HotSpot continuous profiling JVM используя eBPF

Ибрагимов Эдуард

Software Engineer at Huawei

@eduardibr

eduardibr04@gmail.com

О чём будем говорить?

1) JVMTI профайлеры
2) Что такое eBPF?
3) Как профилировать с eBPF?
4) Боли при использовании
5) Выводы

Сценарий использования

1) Continuous profiling – профилирует прод и делает это долго

2) Есть проблема -> подключаемся и профилируем (*)

За чем следим:

- Потребление CPU
- Потребление памяти
- ...

Инструменты:

- JFR
- async-profiler
- Visual VM

(*) Но чтобы постоянно профилировать обычно требуется модификация контейнера

Сценарий использования

1) Continuous profiling – профилирует прод и делает это долго

Сценарий использования

1) Continuous profiling – профилирует прод и делает это долго

2) Есть проблема -> подключаемся и профилируем (*)

(*) Но чтобы постоянно профилировать обычно требуется модификация контейнера

Сценарий использования

1) Continuous profiling – профилирует прод и делает это долго

2) Есть проблема -> подключаемся и профилируем (*)

За чем следим:

- Потребление CPU
- Потребление памяти
- ...

(*) Но чтобы постоянно профилировать обычно требуется модификация контейнера

Сценарий использования

1) Continuous profiling – профилирует прод и делает это долго

2) Есть проблема -> подключаемся и профилируем (*)

За чем следим:

- Потребление CPU
- Потребление памяти
- ...

Инструменты:

- JFR
- async-profiler
- Visual VM

(*) Но чтобы постоянно профилировать обычно требуется модификация контейнера

Async-profiler

Плюсы:

1) Нет проблемы safepoint-bias

2) Малый оверхед

3) Открытый код

4) Развивается долгие годы

Async-profiler

Плюсы:

1) Нет проблемы safepoint-bias

2) Малый оверхед

3) Открытый код

4) Развивается долгие годы

Является стандартом для профилирования JVM

Async-profiler

Плюсы:

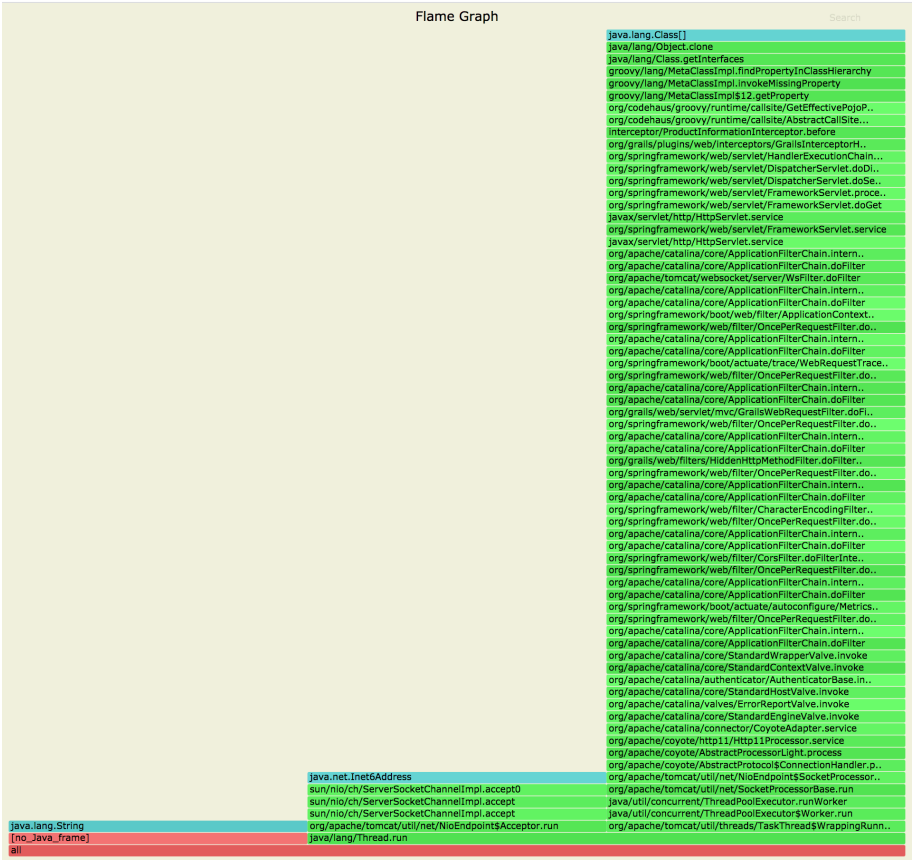
1) Нет проблемы safepoint-bias

2) Малый оверхед

3) Открытый код

4) Развивается долгие годы

Является стандартом для профилирования JVM



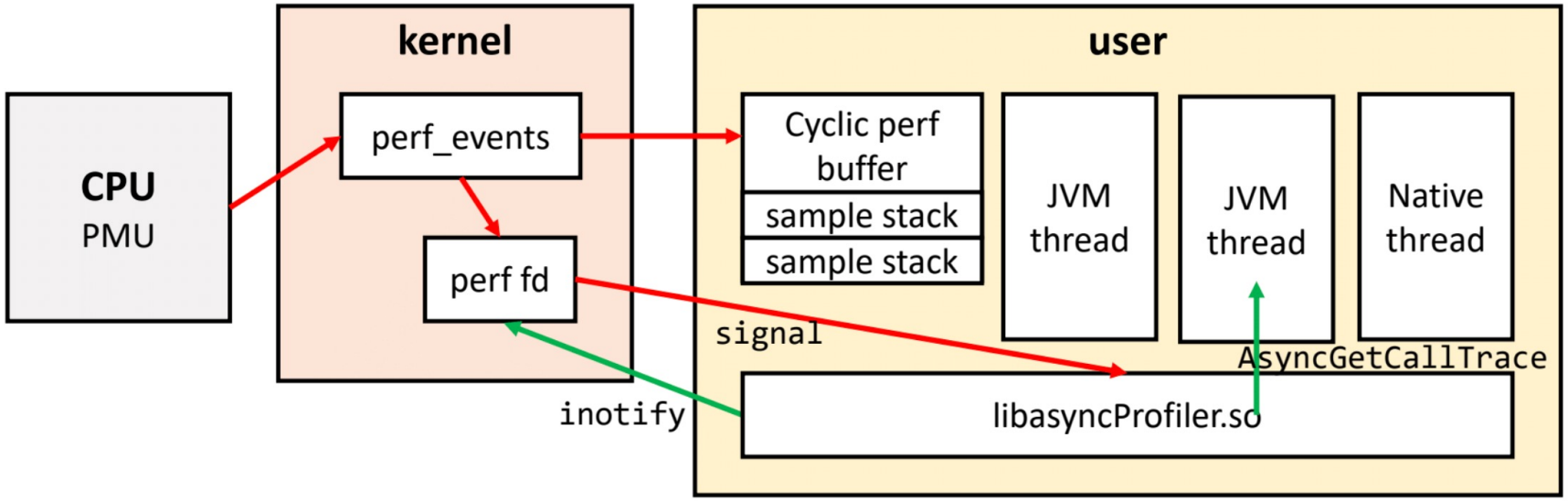
Пример флеймграфа

Как он работает?

Основан на вызове функции AsyncGetCallTrace из JVMTI

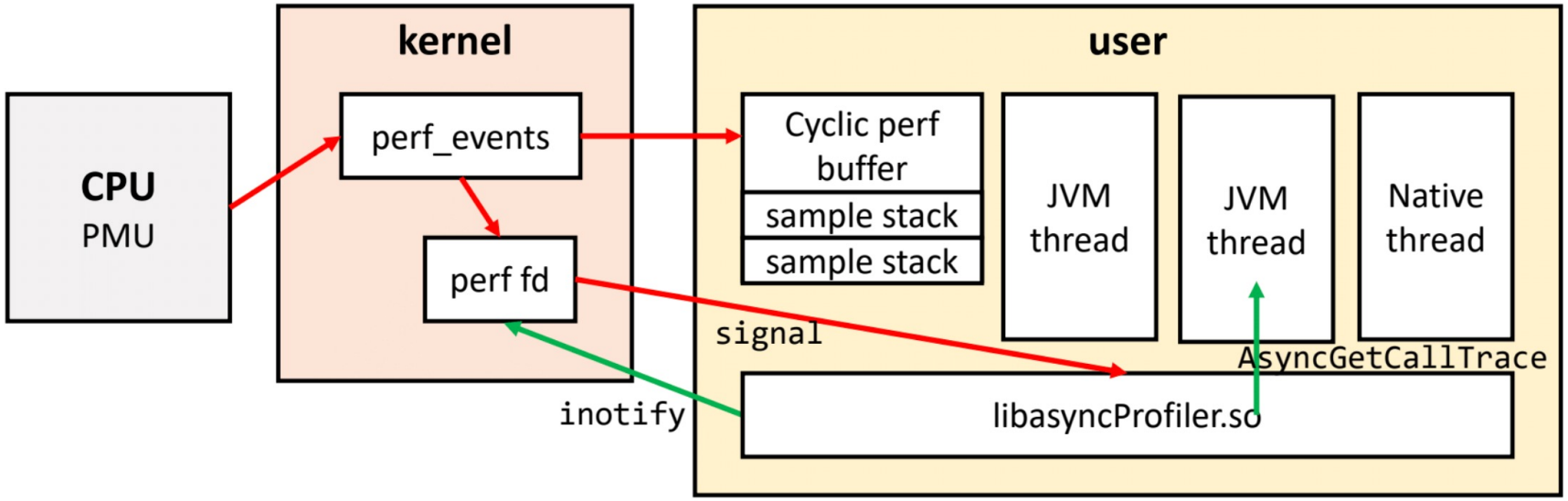
Как он работает?

Основан на вызове функции AsyncGetCallTrace из JVMTI



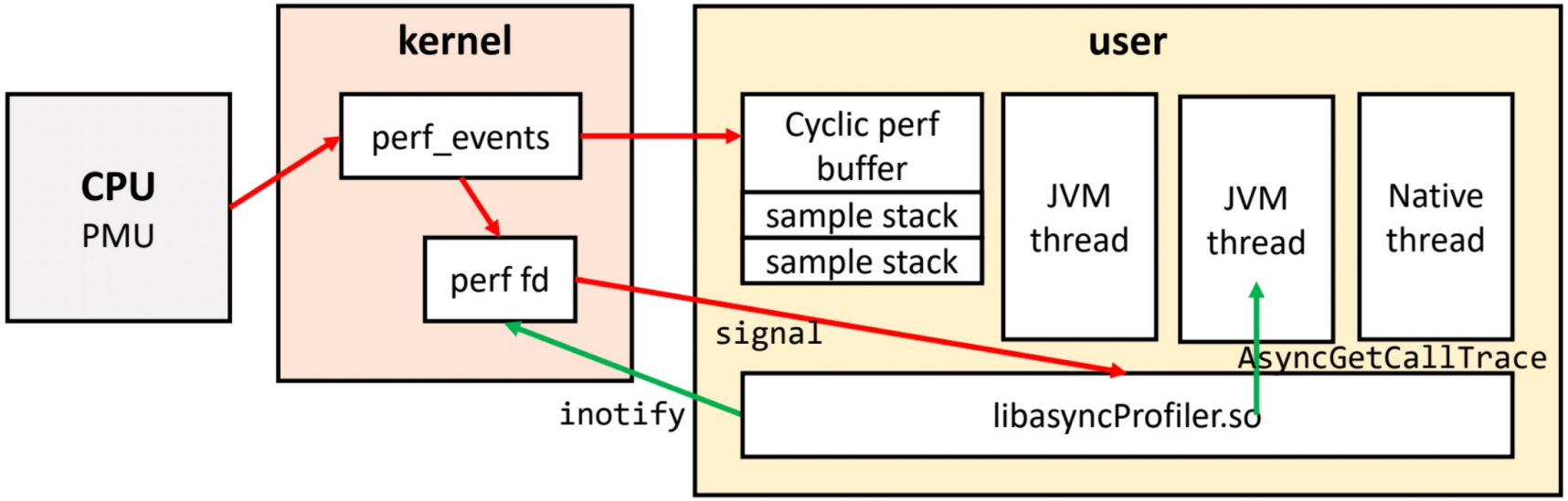
Как он работает?

Основан на вызове функции AsyncGetCallTrace из JVMTI



Как он работает?

Основан на вызове функции AsyncGetCallTrace из JVMTI



Какие есть недостатки?

Все профилировщики попадают внутрь процесса, используя JVMTI

Какие есть недостатки?

Все профилировщики попадают внутрь процесса, используя JVMTI

Поэтому:

1) Нужно иметь доступ к командной строке контейнера

2) Профилировщик внутри адресного пространства процесса

3) Чтобы всё работало нужен запуск контейнера с `--privileged`

Вопрос-Ответ

Вопрос: Можно-ли как-то профилировать без вмешательства в процесс?

Вопрос-Ответ

Вопрос: Можно-ли как-то профилировать без вмешательства в процесс?

Ответ: Конечно, используем eBPF!

Что за eBPF?

Что такое eBPF

Позволяет выполнять программы внутри kernel space’а

Что такое eBPF

Позволяет выполнять программы внутри kernel space’а

Появилась данная технология в 1990 году для фильтрации сетевого трафика

Что такое eBPF

Позволяет выполнять программы внутри kernel space’а

Появилась данная технология в 1990 году для фильтрации сетевого трафика

В 2014 году вышло крупное обновление:

- JIT
- Расширен функционал

Почему eBPF

Изначально была идея написать свою AsyncGetCallTrace на чистом eBPF

Почему eBPF

Изначально была идея написать свою AsyncGetCallTrace на чистом eBPF

Мы посмотрели на исходный код данной функции и нам показалось, что там не так много кода...

Почему eBPF

Изначально была идея написать свою AsyncGetCallTrace на чистом eBPF

Мы посмотрели на исходный код данной функции и нам показалось, что там не так много кода...

Тогда бы мы могли делать всю работу внутри kernel'а и возвращать стектрейс на сторону user-space.

Наивный был конечно подход, но об этом позже...

Что умеет eBPF

Основные сценарии использования:

1) Наблюдение за сетью

2) Управление сетевыми коммуникациями

3) Перехват событий ядра

4) ...

Что умеет eBPF

Основные сценарии использования:

1) Наблюдение за сетью

2) Управление сетевыми коммуникациями

3) Перехват событий ядра

4) ...

Нам интересен вариант 3

Какие события бывают?

Можно выполнять свой код, когда выполняется событие внутри операционной системы.

Какие события бывают?

Можно выполнять свой код, когда выполняется событие внутри операционной системы.

1) Kernel Probe
2) User Probe
3) Perf-event
4) eBPF-iterators
5) ...

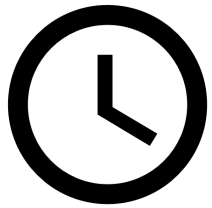
Какие события бывают?

Можно выполнять свой код, когда выполняется событие внутри операционной системы.

1) Kernel Probe
2) User Probe
3) Perf-event
4) eBPF-iterators
5) ...

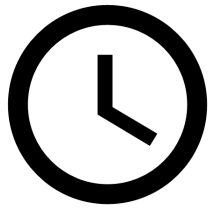
Из данного списка нам интересны kernel probe’ы и perf-event’ы

Perf event



sys_perf_event_open - ставит таймер, который вызывается через равные промежутки времени, если интересующий нас поток исполняется на ядре.

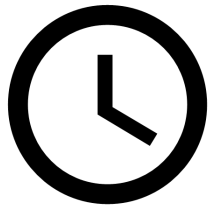
Perf event



sys_perf_event_open - ставит таймер, который вызывается через равные промежутки времени, если интересующий нас поток исполняется на ядре.

Многие профилировщики используют его под капотом

Perf event

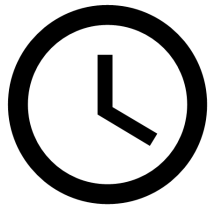


sys_perf_event_open - ставит таймер, который вызывается через равные промежутки времени, если интересующий нас поток исполняется на ядре.

Многие профилировщики используют его под капотом

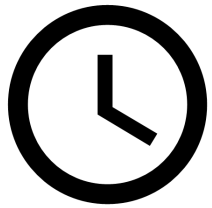
Функция по данному таймеру может посылать сигнал потоку (async-profiler) или вызывать ebpf код

Perf event



А вот как это выглядит в коде:

Perf event



А вот как это выглядит в коде:

```
#includes ...

// load programm into kernel
programFd = bpf(BPF_PROG_LOAD, &compiled_code);

// open perf event
fd = perf_event_open(some_other_params);

// link ebpf prog with perf_event
ioctl(fd, PERF_EVENT_IOC_SET_BPF, programFd);
```

Kernel-probe

Позволяет прервать исполнение kenel-кода в определенной точке (обычно начало функции) и передать управление ebpf-программе.

Kernel-probe

Позволяет прервать исполнение kenel-кода в определенной точке (обычно начало функции) и передать управление ebpf-программе.

Код, который будет срабатывать, при вызове функции `open()`

Kernel-probe

Позволяет прервать исполнение kenel-кода в определенной точке (обычно начало функции) и передать управление ebpf-программе.

Код, который будет срабатывать, при вызове функции open()

```
SEC("kbrobe/do_sys_openat")
int kbrobe__sys_open(struct pt_regs *ctx) {
    ...
}
```

Kernel-probe

Позволяет прервать исполнение kenel-кода в определенной точке (обычно начало функции) и передать управление ebpf-программе.

Код, который будет срабатывать, при вызове функции open()

```
SEC("kbrobe/do_sys_openat")
int kbrobe__sys_open(struct pt_regs *ctx) {
    ...
}
```

Из структуры pt_regs можно достать состояние регистров процессора

Как нам поможет eBPF?

Изначальный план:

1) Парсить коллстек приложения в kernel-space
2) Сопоставлять имена функций с их адресами
3) Отправлять коллстек в user-space

Как нам поможет eBPF?

Изначальный план:

1) Парсить коллстек приложения в kernel-space
2) Сопоставлять имена функций с их адресами
3) Отправлять коллстек в user-space

Для отправки данных в user-space есть спец. структуры данных

Как нам поможет eBPF?

Изначальный план:

1) Парсить коллстек приложения в kernel-space
2) Сопоставлять имена функций с их адресами
3) Отправлять коллстек в user-space

Для отправки данных в user-space есть спец. структуры данных

+ JVMTI устроена очень удобно для нашего применения и выставляет необходимую нам информацию наружу

Про структуры данных

Нужно заранее определить eBPF maps - объекты ядра, с помощью которых можно передавать данные

Про структуры данных

Нужно заранее определить eBPF maps - объекты ядра, с помощью которых можно передавать данные

Примеры структур данных:

1) HashMap
2) Per CPU array
3) RingBuffer
4) ...

Инициализация происходит до загрузки программы

Про структуры данных

Нужно заранее определить eBPF maps - объекты ядра, с помощью которых можно передавать данные

Примеры структур данных:

1) HashMap
2) Per CPU array
3) RingBuffer
4) ...

Для взаимодействия с этими структурами есть набор функций – BPF helpers.

Инициализация происходит до загрузки программы

Hash Map

Мапа как и везде - пара ключ и значение

Hash Map

Мапа как и везде - пара ключ и значение

Ключ - это blob, никаких компораторов, сравниваем байты

Hash Map

Мапа как и везде - пара ключ и значение

Ключ - это blob, никаких компораторов, сравниваем байты

```c
// bpf map declaration
struct bpf_map_def SEC("maps") some_states = {
    .type = BPF_MAP_TYPE_HASH,
    .key_size = sizeof(int),
    .value_size = sizeof(struct some_data),
    .max_entries = 42,
};
```

SEC(”maps”) - указывает секцию, в которую попадёт структура

Ring Buffer

Кольцевой буффер - это массив с двумя указателями на начало и конец.

Ring Buffer

Кольцевой буффер - это массив с двумя указателями на начало и конец.

Создаётся как HashMap, но `.type = BPF_MAP_TYPE_HASH`

Ring Buffer

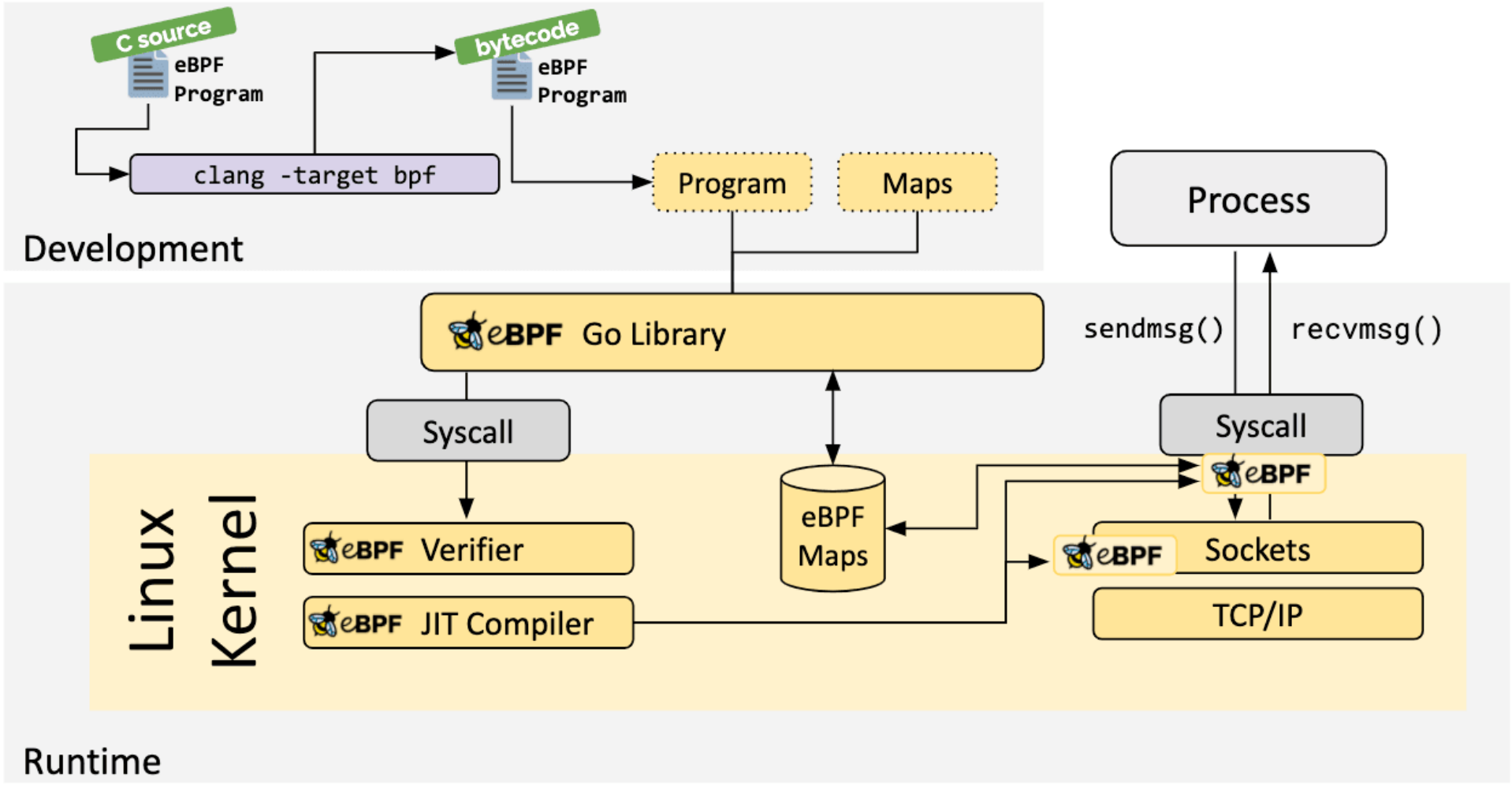
Кольцевой буффер - это массив с двумя указателями на начало и конец.

Создаётся как HashMap, но `.type = BPF_MAP_TYPE_HASH`

Писать и читать из него можно асинхронно, поэтому он подходит для пересылки данных из kernel-space в user-space (*)

(*) Спойлер

Как загружается eBPF



Source: eBPF site

Как запустить eBPF?

1) Пишем код на псевдо-C

Как запустить eBPF?

1) Пишем код на псевдо-C

```c
#include <some_stuff.h>

// map declaration
struct bpf_map_def SEC("maps") some_states = {
    ...
};

// prog code
SEC("kbrobe/do_sys_openat")
int kbrobe__sys_open(struct pt_regs *ctx) {
    ...
}
```

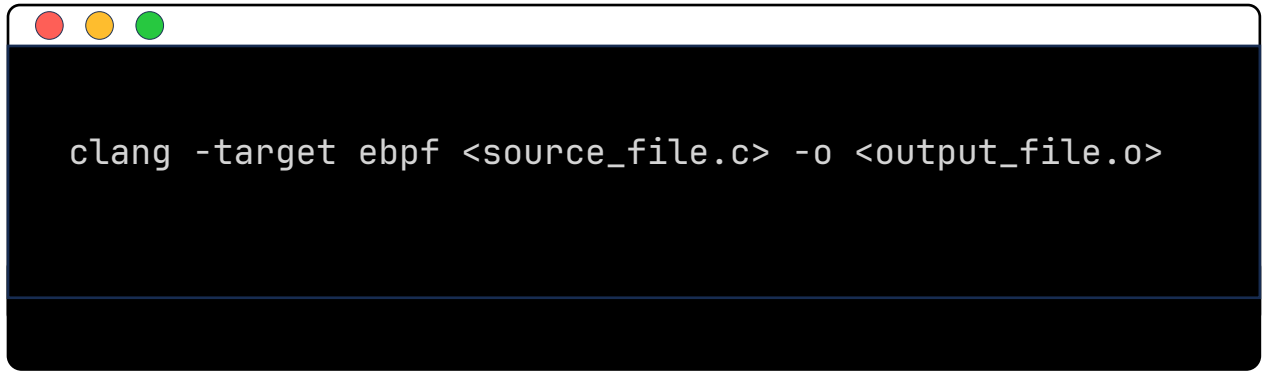
Как запустить eBPF?

2) Компилируем код

Как запустить eBPF?

2) Компилируем код

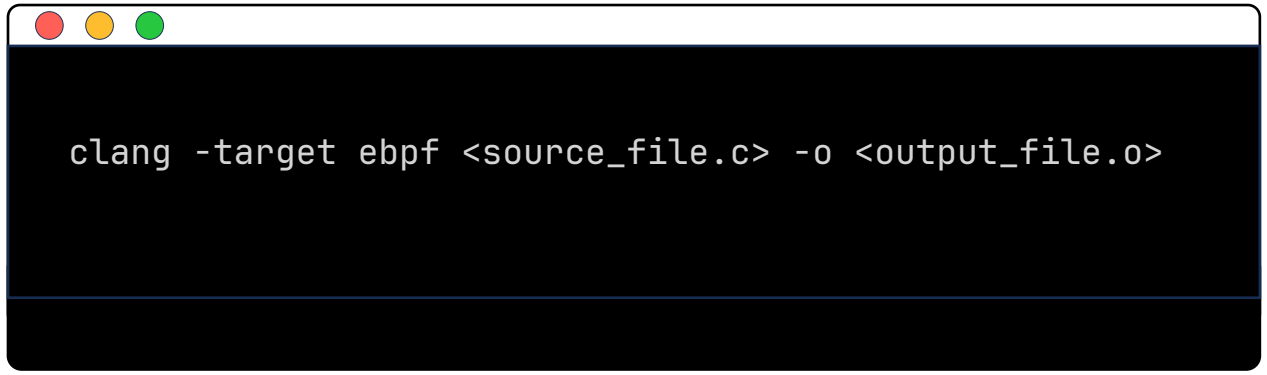
Есть отдельный бекенд clang с таргетом под ebpf



Как запустить eBPF?

2) Компилируем код

Есть отдельный бекенд clang с таргетом под ebpf



На выходе имеем elf файл с расширением *.o

Как запустить eBPF?

3) Запускаем код внутри ядра

Как запустить eBPF?

3) Запускаем код внутри ядра

Так как ядро linux постоянно меняется, то невозможно своими руками следить за offset’ами структур ядра

Как запустить eBPF?

3) Запускаем код внутри ядра

Так как ядро linux постоянно меняется, то невозможно своими руками следить за offset’ами структур ядра

Чтобы облегчить жизнь есть ряд библиотек для запуска eBPF:

А что с профилированием?

Напомним, что изначальная цель – переписать логику AsyncGetCallTrace на чистом eBPF

А что с профилированием?

Напомним, что изначальная цель - переписать логику AsyncGetCallTrace на чистом eBPF

В первом приближении такая задача не казалась чем-то очень сложным

А что с профилированием?

Напомним, что изначальная цель - переписать логику AsyncGetCallTrace на чистом eBPF

В первом приближении такая задача не казалась чем-то очень сложным

Наивный подход, об этом расскажем попозже...

А что с профилированием?

Далее, зная значения stack base и base pointer мы хотели проходится по списку функций и восстановить коллстек

А что с профилированием?

Далее, зная значения stack base и base pointer мы хотели проходится по списку функций и восстановить коллстек

Оказалось, что у eBPF есть значительные ограничения по размеру и сложности выполняемой программы

А что с профилированием?

Далее, зная значения stack base и base pointer мы хотели проходится по списку функций и восстановить коллстек

Оказалось, что у eBPF есть значительные ограничения по размеру и сложности выполняемой программы

Такие ограничения необходимы, чтобы не проседала производительность основной программы

Ограничения eBPF

За то, чтобы eBPF программа работала правильно отвечает верификатор

Ограничения eBPF

За то, чтобы eBPF программа работала правильно отвечает верификатор

Он имеет следующие ограничения:

1) Стек < 512 байт
2) Нет динамической памяти
3) Максимальный размер программы – 4096 инструкций
4) ...

Ограничения eBPF

За то, чтобы eBPF программа работала правильно отвечает верификатор

Он имеет следующие ограничения:

1) Стек < 512 байт
2) Нет динамической памяти
3) Максимальный размер программы – 4096 инструкций
4) ...

- Ну окей, а нам-то что от этих ограничений?

Злобный верификатор

Программа скомпилируется, но её не пропустит верификатор

Злобный верификатор

Программа скомпилируется, но её не пропустит верификатор

```
; void *data = bpf_ringbuf_reserve(&some_data, sp, 0); rename to
allocated buffer
81: (18) r1 = 0xffff89c540f27800
83: R0=inv(id=0) R1_w=map_ptr(id=0,off=0,ks=0,vs=0,imm=0)
83: (bf) r2 = r7
84: R0=inv(id=0) R1_w=map_ptr(id=0,off=0,ks=0,vs=0,imm=0)
84: (b7) r3 = 0
85: R0=inv(id=0) R1_w=map_ptr(id=0,off=0,ks=0,vs=0,imm=0)
85: (85) call bpf_ringbuf_reserve#131
R2 is not a known constant'
```

Злобный верификатор

Программа скомпилируется, но её не пропустит верификатор

```
; void *data = bpf_ringbuf_reserve(&some_data, sp, 0); rename to
allocated buffer
81: (18) r1 = 0xffff89c540f27800
83: R0=inv(id=0) R1_w=map_ptr(id=0,off=0,ks=0,vs=0,imm=0)
83: (bf) r2 = r7
84: R0=inv(id=0) R1_w=map_ptr(id=0,off=0,ks=0,vs=0,imm=0)
84: (b7) r3 = 0
85: R0=inv(id=0) R1_w=map_ptr(id=0,off=0,ks=0,vs=0,imm=0)
85: (85) call bpf_ringbuf_reserve#131
R2 is not a known constant'
```

О видах ошибок и испытанной боли мы еще поговорим..

Как пришлось делать?

Из-за ограничений, которые наложил верификатор нам пришлось пересмотреть схему работы

Как пришлось делать?

Из-за ограничений, которые наложил верификатор нам пришлось пересмотреть схему работы

Будем обрабатывать стек вызываемых функций в user-space

Как пришлось делать?

Из-за ограничений, которые наложил верификатор нам пришлось пересмотреть схему работы

Будем обрабатывать стек вызываемых функций в user-space

Новая схема работы:

1) Вычитать коллстек в бинарном виде (с stack_base до SP)
2) Переслать его в user-space
3) Провести сбор доп. информации не в ядре

Как пришлось делать?

В user-space мы получаем бинарное представление коллстека

Как пришлось делать?

В user-space мы получаем бинарное представление коллстека

Имен функций в таком представлении не имеется, поэтому нам пришлось их дочитывать из памяти процесса (*)

(*) путь в linux: /proc/{pid}/memory

Как пришлось делать?

В user-space мы получаем бинарное представление коллстека

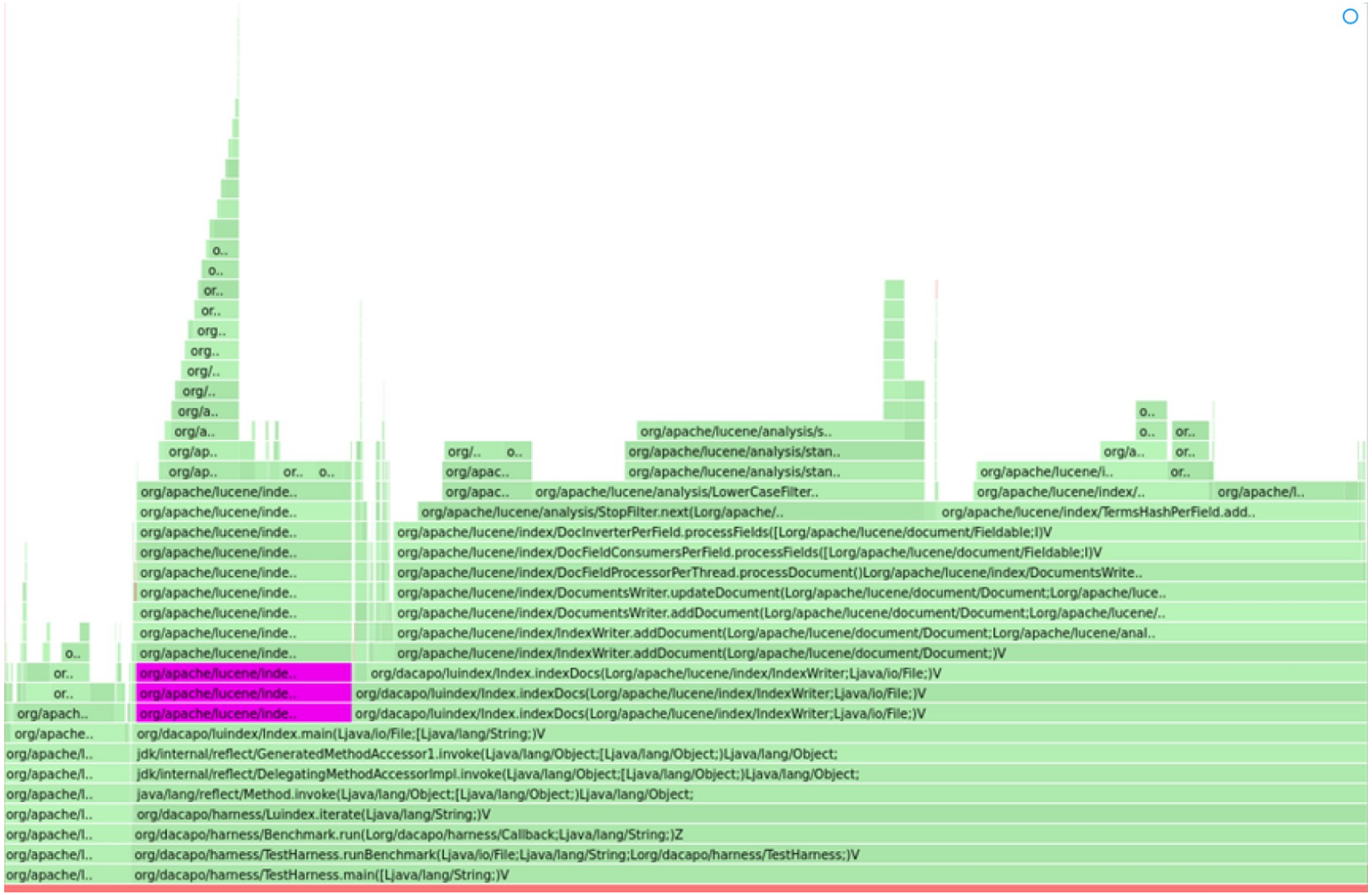
Имен функций в таком представлении не имеется, поэтому нам пришлось их дочитывать из памяти процесса (*)

Дочитывать приходится еще и другую информацию, чтобы различать компилируемые и интерпретируемые фреймы

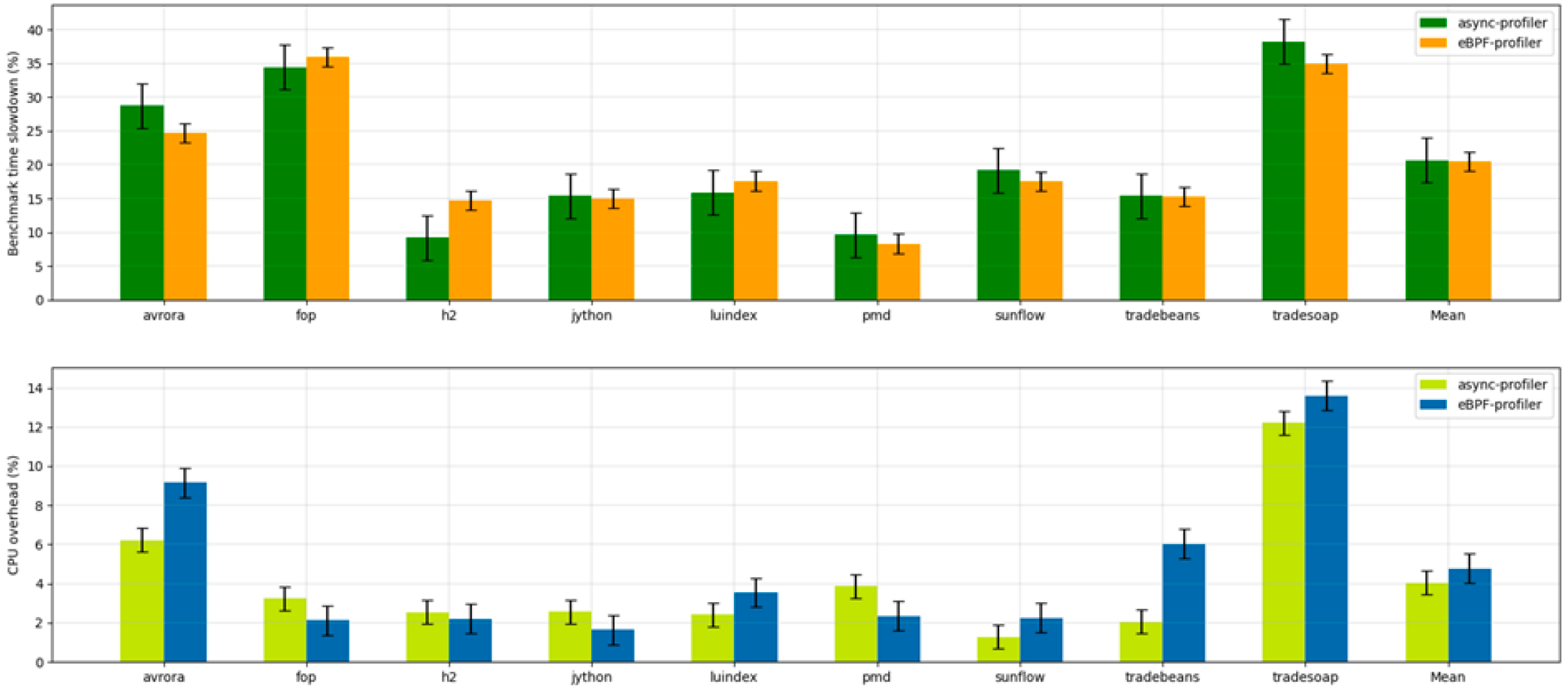
(*) путь в linux: /proc/{pid}/memory

Если коротко, то

Что получаем в итоге?



Что с перформансом?



(*) частота семплирования – 50 гц, бенчмарк - [dacapo]

Про боль и страдания

Боль даже от банальных вещей

Допустим нам нужно сравнить две строки

Боль даже от банальных вещей

Допустим нам нужно сравнить две строки

В обычной программе на C мы бы делали следующее:

Боль даже от банальных вещей

Допустим нам нужно сравнить две строки

В обычной программе на C мы бы делали следующее:

```c
#include <string.h>

char *haystack = "Hello, world!"
char *needle = "wor";

// world!
char result = strstr()
```

Боль даже от банальных вещей

Допустим нам нужно сравнить две строки

В обычной программе на C мы бы делали следующее:

```c
#include <string.h>

char *haystack = "Hello, world!"
char *needle = "wor";

// world!
char result = strstr()
```

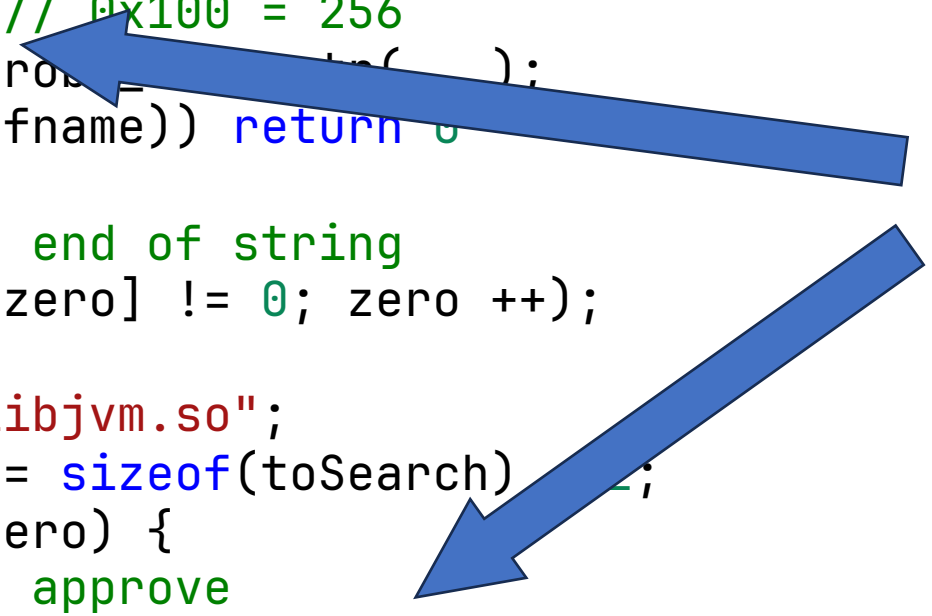
Просто добавляем

библиотеку <string.h>

и дело сделано

Боль даже от банальных вещей

```
int kprobe__sys_open(struct pt_regs *ctx) {
    char fname[0x100]; // 0x100 = 256
    size_t read = bpf_prob
    if (read == sizeof(fname)) return 0
    size_t zero = 0;
    //searching for the end of string
    for (;zero < fname[zero] != 0; zero ++);

    char toSearch = "/libjvm.so";
    size_t toSearchIdx = sizeof(toSearch)
    if (toSearchIdx < zero) {
        // for verifier approve
        size_t idx = (zero - 1) & 0xff;
        size_t i = 0;
        ...
    }
}
```



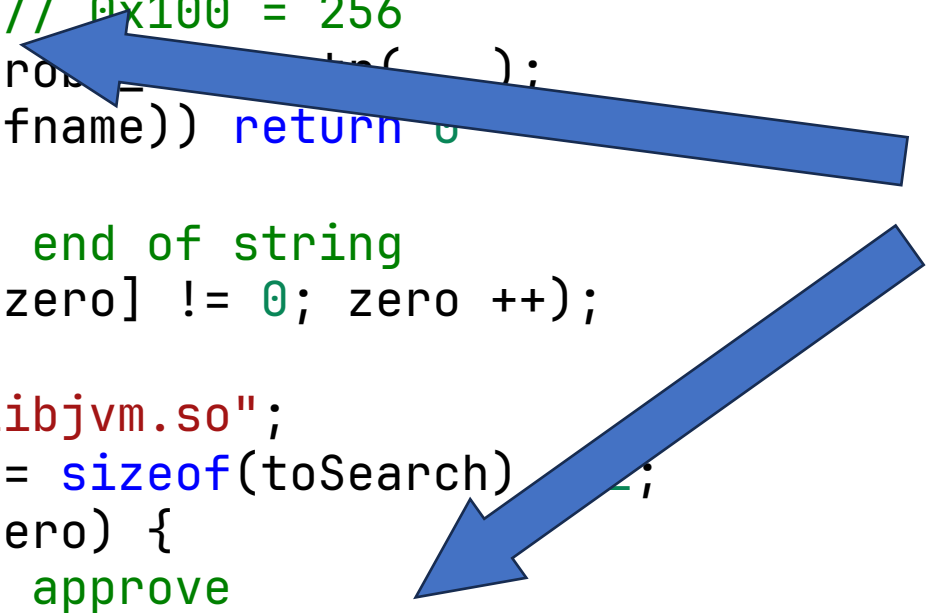
Такое нужно,

чтобы не ругался

верифаер

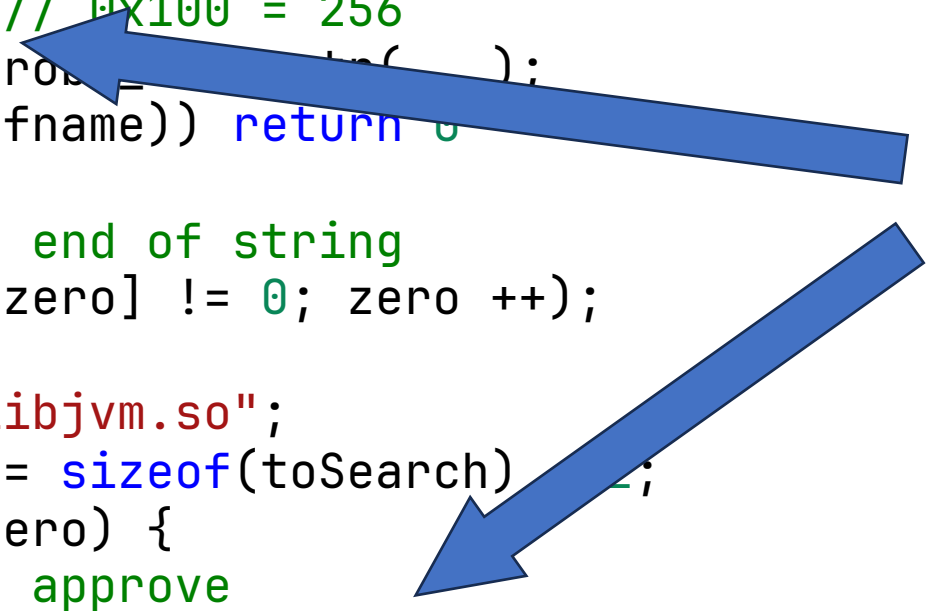
Боль даже от банальных вещей

```
int kprobe__sys_open(struct pt_regs *ctx) {
    char fname[0x100]; // 0x100 = 256
    size_t read = bpf_pro
    if (read == sizeof(fname)) return 0
    size_t zero = 0;
    //searching for the end of string
    for (;zero < fname[zero] != 0; zero ++);

    char toSearch = "/libjvm.so";
    size_t toSearchIdx = sizeof(toSearch)
    if (toSearchIdx < zero) {
        // for verifier approve
        size_t idx = (zero - 1) & 0xff;
        size_t i = 0;
        ...
    }
}
```



Боль даже от банальных вещей

```
int kprobe__sys_open(struct pt_regs *ctx) {
    char fname[0x100]; // 0x100 = 256
    size_t read = bpf_prob
    if (read == sizeof(fname)) return 0
    size_t zero = 0;
    //searching for the end of string
    for (;zero < fname[zero] != 0; zero ++);

    char toSearch = "/libjvm.so";
    size_t toSearchIdx = sizeof(toSearch)
    if (toSearchIdx < zero) {
        // for verifier approve
        size_t idx = (zero - 1) & 0xff;
        size_t i = 0;
        ...
    }
}
```



Такое нужно,

чтобы не ругался

верифаер

Как выделять память?

Нам нужно выделять память под коллстеки разного размера

Как выделять память?

Нам нужно выделять память под коллстеки разного размера

Было бы хорошо выделять память динамически, чтобы правильно использовать память

Как выделять память?

Нам нужно выделять память под коллстеки разного размера

Было бы хорошо выделять память динамически, чтобы правильно использовать память

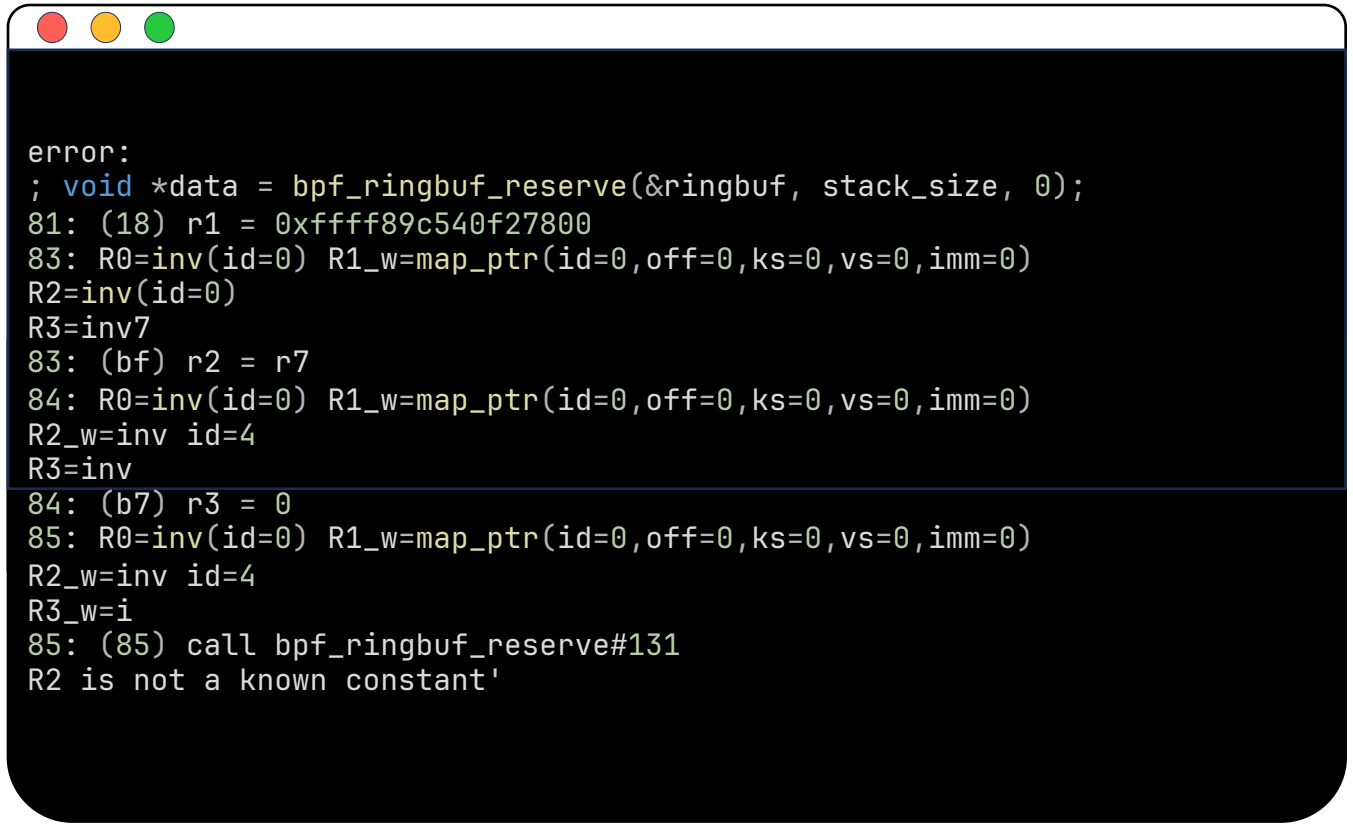
```
void get_cool_callstack(struct bpf_perf_event_data *ctx) {
    u64 stack_size = currThreadState->bp - SP(ctx);
    if (stack_size > 0) {
        void *data = bpf_ringbuf_reserve(&ringbuf, stack_size, 0);
    }
}
```

Как выделять память?

Запускаем этого код для версии ядра ниже 5.19

Как выделять память?

Запускаем этого код для версии ядра ниже 5.19



```
error:
; void *data = bpf_ringbuf_reserve(&ringbuf, stack_size, 0);
81: (18) r1 = 0xffff89c540f27800
83: R0=inv(id=0) R1_w=map_ptr(id=0,off=0,ks=0,vs=0,imm=0)
R2=inv(id=0)
R3=inv7
83: (bf) r2 = r7
84: R0=inv(id=0) R1_w=map_ptr(id=0,off=0,ks=0,vs=0,imm=0)
R2_w=inv id=4
R3=inv
84: (b7) r3 = 0
85: R0=inv(id=0) R1_w=map_ptr(id=0,off=0,ks=0,vs=0,imm=0)
R2_w=inv id=4
R3_w=i
85: (85) call bpf_ringbuf_reserve#131
R2 is not a known constant'
```

Как выделять память?

Запускаем этого код для версии ядра ниже 5.19

```
error:
; void *data = bpf_ringbuf_reserve(&ringbuf, stack_size, 0);
81: (18) r1 = 0xffff89c540f27800
83: R0=inv(id=0) R1_w=map_ptr(id=0,off=0,ks=0,vs=0,imm=0)
R2=inv(id=0)
R3=inv7
83: (bf) r2 = r7
84: R0=inv(id=0) R1_w=map_ptr(id=0,off=0,ks=0,vs=0,imm=0)
R2_w=inv id=4
R3=inv
84: (b7) r3 = 0
85: R0=inv(id=0) R1_w=map_ptr(id=0,off=0,ks=0,vs=0,imm=0)
R2_w=inv id=4
R3_w=i
85: (85) call bpf_ringbuf_reserve#131
R2 is not a known constant'
```

Как выделять память?

До версии ядра 5.19 нужной функции не было

Как выделять память?

До версии ядра 5.19 нужной функции не было

`BPF_FUNC_ringbuf_reserve()`	5.8		`457f44363a88`
`BPF_FUNC_ringbuf_reserve_dynptr()`	5.19		`bc34dee65a65`

Как выделять память?

До версии ядра 5.19 нужной функции не было

`BPF_FUNC_ringbuf_reserve()`	5.8		`457f44363a88`
`BPF_FUNC_ringbuf_reserve_dynptr()`	5.19		`bc34dee65a65`

Так как нужно поддерживать не только свежие ядра, то

1) нужно писать код, который не отвергнет верификатор
2) память всё таки хочется расходовать оптимально

Ссылка на патч

Как выделять память?

Чтобы это обойти приходилось писать следующее:

Как выделять память?

Чтобы это обойти приходилось писать следующее:

```
void get_cool_callstack(struct bpf_perf_event_data *ctx) {
    u64 stack_size = thread_state->bp - SP(ctx);
    int sizes[] = {256, 1024, 4096, 8192, ... };
    for (size_t i = 0; i < sizes.length; ++i) {
        if (stack_size < sizes[i])
           void *data = bpf_ringbuf_reserve(&ringbuf, sizes[i]);
           ...
    }
}
```

Боль с паддингом структур

Работа со структурами данных тоже делается с болью

Боль с паддингом структур

Работа со структурами данных тоже делается с болью

```
struct test_padding {
    bool a; // 1 byte
    int b; // 4 bytes
};
```

Боль с паддингом структур

Работа со структурами данных тоже делается с болью

```
struct test_padding {
    bool a; // 1 byte
    int b; // 4 bytes
};
```

Простенькая структурка

Боль с паддингом структур

Работа со структурами данных тоже делается с болью

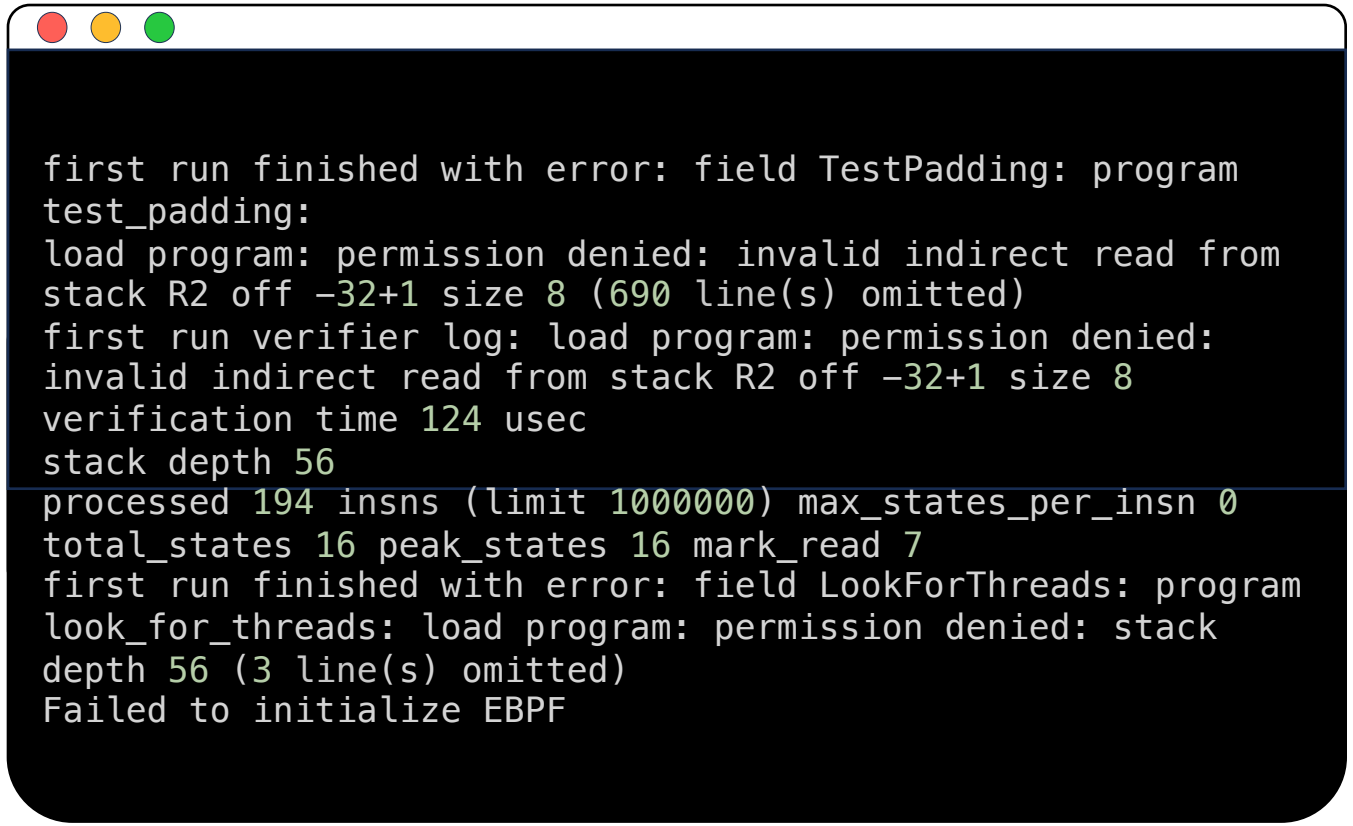
```
struct test_padding {
    bool a; // 1 byte
    int b; // 4 bytes
};
```

Простенькая структурка

```
SEC("...")
void test_padding(...) {
    test_padding t;
    t.a = false;
    t.b = 42;
    size_t structSz = sizeof(test_padding);
    bpf_ringbuf_output(&map, &t, structSz);
}
```

Боль с паддингом структур

Запускаем, и...



```
first run finished with error: field TestPadding: program
test_padding:
load program: permission denied: invalid indirect read from
stack R2 off -32+1 size 8 (690 line(s) omitted)
first run verifier log: load program: permission denied:
invalid indirect read from stack R2 off -32+1 size 8
verification time 124 usec
stack depth 56
processed 194 insns (limit 1000000) max_states_per_insn 0
total_states 16 peak_states 16 mark_read 7
first run finished with error: field LookForThreads: program
look_for_threads: load program: permission denied: stack
depth 56 (3 line(s) omitted)
Failed to initialize EBPF
```

Боль с паддингом структур

Запускаем, и...

```
first run finished with error: field TestPadding: program
test_padding:
load program: permission denied: invalid indirect read from
stack R2 off -32+1 size 8 (690 line(s) omitted)
first run verifier log: load program: permission denied:
invalid indirect read from stack R2 off -32+1 size 8
verification time 124 usec
stack depth 56
processed 194 insns (limit 1000000) max_states_per_insn 0
total_states 16 peak_states 16 mark_read 7
first run finished with error: field LookForThreads: program
look_for_threads: load program: permission denied: stack
depth 56 (3 line(s) omitted)
Failed to initialize EBPF
```

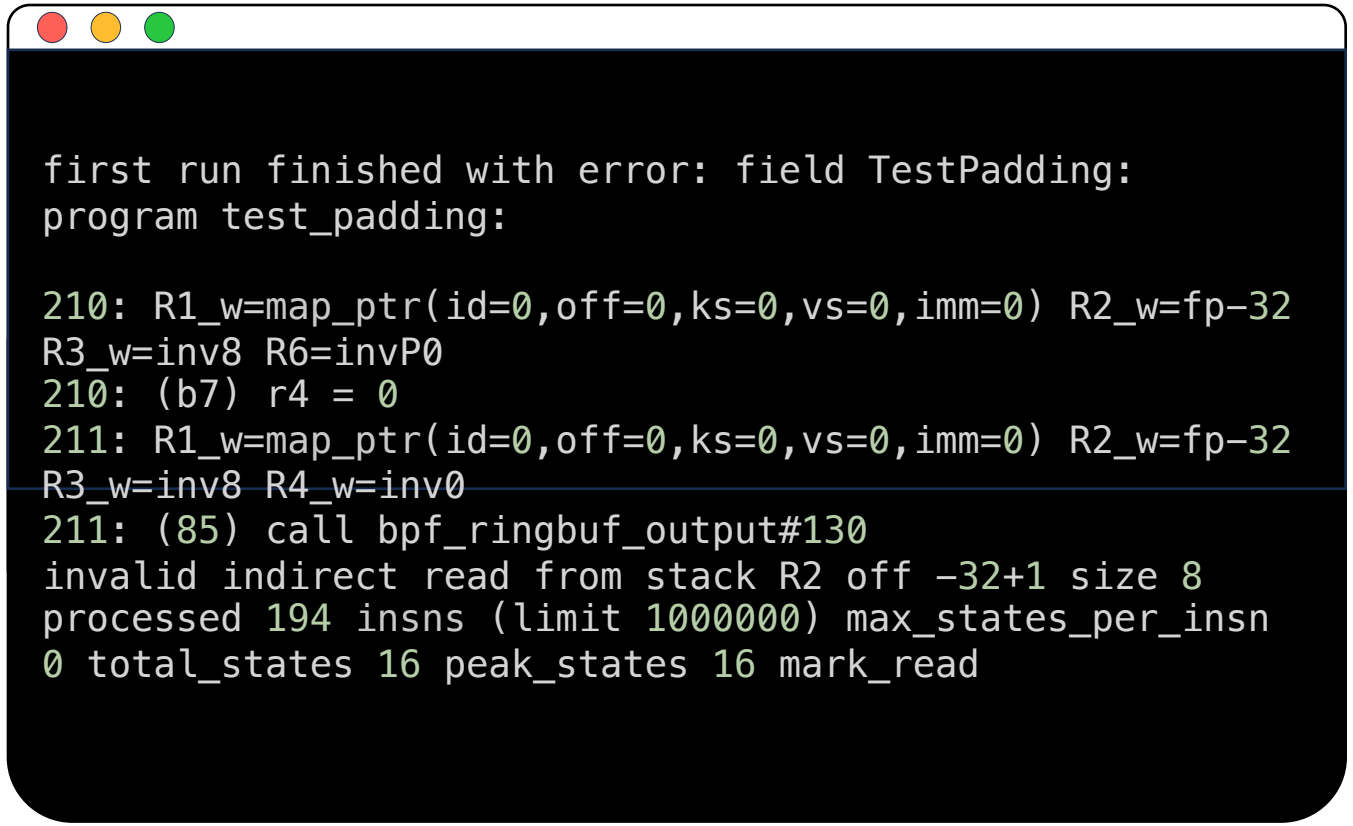
Но код-то простой

Боль с паддингом структур

Запускаем еще раз с ключом для получения подробной инфы

Боль с паддингом структур

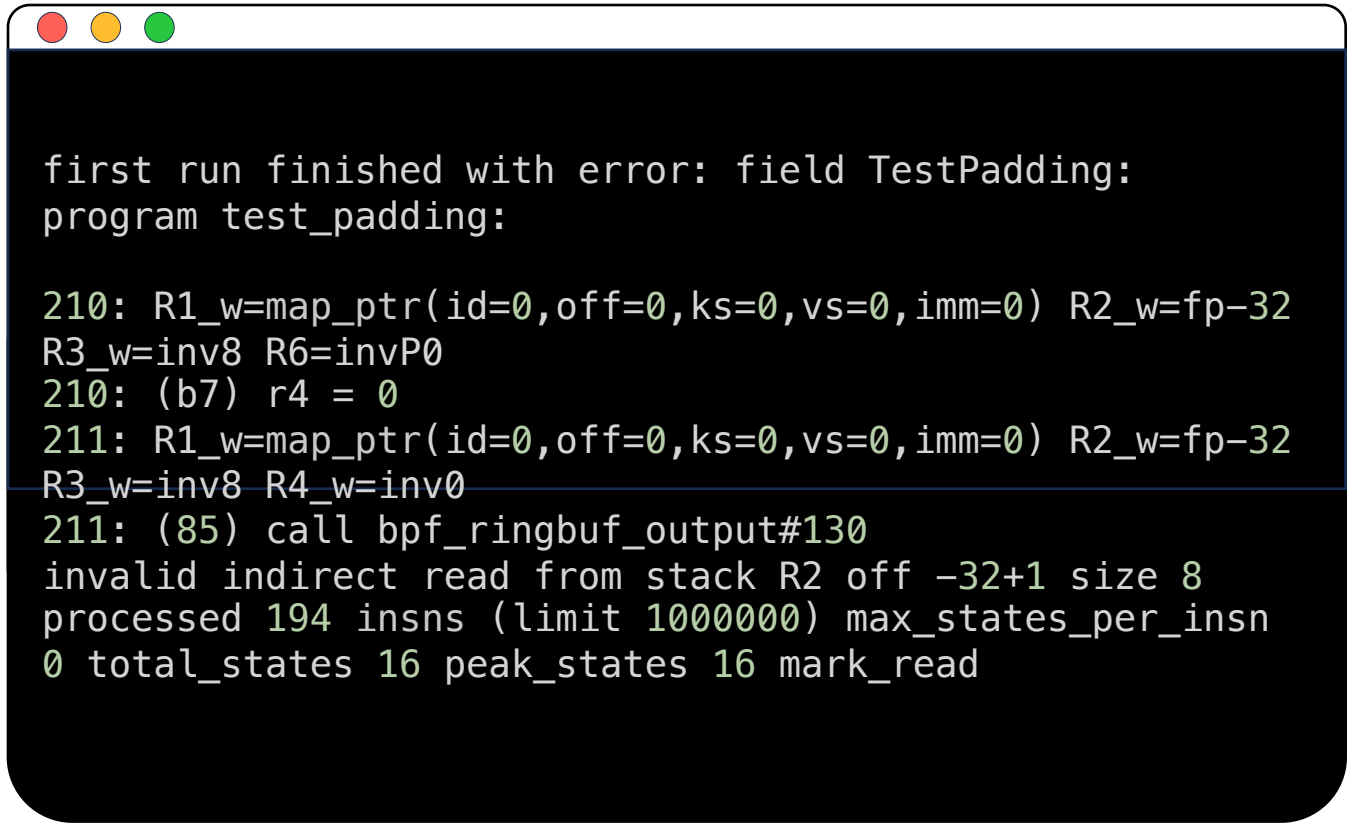
Запускаем еще раз с ключом для получения подробной инфы



Видно, что есть проблема в чтении из памяти

Боль с паддингом структур

Запускаем еще раз с ключом для получения подробной инфы



```
first run finished with error: field TestPadding:
program test_padding:

210: R1_w=map_ptr(id=0,off=0,ks=0,vs=0,imm=0) R2_w=fp-32
R3_w=inv8 R6=invP0
210: (b7) r4 = 0
211: R1_w=map_ptr(id=0,off=0,ks=0,vs=0,imm=0) R2_w=fp-32
R3_w=inv8 R4_w=inv0
211: (85) call bpf_ringbuf_output#130
invalid indirect read from stack R2 off -32+1 size 8
processed 194 insns (limit 1000000) max_states_per_insn
0 total_states 16 peak_states 16 mark_read
```

Видно, что есть проблема в чтении из памяти

Почему так может быть?

Боль с паддингом структур

На самом деле всё дело в выравнивании структур по 8 байт

Боль с паддингом структур

На самом деле всё дело в выравнивании структур по 8 байт

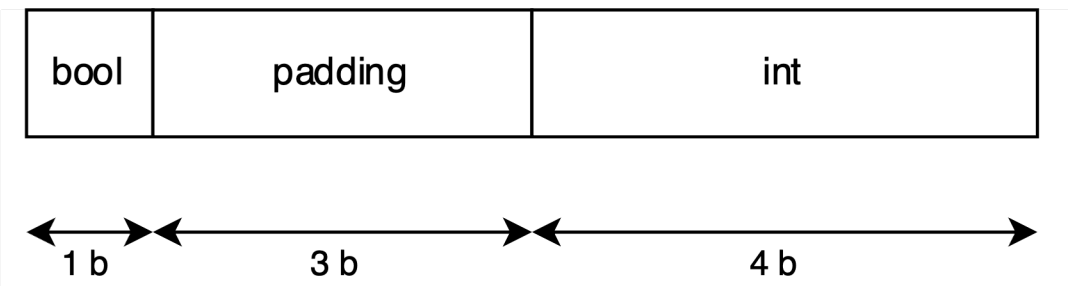
Верификатор не может доказать, что мы не залезем в память, где нет данных и падает

Боль с паддингом структур

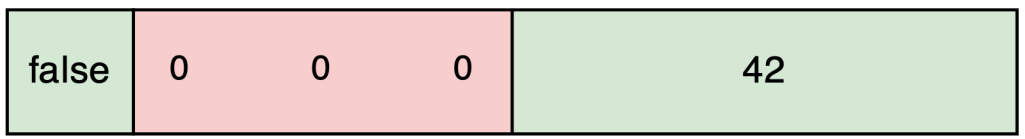
На самом деле всё дело в выравнивании структур по 8 байт

Верификатор не может доказать, что мы не залезем в память, где нет данных и падает

Вот как оно в памяти



А вот так заполняется



Про краш ядра

Дело с версиями оказалось еще интересней

Про краш ядра

Дело с версиями оказалось еще интересней

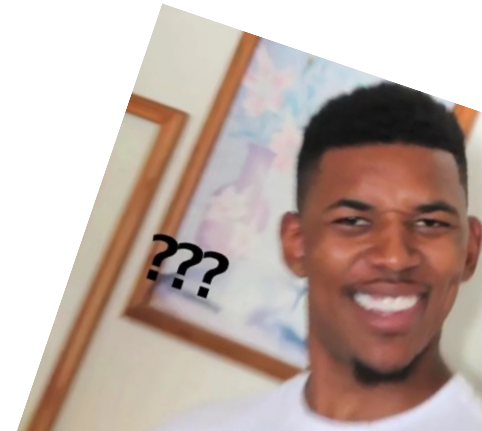
Мы тестировали наш код на разных версиях ядра. На версиях ядра с 5.10 - 5.19 всё работало прекрасно

Про краш ядра

Дело с версиями оказалось еще интересней

Мы тестировали наш код на разных версиях ядра. На версиях ядра с 5.10 - 5.19 всё работало прекрасно

Но был один нюанс, который поставил под сомнение наш проект



Что случилось?

В некотором промежутке версий ядер (5.19 – 6.2) мы словили баг ядра, который фризил всю систему

Что случилось?

В некотором промежутке версий ядер (5.19 – 6.2) мы словили баг ядра, который фризил всю систему

Фризил настолько, что чинилось всё только перезагрузкой

Что случилось?

В некотором промежутке версий ядер (5.19 – 6.2) мы словили баг ядра, который фризил всю систему

Фризил настолько, что чинилось всё только перезагрузкой

Debian Bug report logs - #1033398

reproducible kernel freeze on 5.19+: bpf related issues with copy_from_user_nofault

Package: src:linux; Maintainer for **src:linux** is **Debian Kernel Team <debian-kernel@lists.debian.org>**;

Да, такое есть, но фикс только в версии 6+

Как с этим жить?

Первым делом хотелось всё бросить и поставить крест на нашем проекте, так как часть устройств оно сломает

Как с этим жить?

Первым делом хотелось всё бросить и поставить крест на нашем проекте, так как часть устройств оно сломает

Но у нас нашлось решение:

1) Найти адрес уязвимой функции
2) Используя eBPF вычитать её бинарный код
3) Диассемблировать и проверить, накачен-ли патч

Как с этим жить?

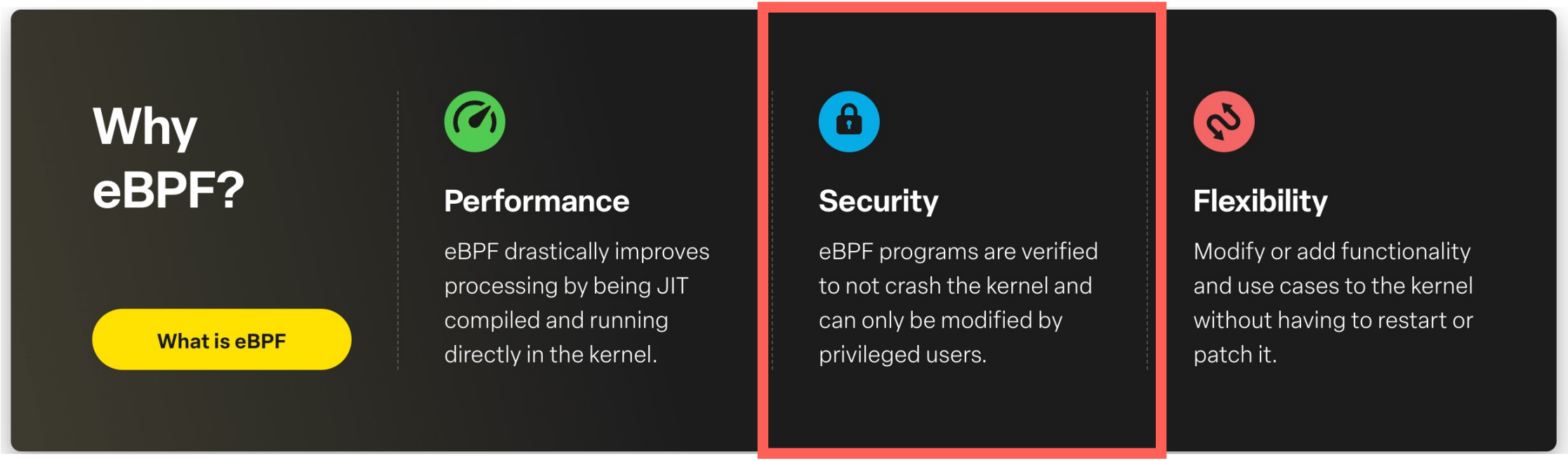
Первым делом хотелось всё бросить и поставить крест на нашем проекте, так как часть устройств оно сломает

Но у нас нашлось решение:

1) Найти адрес уязвимой функции
2) Используя eBPF вычитать её бинарный код
3) Диассемблировать и проверить, накачен-ли патч

Короткое напоминание

Это должно быть безопасно, да?



Выводы

Что хорошего?

Что плохого?