МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ
УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Факультет — Механико-математический

Кафедра — Программирования
Направление подготовки — Математика и компьютерные науки

РЕФЕРАТ

Новиков Сергей Александрович

(Фамилия, Имя, Отчество автора)

Тема:
Исследование статьи ”Engineering Record And Replay For Deployability: Extended Technical Report”

«Реферат принят»

Научный руководитель

Магистр,
индустриальный доцент
кафедры программирования ММФ

Филатов А.Ю. / _____
(ФИО, Подпись)

«. . . » . . . . . 20. . . г.

Новосибирск, 2025

1 Введение

Отладка программного обеспечения является неотъемлемой и зачастую наиболее трудоемкой частью процесса разработки. На заре появления такой возможности, традиционные отладчики, такие как GDB[1] или LLDB[2], имели фундаментальное ограничение: возможность двигаться по коду преимущественно в одном направлении – вперед[3]. Это существенно затрудняет поиск причин сложных, особенно недерминированных или редко воспроизводимых ошибок, когда точные условия возникновения сбоя трудно или невозможно воссоздать вручную.

Так появилась концепция Записи и Воспроизведения. Суть её заключается в фиксации всех недетерминированных входных данных и событий (например, системных вызовов, сигналов, планировщика потоков) во время выполнения программы таким образом, чтобы это выполнение можно было впоследствии точно воспроизвести. Инструмент, реализующий Записи и Воспроизведение, позволяет многократно "проигрывать" проблемный запуск программы, что является важным аспектом для тестирования и выявления ошибок, гарантируя их стопроцентную повторяемость в рамках записанного сеанса.

Тем не менее, возможности простого воспроизведения не всегда достаточны для глубокого понимания причин сбоя. Современные требования к отладке диктуют необходимость в более мощных инструментах. Разработчики стремятся не просто повторить ошибку, а получить возможность "путешествовать во времени" по истории выполнения программы: выполнять шаги назад (reverse debugging), устанавливать точки останова на прошлые события и исследовать состояние программы в любой произвольный момент времени (time travel debugging). Эти возможности позволяют быстро локализовать источник ошибки, отслеживая, как некорректное состояние программы возникло и распространялось.

Одним из наиболее ярких и эффективных инструментов, реализующих эти продвинутые возможности на базе механизма записи и воспроизведения, является отладчик rr (Record and Replay debugger), изначально разработанный в Mozilla. В данном реферате мы подробно рассмотрим подходы и технические решения, предложенные инженерами проекта rr (далее – RR) в их статье [1] для создания этого отладочного инструмента.

2 Дизайн

Основной дизайн отладчика RR строится вокруг двух взаимодополняющих компонентов — механизмов записи (Record) и воспроизведения (Replay). Поговорим подробнее про каждый из этих компонентов.

2.1 Record-and-Replay: способ записи и воспроизведения

- **Концепция Record**

  Когда RR запускает программу в режиме записи, то он делает следующее:

  1. **Наблюдение за взаимодействием с внешним миром:**

     Программа постоянно общается с операционной системой: читает файлы, отправляет данные по сети, получает текущее время, реагирует на ввод пользователя и т.д. Эти взаимодействия могут давать разные результаты при каждом запуске. RR тщательно следит за каждым таким обращением программы к "внешнему миру" (к ядру ОС) и аккуратно записывает:

     – Что программа попросила? (Например, "прочитать 100 байт из файла X").
     – Что ей ответил внешний мир? (Например, "вот эти 100 байт" или "ошибка, файл не найден"). Все эти ответы и данные записываются.

  2. **Фиксация "поворотных моментов" во времени:**

     В программе могут происходить события, время которых заранее неизвестно:

     – **Сигналы:** Внезапные уведомления от операционной системы (например, Ctrl+C или ошибка в другом процессе). RR записывает, какой именно сигнал пришел и в какой момент выполнения программы это случилось.

[1] `https://sourceware.org/gdb/`
[2] `https://lldb.llvm.org/`
[3] Поправка: на текущий момент (2025 год) и GDB, и LLDB могут двигаться и вперед, и назад, но мы будем говорить о том, что они умели в прошлом.

- **Работа с потоками (threads):**
  Если программа многопоточная, порядок, в котором потоки получают процессорное время и взаимодействуют друг с другом (через общую память, синхронизацию и т.д.), может меняться от запуска к запуску. RR старается упорядочить эту работу или, как минимум, очень точно зафиксировать, какой поток сколько "поработал", прежде чем управление перешло к другому потоку или произошло какое-то внешнее событие. Он записывает эту последовательность переключений и "объем работы", выполненный каждым потоком между этими переключениями.
- **Отслеживание некоторых внутренних непредсказуемостей:**
  Некоторые процессорные инструкции могут давать разные результаты (например, чтение счетчика тактов). RR также старается зафиксировать результаты таких операций.

В итоге получается некий "лог" (журнал) всех внешних воздействий и внутренних непредсказуемых событий, а также точных моментов, когда они произошли, относительно "внутреннего времени" программы, который можно использовать для воспроизведения программы.

- **Концепция Replay**

Теперь, имея этот детальный лог, RR может запустить программу снова, но уже в совершенно других условиях:

1. **Изоляция от реального внешнего мира:**
   Когда программа во время воспроизведения пытается снова обратиться к операционной системе (прочитать файл, получить время и т.д.), RR перехватывает этот запрос. Вместо того чтобы обращаться к реальной ОС, RR заглядывает в свой записанный лог и говорит программе: "В прошлый раз, когда ты просила это, тебе ответили вот так. . . ". И подсовывает ей те же самые данные или тот же самый код ошибки, который был во время записи. Программа "думает", что общается с реальным миром, но на самом деле получает ответы из прошлого.
2. **Воссоздание последовательности событий:**
   - **Сигналы:** RR следит за ходом выполнения программы. Когда программа доходит до того "момента" (измеренного по ее "внутреннему времени", как при записи), когда в оригинальном запуске пришел сигнал, RR искусственно доставляет ей точно такой же сигнал.
   - **Работа с потоками:** RR использует записанную информацию о переключениях и "объеме работы" потоков, чтобы заставить их выполняться в точно такой же последовательности и с такими же интервалами, как это было при записи. Никакой "случайности" в планировании потоков больше нет.
3. **Повторение внутренних непредсказуемостей:**
   Если программа обращается к инструкции, результат которой был записан, RR предоставляет ей этот сохраненный результат.

В итоге программа выполняется **абсолютно идентично** тому, как она выполнялась во время записи.

2.2 Ограничения многопоточного исполнения

Многопоточные ограничения при записи:
Когда RR записывает многопоточное приложение, он, по сути, берет на себя значительную часть функций планировщика ОС, но только для потоков целевого процесса.

- **Снижение истинного параллелизма:** Вместо того чтобы позволить ядру ОС свободно планировать потоки приложения по доступным CPU-ядрам, RR обычно **ограничивает их выполнение одним ядром** и сам решает, какой поток будет выполняться следующим. Эмуляция работы на одном ядре означает, что программа во время записи не использует всю мощь многоядерного процессора.
- **Выбор конкретного сценария планирования:** RR записывает один конкретный сценарий взаимодействия потоков, который случился во время сессии записи. Он не пытается исследовать все возможные сценарии.

Многопоточные ограничения при воспроизведении:
Во время воспроизведения RR использует свой лог, чтобы заставить потоки вести себя идентично.

- **Воспроизводится только записанный сценарий:** Если ошибка синхронизации (например, гонка данных) не проявилась во время конкретной сессии записи (из-за того, как RR упорядочил потоки), она не проявится и при воспроизведении этой записи. Для ее поимки может потребоваться несколько попыток записи или специфические условия.

В целом, RR для многопоточности создает управляемую, детерминированную среду, фиксируя одно из возможных поведений системы.

3 Технические детали реализации

Реализация RR ориентирована на минимальное вмешательство в работу целевого приложения и окружения, используя исключительно возможности, предоставляемые современными процессорами и ОС Linux. Нужно понять, за счет чего это достигается.

3.1 Ptrace

Ptrace[4] **(process trace)** — системный вызов в Linux, который предоставляет одному процессу (трассировщику, `'tracer'`) возможность контролировать и наблюдать за выполнением другого процесса (трассируемого, `'tracee'`). Это ключевое средство, за счет которого достигается большинство возможностей RR.

В его реализации нам важно:

1. Трассировщик использует ptrace с определенным запросом (например, `PTRACE_ATTACH` или `PTRACE_TRACEME` при запуске), чтобы "прикрепиться" к трассируемому процессу. С этого момента ядро знает, что этот процесс находится под внешним контролем.
2. Когда в трассируемом процессе происходит "интересное" событие (например, он входит в системный вызов, собирается получить сигнал, выполняет инструкцию точки останова), ядро **приостанавливает** выполнение трассируемого процесса. Затем ядро **уведомляет** трассировщика об этом событии, посылая ему сигнал (обычно, `SIGTRAP`).
3. Если трассируемый процесс остановлен, трассировщик может использовать другие запросы к ptrace для **чтения/записи** регистров CPU трассируемого процесса или его памяти, а также для **управления его дальнейшим выполнением** (например, продолжить нормально, выполнить одну инструкцию, пропустить текущий системный вызов, инжектировать сигнал и т.д.).

Именно за счет этого системного вызова RR способен перехватывать системные вызовы, обрабатывать сигналы, управлять выполнением и получать доступ к памяти и состоянию отслеживаемой программы.

Тем не менее, ptrace имеет свои ограничения и проблемы:

- Каждое событие, перехватываемое ptrace (например, вход/выход из системного вызова, сигнал), требует как минимум двух переключений контекста: от трассируемого процесса к ядру, затем от ядра к трассировщику, и обратно. Это значительно замедляет выполнение трассируемого процесса, особенно если системных вызовов много.
- Сложно контролировать доступы к какому участку пользовательской памяти вне системных вызовов. Тут может помочь введение точек останова по данным или же небольшая модификация кода без изменения глобальных эффектов. В принципе, и тем и другим RR пользуется.
- Некоторые системные вызовы могут быть сложны для точного перехвата и эмуляции всех их побочных эффектов, особенно если они сильно завязаны на внутреннее состояние ядра, информация о котором недоступна через ptrace. RR решает это, фокусируясь лишь на том, что видит трассируемый процесс, а не на глобальном состоянии системы.
- Хотя концепция ptrace есть во многих `UNIX`-подобных системах, конкретная реализация, доступные запросы и их поведение могут немного отличаться, что усложняет создание полностью кроссплатформенных отладчиков, опирающихся только на ptrace. RR в основном сфокусирован на Linux.

[4] `https://man7.org/linux/man-pages/man2/ptrace.2.html`

3.2 Контроль системных вызовов

Для RR, точный контроль и контроль системных вызовов – это фундамент, на котором строится вся система записи и воспроизведения. Ведь именно через системные вызовы программа взаимодействует с недетерминированным внешним миром (файловая система, сеть, ядро, другие процессы).

RR использует ptrace (в частности, его возможность останавливать процесс на входе и выходе из системного вызова, обычно через `PTRACE_SYSCALL`) для полного контроля. В качестве примера, рассмотрим системный вызов `read` на рисунке 1 (на нем отображена фаза Записи):

1. **Фаза Записи (Record):**

   - **Остановка перед вызовом:** Когда трассируемый процесс собирается выполнить системный вызов `sys_read`, ptrace уведомляет RR, выступающим в роли трассировщика (Tracer Thread). Трассирующий процесс останавливается перед тем, как ядро начнет обрабатывать вызов.

     В этот момент RR до срабатывания уведомления узнает об остановке через `wait_pid(T)`[5], считывает номер системного вызова и все его аргументы (из регистров, из памяти по указателям), выполняет **redirect arg2 to scratch**. Это важный момент: **arg2** для `read` – это указатель на буфер, куда будут читаться данные. RR может подменить этот указатель на свой собственный "черновой" (scratch) буфер. Это дает RR полный контроль над тем, куда ядро запишет данные, и позволяет избежать проблем с доступом к памяти трассируемого процесса или изменением этой памяти до того, как RR ее залогирует.

     Затем RR разрешает трассируемому процессу продолжить выполнение внутрь системного вызова, с помощью **ptrace(T, CONT_SYSCALL)**.

   - **Остановка после вызова:** После того как ядро завершило обработку системного вызова, но до того, как управление вернулось обратно в пользовательский код трассируемого процесса, ptrace снова останавливает трассируемый процесс и уведомляет RR.

     RR снова ждет `wait_pid(T)`. Он получает результат **N = syscall_result_reg** (количество прочитанных байт). Затем **save N scratch bytes to trace** – эти N байт из "чернового" буфера сохраняются в лог. И очень важный шаг: **copy N scratch bytes to buf** – RR копирует данные из своего "чернового" буфера в оригинальный буфер трассируемого процесса. Теперь для процесса все выглядит так, как будто `read` выполнился совершенно нормально и записал данные в его буфер.

     RR разрешает трассируемому процессу продолжить обычное выполнение, используя **ptrace(T, CONT_SYSCALL)**.

2. **Фаза Воспроизведения (Replay):**

   Когда программа при воспроизведении доходит до места, где ранее был системный вызов, RR (снова через ptrace) перехватывает эту попытку. **Ключевой момент:** RR **не позволяет** ядру выполнить этот системный вызов повторно.

   Вместо этого RR обращается к своему логу (trace), находит запись об этом конкретном экземпляре системного вызова, **"подделывает"** результат, устанавливая регистр возвращаемого значения (например, `rax`) тем значением, которое было записано в логе. Если системный вызов при записи изменял память (как `read`), RR копирует записанные ранее данные из лога в соответствующие области памяти трассируемого процесса.

   Программа **продолжает** выполнение, "думая", что системный вызов только что успешно (или неуспешно, как было записано) завершился.

[5] `https://man7.org/linux/man-pages/man3/wait.3p.html`

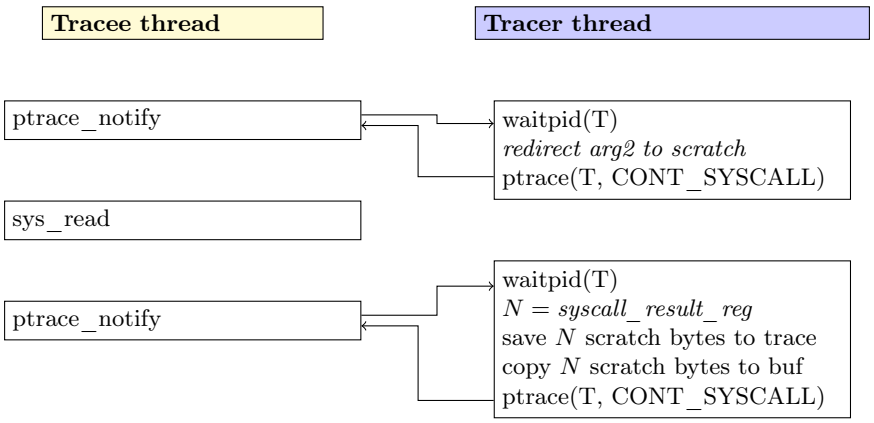

Рисунок 1: Процесс записи системного вызова

Есть еще нюансы, связанные системными вызовами:

1. Системные вызовы могут быть определены **слишком произвольно**[6] (например, `ioctl`[7]). RR должен уметь для распространенных реализаций `ioctl` определять, какие области памяти являются входными, какие выходными, и записывать/воспроизводить их содержимое.
2. Некоторые системные вызовы взаимодействуют с потоками или памятью (например, `mmap`[8] и `execve`[9]):
   - **mmap:** изменяет адресное пространство процесса. RR должен точно записать параметры mmap (адрес, размер, права доступа, флаги, дескриптор файла, смещение) и его результат (полученный адрес). При воспроизведении он должен эмулировать этот `mmap`, чтобы адресное пространство было идентичным (в частности, аллокация страницы на фиксированный адрес достигается с помощью флага `MAP_FIXED`).
   - **execve:** создает новую программу, с новым стеком, кучей, и сегментами данных. RR должен это зафиксировать, записать аргументы, окружение нового процесса и повторить то же поведение `execve`, что и во время записи, загружая те же данные.

3.3 Контроль асинхронных событий

Когда программа выполняется, на нее могут влиять события, происходящие в непредсказуемые моменты – асинхронные события. Ключевыми из них для RR являются системные сигналы и решения операционной системы о переключении между потоками (смена контекста). Чтобы отладчик мог точно воспроизвести выполнение, он должен не только знать, что произошло, но и когда именно это произошло с точки зрения самой программы. Так как по принципу возникновения это одинаковые события, то для удобства восприятия, далее, и то, и другое событие будем называть **сигналами**.

3.3.1 "Внутренние часы"

Сигнал может прервать программу где угодно. Если мы просто запишем "пришел сигнал X", этого будет недостаточно. При повторном запуске тот же сигнал может прийти чуть раньше или чуть позже, застав программу в другом состоянии (с другими значениями в регистрах, на другой инструкции), и ошибка либо не воспроизведется, либо проявится иначе.

Поскольку сигнал может прервать программу где угодно, непонятно, как измерить эту самую "точку выполнения",. Нужны какие-то **"внутренние часы"**. Причем простого использования системного времени будет недостаточно, потому что оно как минимум довольно грубое и подвержено колебаниям.

Для реализации подобных "внутренних часов" инженерами Mozilla [1] были взяты счетчики производительности (`HPC`[10]) на базе современных процессоров Intel. Это специальные регистры процессора, которые аппаратно считают всякие события, происходящие во время исполнения программы. Среди них:

[6] Системный вызов выполняется **слишком произвольно** означает, что его поведение полностью зависит от переданных ему аргументов, в частности от переданного кода команды и указателя на данные, то есть в зависимости от этих аргументов реализации данного системного вызова могут кардинально отличаться.

[7] `https://man7.org/linux/man-pages/man2/ioctl.2.html`

[8] `https://man7.org/linux/man-pages/man2/mmap.2.html`

[9] `https://man7.org/linux/man-pages/man2/execve.2.html`

[10] `https://en.wikipedia.org/wiki/Hardware_performance_counter`

- **Instructions retired:** количество выполненных инструкций. Это количество может быть как с учетом спекулятивных инструкций (то есть инструкций, выполненных процессором заранее, но результат исполнения которых может быть проигнорирован), так и без учета.
- **Retired conditional branches:** количество выполненных условных переходов (также может быть как с учетом спекулятивных ветвлений, так и без). Условным переходом считается команда, которая передаёт управление по другому адресу (на метку) только если выполняется определённое условие (например, такими командами являются `je`, `jl`, `jz` и т.д.). RR в основном используют именно этот счетчик в роли "внутренних часов", потому что, как показала практика, этот счетчик меньше всего подвержен влиянию состояния процессора, на которое RR не может повлиять, а именно содержание кэшей или спекулятивное исполнение инструкций. **Важно:** RR использует `RCB` **без спекулятивных ветвлений**, поскольку в противном случае этот счетчик будет увеличиваться даже в тех случаях, когда RR не может гарантировать, что условное ветвление произошло.

В режиме Записи, когда RR перехватывает сигнал, он немедленно считывает текущее значение этих "внутренних часов" и записывает в лог тип сигнала, показание "внутренних часов" в этот момент, а также полное состояние регистров CPU и, возможно, ключевые участки стека[11] (далее станет ясно, почему это важно).

В режиме Воспроизведения RR также отслеживает показания "внутренних часов". Как только они достигают значения, которое было записано в логах для данного сигнала, RR останавливает программу, восстанавливает записанное состояние регистров/стека и "доставляет" программе тот же самый сигнал. Это гарантирует, что сигнал застанет программу в абсолютно идентичном состоянии.

3.3.2 Ограничения счетчиков

Как уже было сказано, помимо значения "внутренних часов", RR также записывает полное состояние регистров и иногда даже состояние стека. Для того, чтобы понять, зачем так нужно, рассмотрим следующие примеры кода:

```
1   loop:
2     mov eax, 1
3     mov ebx, 2
4     inc eax
5     jmp loop
```

Листинг 1: Ассемблерный код с необходимостью записи регистров

```
1   bar():
2     push    rbp
3     mov     rbp, rsp
4     mov     DWORD PTR [rbp-4], 1    ; int a = 1;
5     pop     rbp
6     ret
7   foo():
8     push    rbp
9     mov     rbp, rsp
10    call    bar()
11    call    bar()
12    call    bar()
13    pop     rbp
14    ret
```

Листинг 2: Ассемблерный код с необходимостью записи регистров и стэка

[11] Уточнение про сохранение "состояние стека": По большей части под ним подразумевается сохранение значения регистров `rsp` и `rbp` (они определяют текущий кадр стека) и последовательности сохраненных регистров `rbp` (это адреса на фреймы предыдущих функций, которые помогают определить стек вызовов).

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

Листинг 3: Ассемблерный код с проблемой

Уже в листинге 1 легко понять, что одних "внутренних часов" 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. Логично предположить, что одного регистра `IP`, хранящего адрес смещения следующей инструкции, будет достаточно. Но это не так, потому что в качестве контрпримера достаточно рассмотреть любой цикл, который проходит несколько раз некоторую инструкцию (например, `add eax, 1`). В каждый момент прохождения в цикле данной инструкции регистр `IP` будет указывать на нее (так как адрес в памяти тот же), а счетчик `RCB` не изменится (так как инструкция `jmp` не является условным ветвлением). Однако, если бы мы сохранили состояние регистра `eax`, то смогли бы определить конкретную итерацию цикла, на которой нам нужно воспроизвести сигнал.

В листинге 2 также не происходит изменения "внутренних часов", так как переход в другую функцию (вызов `call` не является условным ветвлением). Помимо этого, сохранения регистров будет недостаточно, поскольку сведения лишь о регистрах и "внутренних часах" не позволят понять, какая функция была вызвана. Чтобы однозначно определить функцию среди набора идентичных, нужно хранить ключевые участки стека, потому что там гарантированно есть различная информация, по которой можно отличить вызовы функций. Так, в листинге 2 на стеке будет храниться адрес возврата из соответствующей функции `bar` в функцию `foo`, а именно на соответствующую `call` инструкцию, адрес смещения которой отличается от адреса двух других.

В листинге 3 не будет меняться ничего из выше перечисленного, и, следовательно, вызовет проблемы. Тем не менее, в статье [1] упоминается, что на практике в большинстве случаев подобный сценарий возникает очень редко, поскольку код, сгенерированный компилятором, почти всегда взаимодействует со стеком (как минимум в прологе и эпилоге, и этого уже в основном достаточно, чтобы однозначно определить конкретную функцию), однако если он возникнет, то RR просто не сможет воспроизвести его корректно.

4 Техники ускорения и оптимизации

4.1 Снижение места под данные логов

Цель RR – записать ровно столько информации, сколько необходимо для точного детерминированного воспроизведения. Какие вопросы нужно решить:

- Между двумя событиями, которые RR должен зафиксировать (например, двумя системными вызовами или сигналом), программа может выполнить миллионы инструкций. Если эти инструкции не взаимодействуют с внешним миром и их поведение полностью предсказуемо, то нет смысла записывать состояние программы после каждой из них. Это достигается за счет упомянутых выше **аппаратных счетчиков производительности (HPC)**, которые позволяют RR "пролетать" с точки зрения логирования большие участки кода, где ничего недетерминированного не происходит.

- Представьте системный вызов **read()**, который просит прочитать 1 мегабайт данных из файла. Если в файле осталось только 10 килобайт, то будет прочитано именно 10 КБ. Если бы RR всегда резервировал в логе место под запрошенный 1 МБ, лог бы быстро раздулся.

  RR записывает в лог только те данные, которые фактически были переданы или изменены системным вызовом. В нашем примере с **read()**, если было прочитано 10 КБ, то именно эти 10 КБ (и информация о том, что было прочитано 10 КБ) и попадут в лог. Это значительно экономит место, особенно для программ с активным файловым или сетевым `IO`.

- Файлы логов, создаваемые RR, часто автоматически сжимаются с использованием стандартных алгоритмов компрессии (например, с помощью `zlib` в deflate-формате).

4.2 In-process system call interception

Это техника, при которой RR встраивает небольшой код непосредственно в адресное пространство отлаживаемой программы (tracee) для перехвата системных вызовов.

В память трассируемого процесса инъецируется[12] специальный "буфер системных вызовов" (`syscall buffer`) и небольшой код-обработчик (`stub`). Точки входа в системные вызовы (например, в `libc`[13]) модифицируются так, чтобы вызывать этот инъецированный stub. Через syscall buffer RR может взаимодействовать со stub, который в располагается в адресном пространства трассируемого процесса.

В фазе Записи Stub записывает аргументы системного вызова в `syscall buffer`. Затем он может либо выполнить оригинальный системный вызов прямо внутри себя, либо сигнализировать RR (как трассировщику), который затем через ptrace даст команду на выполнение. После выполнения результат также записывается.

В фазе Воспроизведения Stub записывает "намерение" **вызвать syscall** в буфер. Основной процесс RR видит это, находит в логе записанный ранее результат этого вызова и передает его обратно в stub. Stub эмулирует завершение syscall'а, подставляя записанные значения (возвращаемое значение, данные в буферах). Реальный системный вызов к ядру не происходит.

Из плюсов данной техники можно выделить:

- **Снижение накладных расходов:** Вызов системного вызова через `ptrace` требует переключения контекста между процессами, что значительно замедляет выполнение. Используя in-process interception, RR может избежать этого переключения, что приводит к значительному увеличению производительности.
- **Гибкость:** Дает больше контроля над исполнением внутри трассируемого процесса.

Из минусов можно отметить:

- **Сложность:** Инъекция кода, его синхронизация с внешним трассировщиком, обработка пограничных случаев (сигналы, многопоточность) — всё это очень непросто.

5 Выводы

Разработка RR отладчика основана на принципе **записи и детерминированного воспроизведения** исполнения программы. На этапе **записи** RR перехватывает все источники недетерминизма: системные вызовы (их аргументы и результаты логируются, часто с помощью in-process syscall interception и syscall-буфера для производительности), а также сигналы (время и тип доставки). Все эти события и их исходы записываются в детальный лог. На этапе **воспроизведения** RR заставляет программу выполняться, но вместо реальных обращений к ОС или выполнения недетерминированных инструкций, он "подсовывает" ей значения из этого лога, точно воссоздавая первоначальное поведение. **Time-travel debugging**, включая **reverse debugging**, реализуется поверх этого: "перемотка вперед" — это просто воспроизведение до нужной точки, а "шаг назад" достигается не истинным обратным исполнением, а восстановлением состояния программы из ранее сохраненного **чекпоинта** (снимка памяти и регистров) и быстрым воспроизведением вперед с этого чекпоинта до точки, непосредственно предшествующей желаемому моменту отката.

Ключевые выводы разработки: борьба с недетерминизмом — основная сложность; производительность записи критична (требуются оптимизации вроде in-process перехвата); воспроизведение эффективно за счет эмуляции syscall'ов по логу; система получается сложной, хрупкой и сильно завязанной на ОС и архитектуру, требуя баланса между полнотой покрытия и практической реализуемостью.

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.

[12] `https://en.wikipedia.org/wiki/DLL_injection`
[13] `https://www.gnu.org/software/libc/`