Реферат

11 апреля 2025 г.

1 Введение

Запись и воспроизведение программы — процесс записи исполнения программы так, что она может быть воспроизведена с помощью этого же инструмента с целью анализа и выявления ошибок. Он является довольно удобным инструментом для тестирования и отладки. Тем не менее, хочется иметь возможность не просто реализовать функции записи и воспроизведения, но и поддержать действительно мощные возможности отладки, такие как reverse debugging и time travel debugging.

Давайте рассмотрим, как к этой задаче подошли инженеры проекта Record-and-Replay (Далее, RR) в своей статье [1] о разработке этого инструмента.

2 Дизайн

2.1 Record-and-Replay: способ записи и воспроизведения

Задача Record — сохранить некую часть информации об исполнении программы и ее используемых данных, необходимую для того, чтобы в дальнейшем точно воспроизвести поведение записываемой программы.

Задача Replay — воспроизвести исполнение программы, сохраненное в ходе процесса записи, чтобы получить результаты, которые были получены во время записи.

Система RR ставит выполнение данных задач для программы с минимальными накладными расходами, без внесения каких-либо низкоуровневых изменений в окружение и используя лишь нативные возможности, предоставляемые операционной системой.

2.2 Ограничения исполнения

В системе RR используется модель записи исполнения потоков N:1. Это означает, что из всех N имеющихся потоков, одновременно исполняться и, соответственно, записываться, будет лишь один поток. Причина такого решения в сложности контроля исполнения многопоточных программ. Во время записи из-за многопоточности некоторые данные могут быть либо записаны некорректно, либо вовсе потеряны, что недопустимо. Во время воспроизведения требуется корректная синхронизация потоков, добиться которой может быть сложно.

2.3 Детерминизм

При воспроизведении любой произвольной записанной программы необходимо воспроизвести ее в точном совпадении. Это означает, что требуется линеаризовать или же точно определять ход исполнения программы в каждый момент времени. Это не такая простая задача, как может показаться на первый взгляд, так как детерминизм должен поддерживаться за счет контроля со стороны пользовательской программы, не имеющей доступа к большинству возможностей ядра ОС. Конкретный способ реализации будет рассмотрен далее.

3 Технические детали реализации

3.1 Ptrace

`Ptrace` — системный вызов в Linux, позволяющий не только наблюдать за выполнением программы, но и контролировать его. Это ключевое средство, за счет которого достигается большинство возможностей RR.

3.2 Контроль системных вызовов

`Ptrace` дает хорошую вариативность в управлении пользовательских данных для наблюдаемого потока, а также в контроле исполнений системных вызовов внутри него. Это удобно тем, что во время уведомлений можно сохранить и даже изменить состояние пользовательских данных. На рисунке 1 показан пример записи системного вызова `read`.

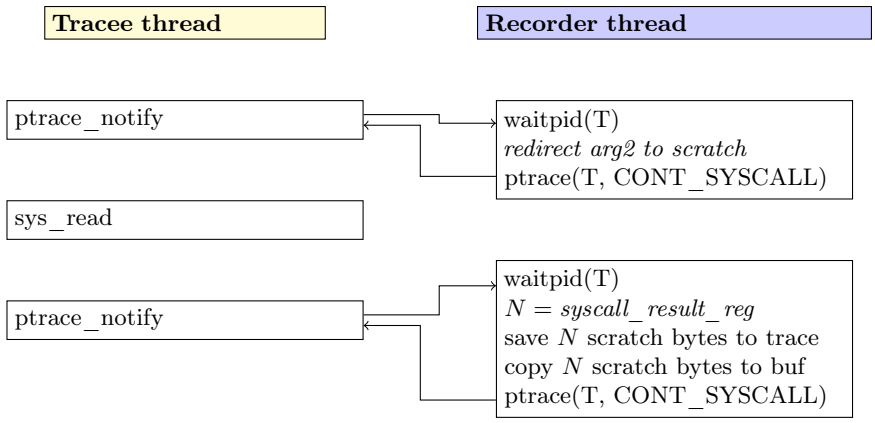

Рис. 1: Процесс записи системного вызова

Есть еще проблемы, связанные системными вызовами:

- Системные вызовы могут быть определены слишком произвольно (например, `ioctl`). Тогда нужно запоминать всевозможные варианты их поведения.
- Некоторые системные вызовы взаимодействуют с потоками или памятью (например, `mmap` и `execve`). С каждым из них требуются собственный подход.

3.3 Обработка недетерминированных инструкций

Существуют инструкции, которые не имеют определенного поведения в зависимости от контекста исполнения (например, `RDTSC`, `RDRAND`, `CPUID` и т.д.). Все они требуют собственного подхода к их обработке.

3.4 Асинхронные события: контроль и обработка

Есть два типа асинхронных событий, которые RR должен поддерживать: смена контекста и сигналы. Эмуляция их срабатываний будет происходить через специальную смену контекста между наблюдателем и наблюдаемым посредством отправки сигнала наблюдаемому потоку. Важно, чтобы во время воспроизведения эти события срабатывали точно в тот же момент, в который они возникли во время записи, то есть когда сигнал был доставлен, состояние исполняющегося потока должно быть идентичным. Об этом в следующей секции.

Во время обработки, когда RR переходит в обработчик сигнала, он записывает содержимое фрейма стека и регистров, установленных ядром. Во время воспроизведения, никакие обработчики не исполняются и никакие сигналы не доставляются по стандартной схеме. Вместо этого, RR просто вставляет записанные данные.

3.5 Детерминизм смены контекста

Для реализации подобного детерминизма будут рассматриваться аппаратные счетчики производительности (`HPC`). Конкретная проблема детерминизма заключается в том, что большинство поддерживаемых счетчиков в Linux не удовлетворяют этому условию по разным причинам. Например, некоторые значения счетчиков могут отличаться от одного воспроизведения к другому из-за содержания кэшей или спекулятивного исполнения инструкций в процессоре.

Тем не менее, современные процессоры на базе Intel имеют поставщик детерминированных данных — счетчик производительности 'количества отработанных ветвлений' (`RCB`). Все же, нельзя полагаться исключительно на него во время определения вызова события как минимум потому, что важна конкретная инструкция, после которой сработало событие во время записи. Следовательно, к счетчику добавляется еще записанное состояние регистров. В большинстве случаев достаточно регистров общего назначения, но иногда записывается также состояние стека.

Для того, чтобы понять, зачем так нужно, рассмотрим следующие примеры кода:

```
loop:
  mov eax, 1
  mov ebx, 2
  inc eax
  jmp loop
```

Листинг 1: Ассемблерный код с необходимостью записи регистров

```
def foo():
  bar()
  bar()
  bar()

def bar():
  some_logic_here()
  return

foo()
```

Листинг 2: Код на Python с необходимостью записи регистров и стэка

```
loop:
  inc [address]
  jmp loop
```

Листинг 3: Проблемный ассемблерный код

Уже в первом примере легко понять, что одного `RCB` не достаточно для определения конкретной инструкции, поскольку здесь нет условных ветвлений и данный счетчик вообще не изменится. Тем не менее, меняется состояние регистров, которое нужно сохранить.

Во втором примере также не происходит изменения `RCB`, так как переход в другую функцию не является условным ветвлением. Помимо этого, сохранения регистров будет недостаточно, поскольку сведения лишь о регистрах и `RCB` не позволят понять, какая функция была вызвана. Следовательно, нужно также хранить состояние стэка.

В третьем примере не будет меняться ничего из выше перечисленного, и, следовательно, вызывает проблемы. Тем не менее, в статье [1] упоминается, что на практике в большинстве случаев подобный сценарий возникает очень редко, однако если он возникнет, то RR просто не сможет воспроизвести его корректно. Полагаю, на практике подобный код редкость из-за того, что большинство сегодняшнего кода не может быть построено без взаимодействия со стеком, а это уже решает проблему.

4 Техники ускорения и оптимизации

4.1 Снижение места под данные трэйса

Простое копирование необходимых бинарников, исполняемых файлов и библиотек быстро раздует память, поэтому вместо этого создаются жесткие ссылки на все необходимые файлы.

Также, современные файловые системы (`XFS`) поддерживают механизм `COW` (copy-on-write). RR просто клонирует нужные области в каталоге трэйса (записанного наблюдаемого потока), не тратя

почти места и времени. Фактическое копирование произойдёт лишь в случае модификации исходного файла.

4.2 Эмуляция системных вызовов

Практика показала, что простое использование `ptrace` неэффективно из-за частой смены контекста. Инженерами [1] были придуманы решения, которые позволяют эмулировать системные вызовы разными способами:

- Ставится точка останова (breakpoint) на момент входа в некоторый системный вызов. Программный счетчик наблюдаемого потока доходит до инструкции системного вызова и его исполнение останавливается. RR это понимает, и в этот момент подменяет данные на те, что должны получиться в результате исполнения этого системного вызова.
- В момент запуска RR размещает в адресном пространстве каждого наблюдаемого процесса специальную библиотеку (interception library). Об этом в следующей секции.

4.3 Interception Library

Interception library — ключевая оптимизация в RR, позволяющая достичь низких накладных расходов за счёт минимизации взаимодействия с `ptrace`. Она внедряется в каждый записываемый процесс, выполняет системные вызова, не вызывая `ptrace`, и записывает результаты исполнения в специальный буфер, разделяемый с RR. В результате, с её помощью удается снизить количество смен контекста до с 2 до 0.

4.4 Оптимизация чтения с клонированием блоков

Если и трэйс, и входные данные находятся в одной файловой системе, поддерживающей copy-on-write, то в коде interception library системного вызова `read` реализуется клонирование блоков в трэйс файл, свой для каждого наблюдаемого потока.

5 Выводы

Проект RR доказал, что эффективная запись и воспроизведение выполнения пользовательских процессов возможны на стандартном оборудовании (x86/x86-64) и ОС (Linux) без модификаций ядра или использования виртуальных машин. Это открывает новые возможности для отладки сложных приложений (например, Firefox, Chromium) с низкими накладными расходами. С более подробными результатами тестирования можно ознакомиться в статье [1].

Список литературы

[1] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush. Engineering record and replay for deployability: Extended technical report. *CoRR*, abs/1705.05937, 2017.