Racehound: система для выявления состояний гонки в ядре Linux

Комаров Н.Ю., аспирант ИСП РАН II г.о.
Научный руководитель: д.ф-м.н., проф. Петренко А.К.

Состояние гонки

Одновременное обращение двух или более потоков в паралелльной программе к одной области памяти при отсутствии между этими обращениями принудительного упорядочивания по времени, когда хотя бы одно из обращений — на запись

Состояние гонки: пример

```
Thread1 {
    while (true) {
        x = x + 1;
    }
}

Thread2 {
    while (true) {
        x = x - 1;
    }
}
```

Состояние гонки: пример

```
Thread1 {
   while (true) {
      mov ax, x
      inc ax
      mov x, ax
   }
}
```

```
Thread2 {
   while (true) {
      mov bx, x
      dec bx
      mov x, bx
   }
}
```

Состояние гонки: пример

		x=1,ax=0,bx=0
mov ax, x		x=1,ax=1,bx=0
inc ax		x=1,ax=2,bx=0
mov x, ax		x=2,ax=2,bx=0
	mov bx, x	x=2,ax=2,bx=2
	dec bx	x=2,ax=2,bx=1
	mov x, bx	x=1,ax=2,bx=1

Состояние гонки: пример

		x=1,ax=0,bx=0
mov ax, x		x=1,ax=1,bx=0
	mov bx, x	x=1,ax=1,bx=1
inc ax		x=1,ax=2,bx=1
	dec bx	x=1,ax=2,bx=0
mov x, ax		x=2,ax=2,bx=0
	mov x, bx	**x=0**,ax=2,bx=0

Linux

- Ядро — многопоточная система
- Исследование комментариев к изменениям в ядре: состояния гонки — первое место, ~17% типовых ошибок[1]
- На II и III месте — утечки специфичных объектов и разыменование нулевого указателя (по ~9%)

[1] В.С. Мутилин, Е.М. Новиков, А.В. Хорошилов. Анализ типовых ошибок в драйверах операционной системы Linux

Выявление состояний гонки

- Lockset
- Happens-before

Lockset

- Предположение: состояния гонки происходят, когда используемая разными потоками память не защищена механизмами синхронизации
- Проверка наличия синхронизации при обращении к общим переменным

Lockset

- Позволяет выявлять значительную часть потенциальных состояний гонки
- Большое количество ложных срабатываний

- Самый известный инструмент — Eraser

Happens-before

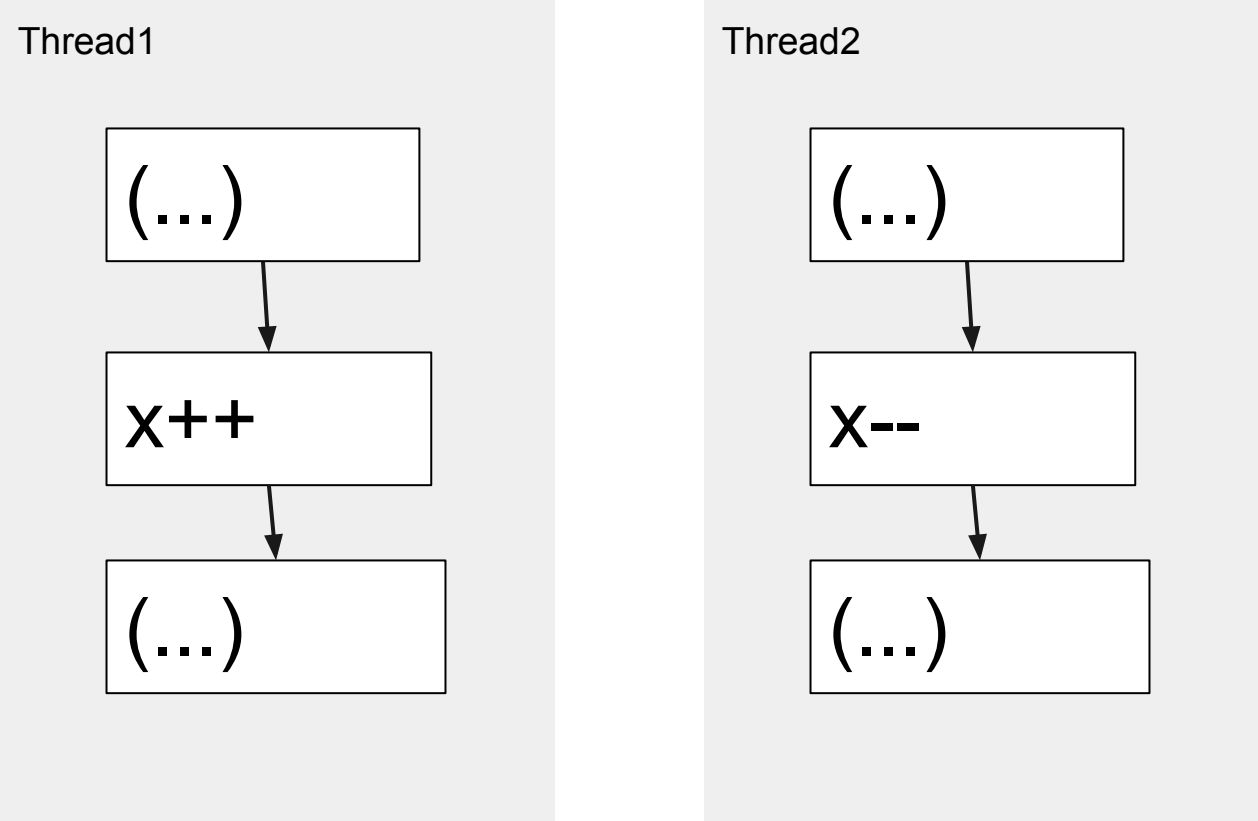
Определение:

- Из двух последовательных событий в одном потоке следующее первым имеет отношение happens-before к следующему вторым
- Операция разблокировки в одном потоке имеет отношение happens-before к операции блокировки в другом потоке для одного синхронизационного объекта
- Отношение happens-before транзитивно

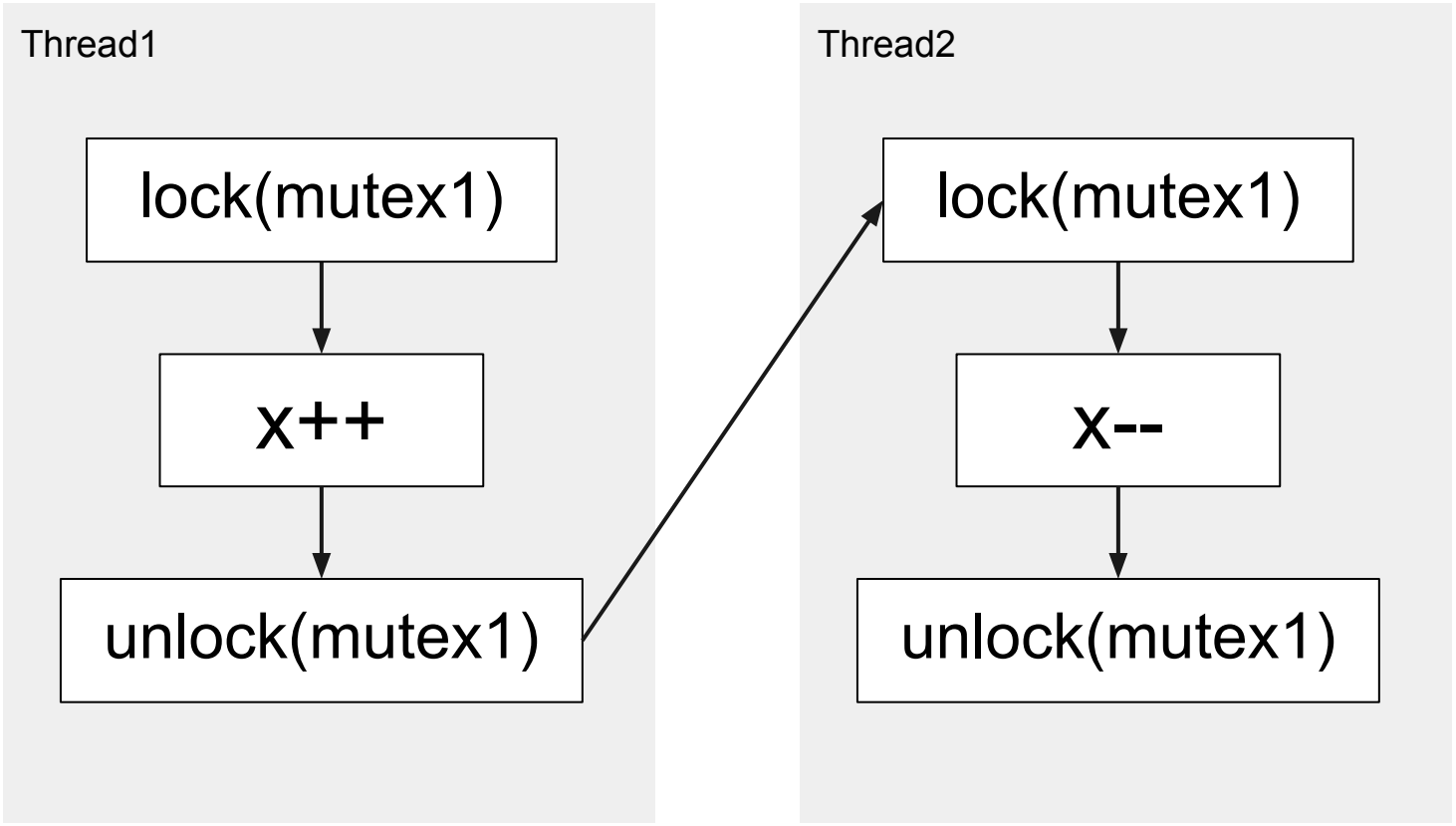
Happens-before

Таким образом, для двух обращений к одной переменной может существовать состояние гонки, если между ними нельзя установить отношение happens-before

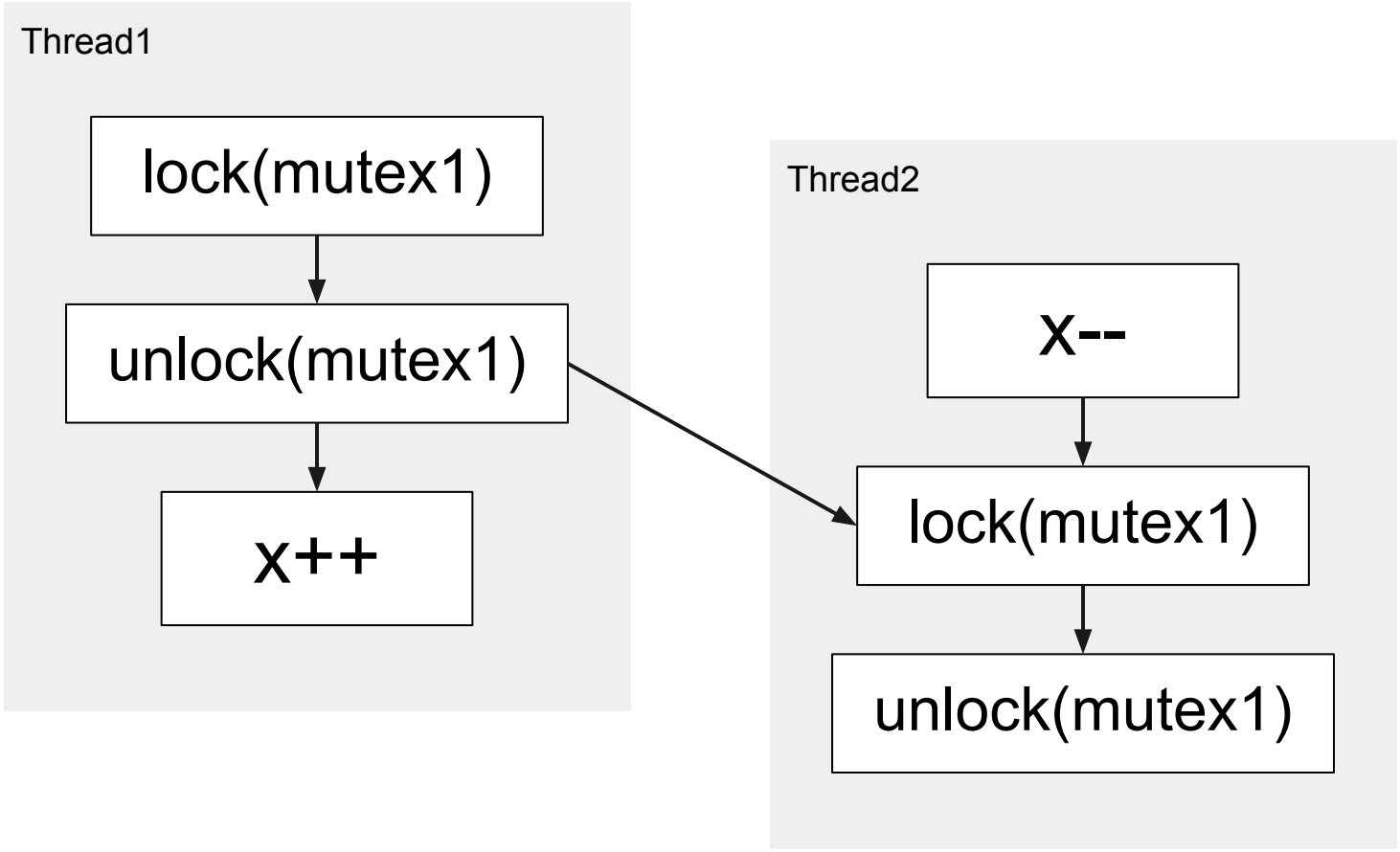
Happens-before



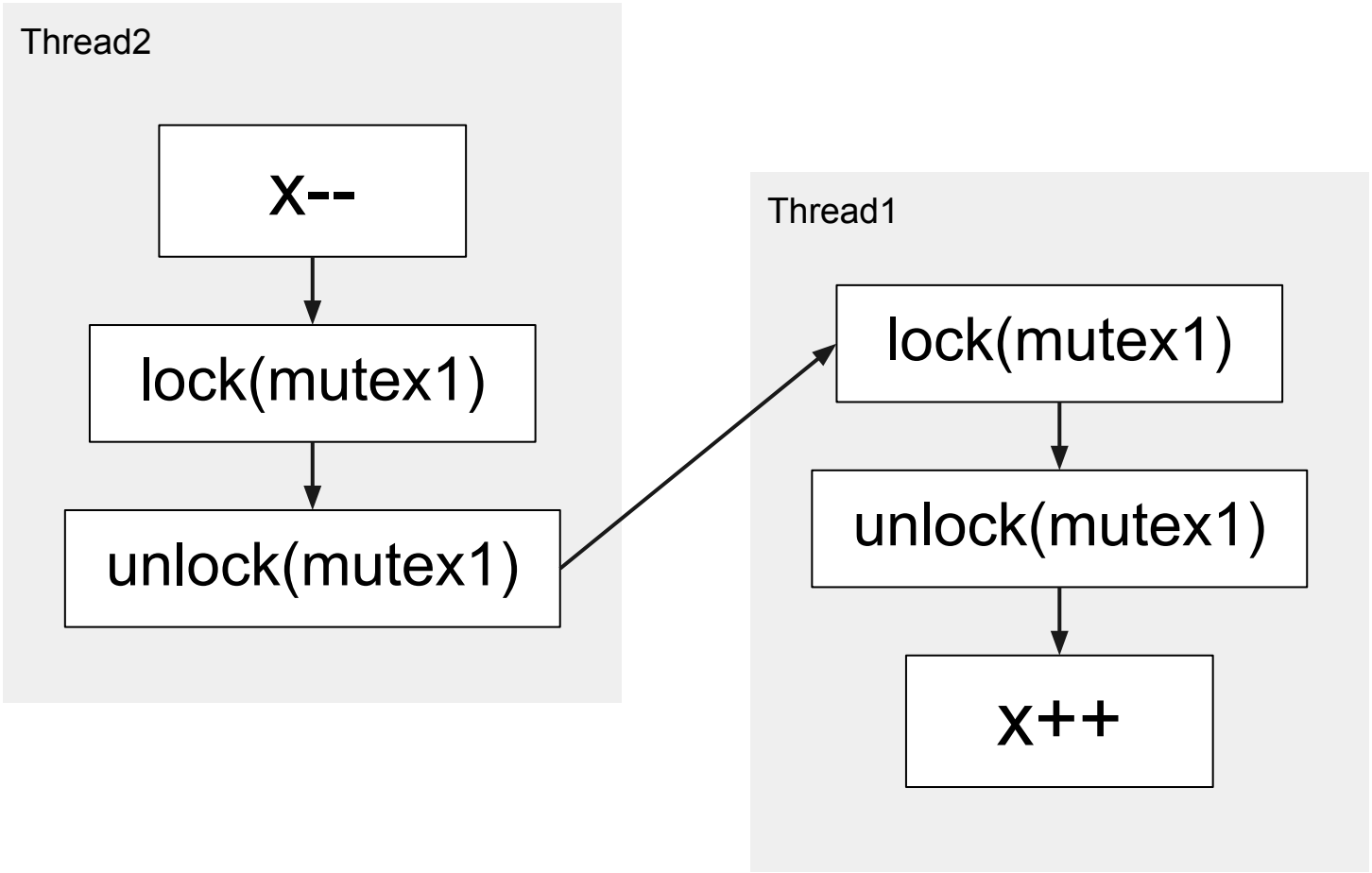
Happens-before



Happens-before



Happens-before



Happens-before

- Позволяет выявлять меньшее количество ошибок, чем Lockset
- Меньше ложных срабатываний, чем у Lockset

ThreadSanitizer

- Happens-before + гибридный режим
- Инструментирование кода: добавление вызовов специальных функций при каждом обращении к памяти и при каждом использовании механизмов синхронизации
- Offline-режим; Kernel Strider

Data Collider

- Microsoft Research
- Для драйверов Windows
- Простой алгоритм, исследующий исключительно реальное выполнение программы

Data Collider: алгоритм

- Периодическая случайная расстановка точек прерывания на выполнение отдельных инструкций программы
- При срабатывании точки прерывания:
  - Декодирование инструкции, получение адреса, по которому она обращается
  - Получение значения по этому адресу
  - Установка точки прерывания на обращение по этому адресу
  - Задержка
  - Повторное получение значения, сравнение с исходным

Data Collider: алгоритм

Состояние гонки точно присутствует, если за прошедшее время:

- сработала точка прерывания на обращение к адресу

или

- значение по адресу изменилось

Data Collider

Достоинства:

- Простой алгоритм, прост в использовании
- Почти не требует вмешательства пользователя при работе
- Не дает ложных срабатываний
- Низкие накладные расходы при выполнении: ~5%

Недостатки:

- Находит небольшую часть ошибок
- Ограничения, связанные с оборудованием

Data Collider

Найдено 25 подтвержденных ошибок
в ядре Windows

Racehound: задача

Постановка задачи:

- Разработать систему для выявления состояний гонки в ядре Linux, использующую алгоритм работы, аналогичный Data Collider

Racehound: компоненты

- Система реализована в виде модуля ядра
- Интерфейс на базе debugfs

Racehound: алгоритм

- Периодическая случайная расстановка программных точек прерывания на выполнение отдельных инструкций программы (Software Breakpoints/Kprobes)
- При срабатывании точки прерывания:
  - Декодирование инструкции, получение адреса, по которому она обращается
  - Получение значения по этому адресу
  - Установка аппаратной точки прерывания на обращение по этому адресу (Hardware Breakpoint)
  - Задержка
  - Повторное получение значения, сравнение с исходным

Racehound: алгоритм

Thread 1	Thread 2	Память
... mov x, %eax ...	... mov %ebx, x ...	... x 0x1234 ...
Software Breakpoint Handler decode(insn) => x set_hw_breakpoint(x) delay() unset_hw_breakpoint(x)	Hardware Breakpoint Handler print(x,caller_address)	Вывод

Racehound: алгоритм

Thread 1	Thread 2	Память
... • mov x, %eax ...	... mov %ebx, x ...	... x 0x1234 ...
Software Breakpoint Handler decode(insn) => x set_hw_breakpoint(x) delay() unset_hw_breakpoint(x)	Hardware Breakpoint Handler print(x,caller_address)	Вывод

Racehound: алгоритм

Thread 1	Thread 2	Память
... • ~~mov x, %eax~~ int 3 ...	... mov %ebx, x ...	... x 0x1234 ...
Software Breakpoint Handler decode(insn) => x set_hw_breakpoint(x) delay() unset_hw_breakpoint(x)	Hardware Breakpoint Handler print(x,caller_address)	Вывод

Racehound: алгоритм

Thread 1	Thread 2	Память
→... • mov x, %eax ...	... mov %ebx, x ...	... x 0x1234 ...
Software Breakpoint Handler decode(insn) => x set_hw_breakpoint(x) delay() unset_hw_breakpoint(x)	Hardware Breakpoint Handler print(x,caller_address)	Вывод

Racehound: алгоритм

Thread 1	Thread 2	Память
... →• mov x, %eax ...	... mov %ebx, x ...	... x 0x1234 ...
Software Breakpoint Handler decode(insn) => x set_hw_breakpoint(x) delay() unset_hw_breakpoint(x)	Hardware Breakpoint Handler print(x,caller_address)	Вывод

Racehound: алгоритм

Thread 1	Thread 2	Память
... →• mov x, %eax ...	... mov %ebx, x ...	... x 0x1234 ...
Software Breakpoint Handler →decode(insn) => x set_hw_breakpoint(x) delay() unset_hw_breakpoint(x)	Hardware Breakpoint Handler print(x,caller_address)	Вывод

Racehound: алгоритм

Thread 1	Thread 2	Память
... →• mov x, %eax ...	... mov %ebx, x ...	... • x 0x1234 ...
Software Breakpoint Handler decode(insn) => x →set_hw_breakpoint(x) delay() unset_hw_breakpoint(x)	Hardware Breakpoint Handler print(x,caller_address)	Вывод

Racehound: алгоритм

Thread 1	Thread 2	Память
... →• mov x, %eax ...	... mov %ebx, x ...	... • x 0x1234 ...
Software Breakpoint Handler decode(insn) => x set_hw_breakpoint(x) →delay() unset_hw_breakpoint(x)	Hardware Breakpoint Handler print(x,caller_address)	Вывод

Racehound: алгоритм

Thread 1	Thread 2	Память
... →• mov x, %eax ...	→... mov %ebx, x ...	... • x 0x1234 ...
Software Breakpoint Handler decode(insn) => x set_hw_breakpoint(x) →delay() unset_hw_breakpoint(x)	Hardware Breakpoint Handler print(x,caller_address)	Вывод

Racehound: алгоритм

Thread 1	Thread 2	Память
... →• mov x, %eax ...	... →mov %ebx, x ...	... • x 0x1234 ...
Software Breakpoint Handler decode(insn) => x set_hw_breakpoint(x) →delay() unset_hw_breakpoint(x)	Hardware Breakpoint Handler print(x,caller_address)	Вывод

Racehound: алгоритм

Thread 1	Thread 2	Память
... →• mov x, %eax ...	... →mov %ebx, x ...	... • x 0x1234 ...
Software Breakpoint Handler decode(insn) => x set_hw_breakpoint(x) →delay() unset_hw_breakpoint(x)	Hardware Breakpoint Handler →print(x,caller_address)	Вывод "Data race on access to *x* from *caller_address*"

Racehound: алгоритм

Thread 1	Thread 2	Память
... →• mov x, %eax ...	... mov %ebx, x →...	... • x 0x1234 ...
Software Breakpoint Handler decode(insn) => x set_hw_breakpoint(x) →delay() unset_hw_breakpoint(x)	Hardware Breakpoint Handler print(x,caller_address)	Вывод "Data race on access to *x* from *caller_address*"

Racehound: алгоритм

Thread 1	Thread 2	Память
... →• mov x, %eax ...	... mov %ebx, x ...	... x 0x1234 ...
Software Breakpoint Handler decode(insn) => x set_hw_breakpoint(x) delay() →unset_hw_breakpoint(x)	Hardware Breakpoint Handler print(x,caller_address)	Вывод "Data race on access to *x* from *caller_address*"

Racehound: алгоритм

Thread 1	Thread 2	Память
... • mov x, %eax →...	... mov %ebx, x ...	... x 0x1234 ...
Software Breakpoint Handler decode(insn) => x set_hw_breakpoint(x) delay() unset_hw_breakpoint(x)	Hardware Breakpoint Handler print(x,caller_address)	Вывод "Data race on access to *x* from *caller_address*"

Racehound: особенности реализации

Декодирование инструкций:

- Декодер из ядра Linux (insn.h, inat.h), доработанный в рамках проекта KEDR
- При старте декодируются все инструкции модуля, выделяются те, которые обращаются к памяти
- При работе на основании инструкции и значений регистров определяется адрес, по которому обращается инструкция, и размер данных по этому адресу

Racehound: проблема

- Проблема: обработчик программной точки прерывания выполняется в атомарном контексте, установка аппаратной точки невозможна
- Решение: возврат в контект процесса, вызов обработчика в нем
- Ограничение: исходная точка прерывания сработала уже в атомарном контексте

Racehound: ограничения

- Архитектура x86 / x86_64
- Бесполезность на одноядерных системах
- Версия ядра Linux >= 3.2

Racehound: состояние

- Работающий вариант системы
- Используется авторами Kernel Strider: подтверждено 3 ошибки в драйвере сетевого адаптера Intel e1000

Racehound: планы

- Дальнейшая апробация на реальных драйверах
- Настройка параметров (количество точек прерывания, алгоритм их расстановки, временные интервалы и т.д.)
- Оценка эффективности работы

- Интеграция с другими методами выявления состояний гонки
- Включение в состав ядра Linux

Вопросы?