фаззинга

Это функция сложения целых чисел с сюрпризом

```
def add(x, y):
    if x == 2022 and y == 2023:
        return y - x    # сюрприз!
    return x + y
```

Тестирование на основе примеров: тесты

```
def test_add_example(self):
    self.assertEqual(add(1, 1), 2)                  # простой пример
    self.assertEqual(add(100, 0), 100)              # сложение с нулём
    self.assertEqual(add(12, 13), add(13, 12))      # коммутативность
```

Тестирование на основе примеров: результат

```
$ python3 -m unittest add_tests.TestSuiteAdd.test_add_example

...

Ran 1 test in 0.000s

OK
```

Тестирование с помощью свойств: тесты Hypothesis

```
# Два случайных целых числа
@given(arg1=st.integers(), arg2=st.integers()
def test_add_hypothesis(self, arg1, arg2):
    self.assertEqual(add(arg1, arg2), arg1 + arg2)
    self.assertEqual(add(arg1, arg2), add(arg2, arg1))
    self.assertEqual(add(arg1, 0), arg1 + 0)
```

Тестирование с помощью свойств: результат

```
$ python3 -m unittest add_tests.TestSuiteAdd.test_add_hypothesis

...

Ran 1 test in 0.113s

OK
```

Тестирование с помощью фаззинга: тесты Atheris (1/2)

```
def TestOneInput(input_bytes):
    fdp = atheris.FuzzedDataProvider(input_bytes)
    arg1 = fdp.ConsumeInt(10)    # генерируем знаковое целое
    arg2 = fdp.ConsumeInt(10)    # генерируем знаковое целое
    self.assertEqual(add(arg1, arg2), arg1 + arg2)
    self.assertEqual(add(arg1, arg2), add(arg2, arg1))
```

Тестирование с помощью фаззинга: тесты Atheris (2/2)

```
def test_add_atheris(self):
      atheris.Setup(sys.argv, TestOneInput)
      atheris.Fuzz()          # запускаем фаззинг
```

Тестирование с помощью фаззинга: результат

```
$ python3 -m unittest add_tests.TestSuiteAdd.test_add_atheris

AssertionError: 1 != 4045

...

Ran 1 test in 0.137s
```

Рандомизированное тестирование **эффективнее** с обратной связью

Вернёмся к Tarantool

Место Lua API в Tarantool

- 37 встроенных модулей Lua
- **Основной интерфейс** для использования Tarantool
- Было бы здорово применить фаззинг к публичному Lua API

Фаззинг в Tarantool

- Используем libFuzzer для публичных C/C++ функций
- Запускаем фаззеры в CI в инфраструктуре OSS Fuzz
- Нашли несколько багов в различных компонентах
- libFuzzer **не подходит** для тестирования Lua API
- А мне так хотелось бы использовать libFuzzer для тестирования Lua API

Почему libFuzzer не подходил, изначально, для тестирования Lua API?

Проблема фаззинга Lua API как C: приватные функции

✓ `datetime.parse` (публичная Lua-функция) →

✗ `datetime_parse_from` (Lua) →

✗ `builtin.tnt_datetime_strptime` (FFI) →

✓ `tnt_datetime_strptime` (C) →

✓ `datetime_strptime` (C) →

✓ `tm_to_datetime` (C)

Проблема фаззинга Lua API как C: функции с lua_State

✓ `msgpack.decode` (публичная Lua-функция) →

✗ `luamp_iterator_decode` (Lua C API) →

✗ `luamp_iterator_decode` (Lua C API) →

✗ `luamp_decode` (Lua C API) →

✓ `mp_decode_*` (C)

Нужен фаззинг с поддержкой Lua

Поиск фаззера для Lua

- Разработать с нуля
- lua-quickcheck - тестирование с помощью свойств
- afl-lua - интеграция с AFL, популярным фаззинг движком
- Интеграция с libFuzzer, популярным фаззинг движком

Поиск фаззера для Lua: написать с нуля

- ✓ Интересная инженерная задача
- ✓ Полная свобода: любую часть фаззера можно будет изменить
- ✗ Все части фаззера надо будет писать с нуля
- ✗ Нужно будет изучать другие фаззеры и читать научные работы

Поиск фаззера для Lua: lua-quickcheck

✓ Тестирование в стиле QuickCheck (тестирование с помощью свойств)

✓ Тесты в стиле юнит-тестов, удобно

✗ Не эффективен - тестирование с помощью перебора входных значений

```
property 'sum of numbers is equal to arg1 + arg2' {
  -- генератор вернёт два целых числа
  generators = { int(10^5), int(10^5) },

  check = function(arg1, arg2)
    -- проверим функцию add() для двух чисел
    return add(arg1, arg2) == arg1 + arg2
  end
}
```

Долго тестировать

Поиск фаззера для Lua: afl-lua

- ✓ Эффективный
  - В основе AFL (American Fuzzy Lop), см. FuzzBench
  - Инструментирует Lua-код
- ✗ Объект тестирования - программа целиком, а не функция
- ✗ Не годится для тестирования встроенных в Tarantool модулей

Не подходит

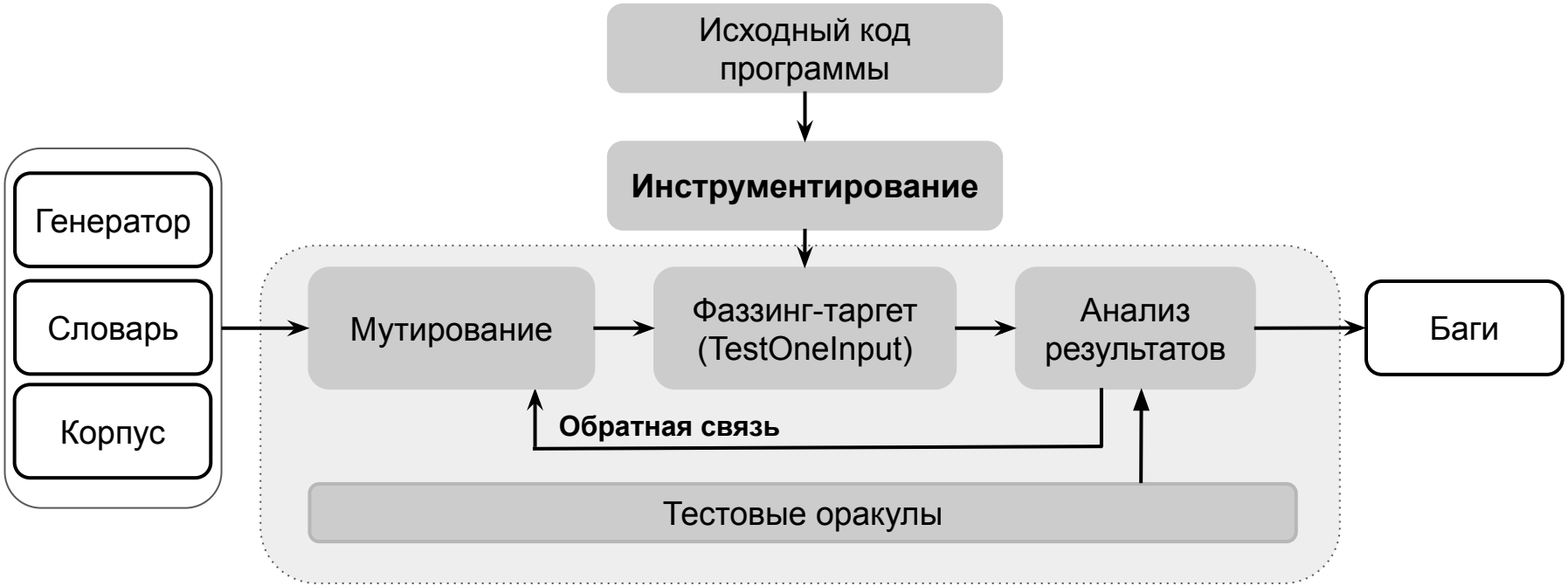
Поиск фаззера для Lua: интеграция с libfuzzer

- ✓ Эффективный, см. результаты FuzzBench
- ✓ Тесты в стиле юнит-тестов
- ✓ Используется в Tarantool для C/C++
- ✓ Простой API - 4 функции
- ✓ Есть интеграция с Python/Java, которые можно использовать как референс
- ✓ Не добавляет зависимостей, libFuzzer это часть Clang
- ✗ Не доступен для изменений
- ✗ Не планируется активно развивать

Подходит

Выбрали интеграцию с libFuzzer

Что такое libFuzzer: общая схема



Весь секрет успеха libFuzzer в **инструментировании** и **обратной связи**

Making Software Dumber~~er~~

Tavis Ormandy, Google, 2011

Обратная связь в Clang: инструментирование

- Инструментирование автоматически включается во время сборки C/C++ опцией `-fsanitize=fuzzer`
- Инструментирование для libFuzzer включает в себя:
  - Инструментирование управления потоком (control flow)
  - Инструментирование операций с данными (data flow)
- Вносит замедление от 0 до 25%, но сильно сокращает время фаззинга

Шаги инструментирования **PC** в Clang

- Определяем функцию __sanitizer_cov_trace_pc_guard_init()
- Определяем функцию __sanitizer_cov_trace_pc_guard()
- Включаем инструментирование - `-fsanitize-coverage=trace-pc-guard`

Пример инструментирования кода

```
$ clang -g -fsanitize-coverage=trace-pc-guard trace-pc.c

$ ./a.out

INIT: 0x55efb467cb70 0x55efb467cb80

guard: 0x55efb467cb74 2 PC 0x55efb4665e7a in main trace-pc.c:24

guard: 0x55efb467cb78 3 PC 0x55efb4665ea9 in main trace-pc.c:25:6
```

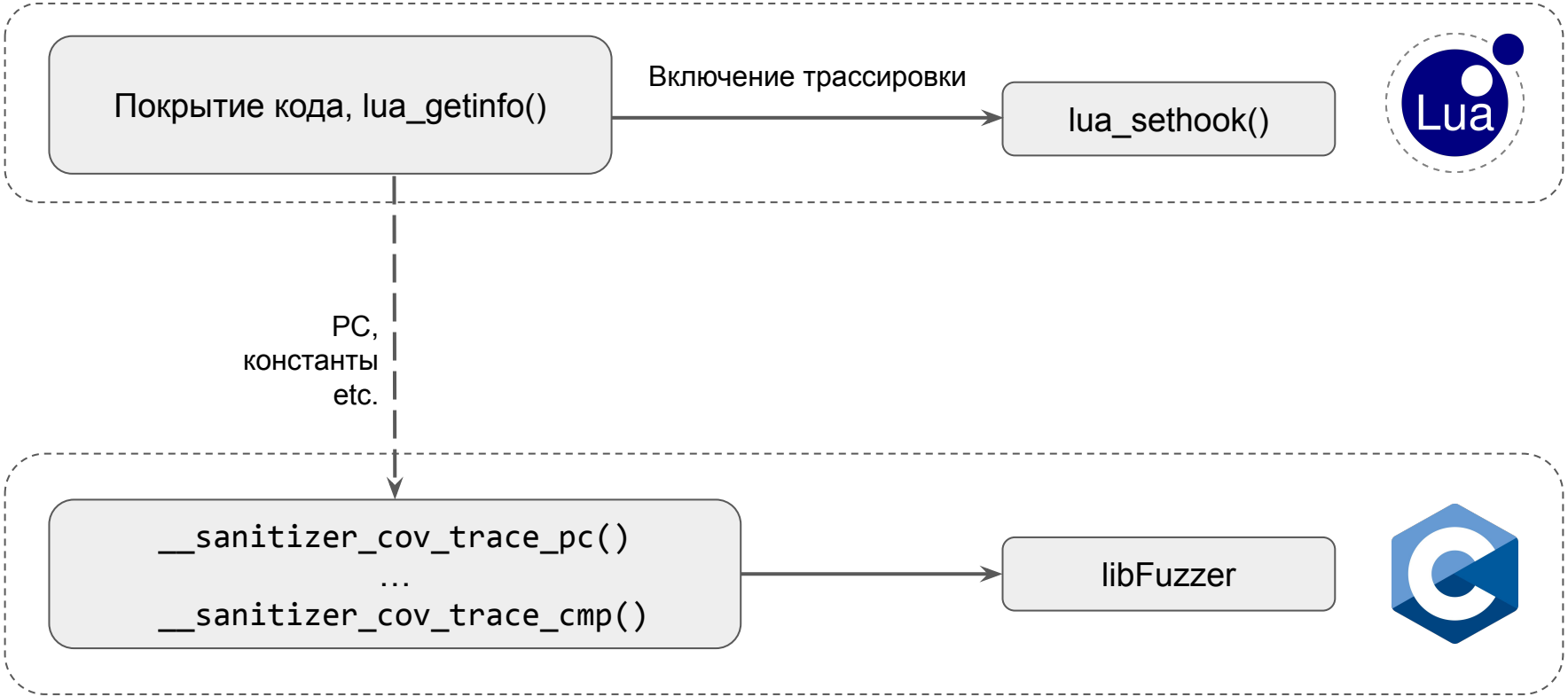
Таким образом
libFuzzer - это фаззер с обратной связью

Интеграция libFuzzer с Lua

Есть 4 точки интеграции с libFuzzer

- Функция задания фаззинг-таргета, **LLVMFuzzerTestOneInput**
  - Обёртка на Lua C API
- Структурирование случайных данных, **FuzzedDataProvider**
  - Обёртка на Lua C API
- Использование пользовательских мутаций, **LLVMFuzzerCustomMutator**
  - Обёртка на Lua C API
- Предоставление обратной связи из Lua в libFuzzer
  - **Тут интересно, расскажу далее**

Интеграция с Lua: схема обратной связи



Как инструментировать Lua для фаззера?

- Доработать `lua_sethook()` для трассировки:
  - Операторов ветвления (`OP_JMP`)
  - Операторов сравнения (`OP_EQ, OP_LT, OP_LE`)
- Доработать `lua_getinfo()` для извлечения информации о покрытии
- Передавать информацию из обработчика в Lua в C:
  - __sanitizer_weak_hook_memcmp
  - __sanitizer_cov_trace_cmp8
  - …

Покрытие кода Lua

- Инициализация счётчиков
  - __sanitizer_cov_pcs_init
  - __sanitizer_cov_8bit_counters_init
- В libFuzzer выполняется в compile-time
- Для Lua инициализация происходит в разделяемых библиотеках

Результаты: пример фаззинг-теста для Lua

```
local function TestOneInput(buf)
    local ok, res = pcall(msgpack.decode, buf)
    if ok == true then
        pcall(msgpack.encode, res)
    end
end

luzer.Fuzz(TestOneInput, nil, args)
```

Результаты

- Сделал модуль для фаззинга Lua приложений
- Нашел 3 бага во встроенных модулях Tarantool
- Код модуля для фаззинга Lua опубликую под свободной лицензией

Выводы

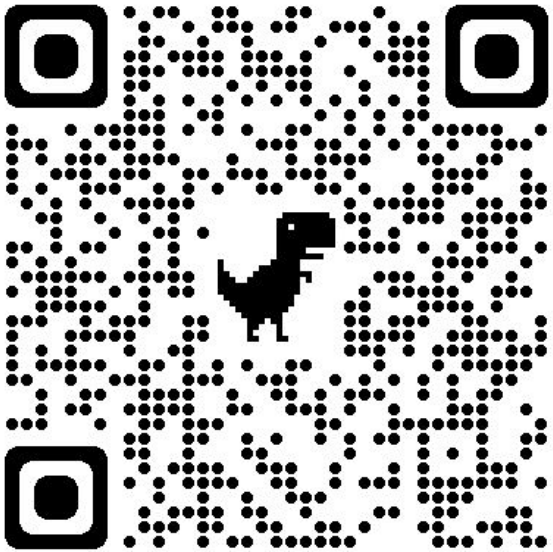
- Тесты на основе примеров это хорошо, а фаззинг их дополняет
- Сделать движок для своего языка на основе libFuzzer несложно
- Инварианты и тестовые оракулы привносят творчество в процесс тестирования
- Фаззинг и PBT это близкие подходы и эффективнее с обратной связью
- Попробуйте фаззинг и сделайте шаг навстречу **автоматическому** тестированию

Спасибо за внимание! Вопросы?

Сергей Бронников

Телеграм: @ligurio

Материалы и слайды: brnkv.ru/heisenbug2022



Что такое libFuzzer: пример фаззинг-таргета

```
int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
    if (size > 0 && data[0] == 'L')
        if (size > 1 && data[1] == 'U')
            if (size > 2 && data[2] == 'A')
            __builtin_trap();
    return 0;
}
```

Что такое libFuzzer: структурирование данных

```
FuzzedDataProvider provider(data, size);

auto val1 = provider.ConsumeIntegral<uint8_t>();
auto val2 = provider.ConsumeBool();

MyStruct val3 = {
    .my_int = provider.ConsumeIntegral<uint32_t>(),
    .my_double = provider.ConsumeFloatingPoint<double>(),
    .my_color = provider.ConsumeEnum<Color>(),
};
```

Интеграция с Lua: LLVMFuzzerTestOneInput

- Реализация с помощью Lua C API + `LLVMFuzzerRunDriver()`
- Пример:

```lua
local luzer = require("luzer")

local function TestOneInput(buf)

    -- ...

end

luzer.Fuzz(TestOneInput, nil, args)
```

Интеграция с Lua: FuzzedDataProvider

- Реализация с помощью Lua C API
- Пример:

```lua
local fdp = luzer.FuzzedDataProvider(buf)

local str1 = fdp:consume_string(1, 20)   -- случайная строка

local str2 = fdp:consume_boolean()       -- значение логического типа

local num1 = fdp:consume_number(1, 100) -- случайное число
```

Интеграция с Lua: LLVMFuzzerCustomMutator

Используем Lua C API

По умолчанию используются встроенные мутации

В Lua: `Fuzz(TestOneInput, CustomMutator)`

LLVMFuzzerCustomMutator подгружается через разделяемую библиотеку

Интеграция с Lua: схема мутаций