DOI: 10.15514/ISPRAS-2019-1(2)-1

CC BY

Идентификация реквизитов сборки через отслеживание системных вызовов

[1,2]А.М. Гранат, ORCID: 0009-0007-6589-3347, <artemiigranat@gmail.com>
[1,2]П.Д. Дунаев, ORCID: 0000-0002-9142-0945, <herrpaulvondonau@outlook.com>
[1,2]А.А. Синкевич, ORCID: 0009-0002-3364-6468, <artsin666@gmail.com>
[2]И.А. Батраева, ORCID: 0000-0002-6539-8473, <batraevaia@info.sgu.ru>
[2,3]Д.Ю. Петров, ORCID: 0000-0001-6364-2084, <iac_sstu@mail.ru>
[1]Институт системного программирования РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25.
[2]Саратовский государственный университет имени Н.Г. Чернышевского, 410012, Россия, Саратов, ул. Астраханская, 83.
[3]Институт проблем точной механики и управления РАН, 410028, Россия, Саратов, ул. Рабочая 24

Аннотация. В статье рассматриваются методы идентификации реквизитов сборки, описываются их сильные и слабые стороны. Представлен инструмент, обеспечивающий журналирование процесса сборки с помощью отслеживания системных вызовов. Приводится оценка временных затрат на сборку с использованием разработанного инструмента.
Ключевые слова: база данных компиляции; перехват сборки; ptrace

Identifying Build Requisites via System Call Tracing

[1,2]A.M. Granat, ORCID: 0009-0007-6589-3347, <artemiigranat@gmail.com>
[1,2]P.D. Dunaev, ORCID: 0000-0002-9142-0945, <herrpaulvondonau@outlook.com>
[1,2]A.A. Sinkevich, ORCID: 0009-0002-3364-6468, <artsin666@gmail.com>
[2]I.A. Batraeva, ORCID: 0000-0002-6539-8473, <batraevaia@info.sgu.ru>
[2,3]D.Yu. Petrov, ORCID: 0000-0001-6364-2084, <iac_sstu@mail.ru>
[1]Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
[2]Saratov State University, 83 Astrakhanskaya Street, Saratov, 410012, Russia.
[3]Institute for Problems of Precision Mechanics and Control, 24 Rabochaya Street, 410028

Abstract. In this work, we discuss methods for identifying build requisites, and describe their strengths and weaknesses. The sbom-trace tool is presented that provides logging of the build process by tracking system calls. An estimate of the time spent on build process using the sbom-trace tool is given.
Keywords: compilation database; build interception; ptrace

1. Введение

В условиях стремительного развития технологий и растущей сложности современных систем, безопасная разработка программного обеспечения приобретает ключевое значение. Организации все чаще сталкиваются с необходимостью обеспечения безопасности на всех этапах жизненного цикла разработки программного обеспечения, чтобы защитить свои системы и данные от потенциальных угроз.

1 апреля 2024 года был введён государственный стандарт «ГОСТ Р 71206-2024 Защита информации. Разработка безопасного программного обеспечения. Безопасный компилятор языков C/C++. Общие требования» [1], направленный на установление единых требований к безопасному компилятору языков Си и C++. Одним из пунктов стандарта является функция журналирования процесса трансляции, с сохранением такой информации, как параметры компиляции и хэш-суммы входных и выходных файлов.

Несмотря на то, что стандарт ориентирован на языки Си и C++, функция журналирования полезна и для программ, написанных на других компилируемых языках. Это подчеркивает актуальность разработки инструмента для журналирования процесса трансляции, независимого от языка программирования. Такой инструмент может стать важным компонентом системы безопасности программного обеспечения, позволяя обнаруживать и устранять уязвимости на этапе компиляции.

Software Bill of Materials (SBoM) представляет собой детализированный список всех компонентов, используемых в программном продукте. Этот инструмент позволяет организациям лучше понимать и управлять компонентами, входящими в их программные продукты, что в свою очередь способствует более эффективному выявлению и реагированию на уязвимости.

Введение стандарта ГОСТ Р 71206-2024 открывает новые возможности для интеграции SBoM в процесс безопасной разработки. Такой подход не только улучшит безопасность разрабатываемых программ, но и обеспечит большую совместимость с международными стандартами по безопасности программного обеспечения.

Цель статьи — анализ методов идентификации реквизитов сборки, описание разработанного инструмента для генерации базы данных компиляции `sbom-trace`, оценка времени работы процесса сборки с использованием инструмента и без него. В разделе 2 описывается проблема генерации базы данных компиляции, существующие инструменты и подходы к данной проблеме, в разделах 3 и 4 — описаны механизмы `ptrace` и `seccomp`, выбранные для использования в итоговом инструменте. Далее в разделе 5 описана архитектура и реализация `sbom-trace`, алгоритм работы его компонентов. В разделе 6 подводятся итоги исследования.

2. Описание проблемы

2.1. База данных компиляции

База данных компиляции представляет собой структурированное хранилище данных, содержащее информацию о процессах компиляции программного кода. Такая база данных может включать детали о каждом шаге компиляции, включая используемые файлы, их версии, а также параметры и результаты работы компилятора.

База данных компиляции может быть полезна для проведения статического анализа кода. Статические анализаторы, такие как Clang Static Analyzer [2], Coverity [3] или Svace [4], используют информацию о процессе компиляции из соответствующего файла (например, Clang Static Analyzer по умолчанию читает информацию из файла `compile_commands.json` в рабочей директории проекта) для выполнения детального анализа исходного кода на предмет потенциальных ошибок и уязвимостей. С помощью базы данных компиляции анализаторы могут точно понять, как код был собран, и, следовательно, провести более точный анализ.

Из других случаев, когда генерация базы данных компиляции может принести пользу, можно выделить портирование программного обеспечения на другие платформы — при портировании программного обеспечения на новые платформы или архитектуры важно знать параметры и конфигурацию компиляции. База данных компиляции позволяет разработчикам понимать исходные параметры компиляции и адаптировать их под новые условия.

В соответствии с государственным стандартом «ГОСТ Р 71206-2024 Защита информации. Разработка безопасного программного обеспечения. Безопасный компилятор языков C/C++. Общие требования», для каждой единицы трансляции база данных компиляции должна хранить:

- рабочую директорию компиляции;
- имя и хэш-сумму файла, представляющего единицу трансляции и выходного файла, а также каждого файла, включаемого в процесс компиляции;
- выполняемую команду компиляции целиком или полный список аргументов компиляции [1].

2.2. Обзор существующих инструментов

Для сбора параметров компиляции и генерации файла, содержащего информацию о командах компиляции, применяемой в различных инструментах анализа кода, используется утилита `Bear` [5]. Она перехватывает вызовы компилятора во время сборки проекта и создаёт соответствующий файл. Команда `bear -- <build_command>` позволяет получить необходимую информацию для последующего анализа кода. Помимо Bear, также применяется, например, `compiledb` [6], подходящая для сборочных систем, основанных на Makefile. Особенность `compiledb` заключается в использовании флага `-n`, который позволяет сгенерировать базу данных компиляции без запуска сборки.

Кроме того, инструменты сборки проектов, такие как `CMake`, `Ninja`, `Qt Build System` и `qmake`, также способны автоматически генерировать файлы конфигурации, содержащие информацию о параметрах компиляции.

`CDE` [7] решает смежную задачу: для транспортировки необходимого пакета в той же среде, в которой он был собран, этот инструмент сохраняет информацию о файлах, открытых в процессе сборки, и включает их в состав пакета. Это позволяет при сборке на другой машине использовать файлы, сохраненные вместе с пакетом на исходной системе.

Еще одним подходом к генерации базы данных компиляции является встраивание механизма непосредственно в компилятор и его активация с помощью определённых флагов компиляции. Например, в компиляторе Clang доступна функциональность генерации JSON-файла с информацией о каждом этапе компиляции при указании флага `-MJ` [8]. В безопасном компиляторе «SAFEC» [9], разработанном в Институте системного программирования РАН на основе GCC [10], также добавлена функциональность генерации базы данных компиляции с помощью флага компиляции `--dump-sbom <dump-dir>`.

Однако, одним из основных ограничений генерации базы данных с помощью внедрения дополнительной функциональности в компилятор является сложность отслеживания всех внешних зависимостей при использовании сложных сценариев сборки или нестандартных инструментов.

Кроме того, прямая работа с компилятором ограничивает доступность данных только той информацией, которую компилятор может предоставить, что может быть недостаточно для некоторых задач.

Для преодоления ограничений возможностей компилятора для генерации базы данных компиляции, существуют альтернативные подходы, которые могут быть использованы для сбора информации о процессе компиляции.

2.3. Альтернативные подходы к генерации базы данных компиляции

В качестве альтернативных подходов для операционных систем семейства Linux можно выделить:

1. `FUSE` (Filesystem in Userspace) — механизм, позволяющий пользователям создавать собственные файловые системы в пользовательском пространстве, что в свою

очередь позволяет реализовывать собственные механизмы отслеживания и журналирования операций над файлами, связанных с процессом компиляции. Несмотря на возможности, `FUSE` может работать медленнее, чем файловая система, реализованная на уровне ядра ОС, поскольку все операции должны проходить через пользовательское пространство, что добавляет накладные расходы при переключении контекста [11]. Также механизм не обеспечивает необходимый инструмент информацией о командах компиляции, что приводит к зависимости от инструмента для их отслеживания.

2. `inotify` — это механизм ядра Linux, который позволяет отслеживать изменения в файловой системе. Он может быть использован для мониторинга действий, связанных с файлами и директориями, с которыми происходит взаимодействие в процессе компиляции, и запуска определённых действий при обнаружении изменения или доступа к файлам, за которыми ведется наблюдение [12]. Плюсом данного подхода является отслеживание всех действий, связанных с работой с файлами, однако `inotify` имеет тот же недостаток, что и `FUSE`, связанный с отсутствием информации о командах компиляции. Другим минусом данного механизма является ограничение на количество отслеживаемых файлов, хранящееся в значении переменной `max_user_watches` конфигурации `inotify`, что приводит к невозможности работы с масштабными проектами;
3. `ptrace` — инструмент, позволяющий отслеживать и манипулировать процессами в операционной системе. Его механизм может быть использован для наблюдения за работой компилятора и сбора информации обо всех его действиях — открытие файлов и работа с ними, вызовы команд компиляции, создание новых процессов, окончание их работы. Из плюсов данного подхода можно выделить возможность отслеживать каждый системный вызов, полученный в процессе компиляции, что даёт полное представление о взаимодействии сборочной системы с ОС, а это, в свою очередь, решает проблему работы с внешними зависимостями и сложными системами сборки. Из минусов можно выделить необходимость учитывать зависимости от процессорной архитектуры при разработке инструмента, например, имена регистров, в которых хранятся данные о системном вызове [13].

Наиболее релевантным подходом для разработки инструмента, решающего задачу сбора информации о процессе компиляции является `ptrace`, так как он обеспечивает полный контроль над исполнением процесса компиляции и собирает подробную информацию о его действиях.

3. Системный вызов ptrace

Основная сфера применения `ptrace` — инструменты для отладки с использованием точек останова (например, `gdb`) и трассировщики системных вызовов (`strace` и подобные инструменты).

Для языка Си интерфейс для работы с `ptrace` предоставлен в заголовочном файле `sys/ptrace.h` — саму функцию для системного вызова и перечисление возможных запросов `__ptrace_request`, необходимых для взаимодействия с трассируемым процессом, получением необходимой информации и управления его выполнением [14].

Начать отладку процесса можно двумя способами:

1. Присоединиться к процессу с помощью запросов `PTRACE_ATTACH` или `PTRACE_SEIZE`;
2. Сделать родителя текущего процесса трассировщиком с помощью запроса `PTRACE_TRACEME`.

Общий механизм работы `ptrace` имеет следующий вид:

1. Процесс, предназначенный для отслеживания, получает сигнал `SIGSTOP` и

трассировщик подключается к нему;
2. Трассируемый процесс останавливается и трассировщик может получить информацию (запросы `PTRACE_PEEK*`, `PTRACE_GET*`) о его текущем состоянии или как-либо его изменить (`PTRACE_POKE*`, `PTRACE_SET*`) [15];
3. Трассировщик перезапускает отслеживаемый процесс одним из способов перезапуска (`PTRACE_CONT`, `PTRACE_SYSCALL`, `PTRACE_SINGLESTEP`).

Проверить, что процесс перешел в состояние остановки, можно с помощью макроса `WIFSTOPPED` (истинно, если процесс находится в состоянии остановки), передав ему статус процесса, полученный из системного вызова `wait`.

Существует несколько видов состояния остановки:

1. Остановка на системном вызове (syscall stop): вызывается непосредственно при системном вызове и сразу после его завершения в случае, если процесс перезапускается с помощью команды `PTRACE_SYSCALL`.
2. Остановка на событии (event stop): вызывается при событии, специфичном для `ptrace`, например:
    1. `PTRACE_EVENT_FORK`, `PTRACE_EVENT_VFORK` или `PTRACE_EVENT_CLONE` — происходит при вызове `fork()`, `vfork()` или `clone()` отслеживаемым процессом;
    2. `PTRACE_EVENT_EXEC` — срабатывает при выполнении `execve()`;
    3. `PTRACE_EVENT_EXIT` — остановка перед завершением работы процесса.

    В основном, чтобы трассируемый процесс останавливался на подобных событиях, необходимо явно прописать для ядра ОС флаги, которые позволяют отслеживать их, например, чтобы происходила остановка на событии `PTRACE_EVENT_EXEC`, при подключении к трассируемому процессу необходимо выставить флаг `PTRACE_O_TRACEEXEC`.
3. Остановка при доставке сигнала (signal-delivery-stop): вызывается при сигнале, полученном трассируемым процессом. Трассировщик получает информацию о полученном сигнале первым, и может модифицировать его обработку, либо перезапустить процесс. Исключение в этом случае — сигнал `SIGKILL`, так как в этом случае трассируемый процесс немедленно завершается.
4. Групповая остановка (group-stop): возникает, когда все потоки в группе процессов находятся в состоянии остановки из-за получения сигнала, который останавливает процесс — `SIGSTOP`, `SIGTSTP`, `SIGTTIN` или `SIGTTOU`. В случае, если трассировщик был подключен с помощью запроса `PTRACE_SEIZE`, такой вид остановки сопровождается событием `PTRACE_EVENT_STOP`. Когда такой сигнал отправляется группе процессов, все потоки в группе приостанавливают свою работу и переходят в состояние остановки. После того как процесс был остановлен, трассировщик может возобновить его выполнение с помощью запроса `PTRACE_CONT` или других аналогичных команд. Однако в случае групповой остановки процесс перестает реагировать на сигналы до тех пор, пока не получит `SIGCONT`. Если групповой процесс был переведен в состояние ожидания запросом `PTRACE_LISTEN`, он также не будет реагировать на команды `ptrace` до получения `SIGCONT` [16].

4. Механизмы seccomp и BPF

`seccomp` (от англ. Secure Computing Mode) — механизм ядра Linux, предназначенный для ограничения системных вызовов, которые может выполнять процесс. Технология повышает безопасность системы за счёт фильтрации доступных системных вызовов, что позволяет

предотвратить множество видов атак, основанных на эксплуатации уязвимостей программного обеспечения.

В Linux `seccomp` реализуется ввиде двух режимов:

1. Строгий режим, который ограничивает процесс только четырьмя системными вызовами: `read()`, `write()`, `_exit()`, и `sigreturn()`. Любые другие системные вызовы в данном режиме приводят к вызову сигнала `SIGKILL`;
2. Фильтрационный режим, разрешающий настройку ограничений с помощью BPF. В этом режиме имеется возможность задать произвольный набор правил в коде программы, определяющих, какие системные вызовы разрешены, а какие блокируются.

BPF (Berkeley Packet Filters) — технология, разработанная в 1992 году для эффективной фильтрации сетевого трафика на уровне ядра операционной системы. В Linux 3.5 BPF был адаптирован для фильтрации системных вызовов через seccomp, что позволило использовать его в различных сценариях безопасности и мониторинга.

Виртуальная машина BPF предоставляет простой набор инструкций, который позволяет выполнять базовые операции, такие как загрузка, хранение, переходы между инструкциями на определенное количество шагов и арифметические операции. Чтобы предотвратить возможное “зависание” ядра операционной системы, в программах BPF запрещены циклы и наложены другие ограничения, включая лимит на 4096 инструкций для одной программы.

Фильтры BPF компилируются во время выполнения и загружаются в ядро для применения к вызовам системных функций. Это дает разработчикам возможность точно настроить поведение приложений в зависимости от их потребностей в системных ресурсах, что значительно повышает уровень безопасности.

В режиме `seccomp` BPF используется для создания детализированных политик безопасности, которые контролируют, какие системные вызовы могут выполняться. Когда процесс переходит в режим `seccomp` и выполняет системный вызов, ядро применяет соответствующий фильтр BPF к этому вызову.

Чтобы установить фильтр BPF, должно быть выполнено одно из условий:

1. Пользователь, инициирующий действие, должен иметь привилегии администратора (`CAP_SYS_ADMIN`);
2. Флаг `no_new_privs` должен быть активирован, ограничивая процесс и все его дочерние процессы от получения новых привилегий.

Для удобства работы с BPF в библиотеке `linux/filter.h` языка Си представлена структура `sock_filter`, хранящая в себе код операции, количество инструкций, на которое должен быть совершен переход в случае, если условие верно или неверно, и операнд, а также макросы `BPF_STMT` и `BPF_JUMP`, делающие более читаемым настройку фильтра BPF.

Для демонстрации связи механизмов seccomp, BPF и ptrace подойдет следующая настройка фильтра:

```
BPF_STMT(BPF_LD | BPF_W | BPF_ABS, (offsetof(struct seccomp_data, nr))),
BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, SYS_openat, 1, 0),
BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW),
BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_TRACE),
```

Программа с установленным фильтром при полученном системном вызове проверяет номер системного вызова (поле `nr` в структуре `seccomp_data`), далее он сравнивается с переданным значением. В случае, если значение совпадает, необходимо пропустить 1 инструкцию, если нет, продолжить выполнение.

В случае, если получен системный вызов `openat` (открытие файла), фильтр возвращает значение `SECCOMP_RET_TRACE`, и ядро выполняет попытку оповестить трассировщик, что получен соответствующий системный вызов перед его выполнением.

Важно, что для подключения фильтра к трассировщику необходимо передать на вход `ptrace` флаг `PTRACE_O_SECCOMP`, иначе трассировщик не сможет получить уведомления о полученных системных вызовах.

5. Инструмент sbom-trace

5.1. Архитектура и используемые инструменты

Инструмент `sbom-trace` состоит из двух частей: трассировщика и постпроцессора. Из-за специфики реализации, а именно из-за работы с системными вызовами, `ptrace` API, механизмами `seccomp` и `BPF`, инструмент поддерживает генерацию базы данных компиляции только для операционных систем семейства Linux.

Для реализации трассировщика был выбран язык Си из-за его высокой производительности и более глубокого управления работой с памятью, в то время как постпроцессор реализован на языке Python в силу простоты реализации скрипта и удобной работы с форматом JSON.

Зависимостями проекта являются следующие библиотеки:

`rhash` — библиотека для хэширования функцией, определенной государственным стандартом ГОСТ Р 34.11-2012;

`uthash` — библиотека для работы с хэш-таблицами.

Задача трассировщика заключается в генерации промежуточного JSON-файла, состоящего из следующих полей:

- `directory` — рабочая директория сборки;
- `command` — полный список аргументов командной строки команды сборки;
- `component_commands` — массив записей, содержащих данные о каждой единице трансляции, состоит из:
    - `command` — список всех аргументов команды;
    - `dependencies` — список, состоящий из имён и хэш-сумм каждого включаемого файла;
    - `output` — список, состоящий из имён и хэш-сумм выходных файлов.

Постпроцессор получает на вход промежуточный файл, обрабатывает его и генерирует следующие поля:

- `input` — список, состоящий из имён и хэш-сумм каждой единицы трансляции;
- `utilities_info` — список, состоящий из инструментов, задействованных в трансляции исходных текстов в исполняемый код, а именно пути к их исполняемым файлам и их хэш-суммы.

5.2. Трассировщик

Для получения необходимой информации для базы данных компиляции трассировщик отслеживает следующие системные вызовы:

Табл. 1. Отслеживаемые системные вызовы

Table 1. Monitored system calls

Системный вызов	**Необходимая информация**
`open/openat/openat2`	Путь к файлу и режим, в котором он был открыт
`execve`	Список аргументов команды
`fork/vfork/clone`	Идентификатор нового процесса

Для отслеживания системных вызовов семейства `open` используется `seccomp`-фильтр со следующими настройками:

```
BPF_STMT(BPF_LD | BPF_W | BPF_ABS, (offsetof(struct seccomp_data, nr))),
#ifdef SYS_openat2
        BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, SYS_openat2, 3, 0),
#endif
    /* There is no open syscall on ARM64 architecture */
#ifdef SYS_open
        BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, SYS_open, 2, 0),
#else
        BPF_JUMP(BPF_JMP | BPF_JA, 1, 0, 0);
#endif
        BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, SYS_openat, 1, 0),
        BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW),
        BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_TRACE),
```

Для остановки процесса на остальных необходимых вызовах используются специальные флаги `ptrace`:

- `PTRACE_O_TRACEEXEC`;
- `PTRACE_O_TRACEFORK`;
- `PTRACE_O_TRACEVFORK`;
- `PTRACE_O_TRACECLONE`.

Для работы трассировщика были реализованы следующие структуры данных:

- Хэш-таблица для хранения информации о файлах: время последнего изменения файла (позволяет отследить изменение файла, открытого на чтение, вне процесса сборки), его хэш-сумма и канонический путь. Ключом в данной хэш-таблице является комбинация номера устройства и индексного дескриптора (inode), позволяющие уникально идентифицировать файл.
- Хэш-таблица, хранящая в себе информацию о процессе: его идентификатор и указатель на данные об образе процесса.
- Данные об образе процесса, состоящие из команды, запустившей образ, списка файлов, открытых на чтение и на запись, а также счётчик ссылок, позволяющий следить за количеством процессов, имеющих одинаковый образ.
- Массивы для хранения хэш-сумм и имён файлов.

Механизм работы трассировщика заключается в следующем:

1. При остановке на системном вызове `execve`, для процесса, из которого был получен вызов, создаётся новая запись об образе процесса и заполняется соответствующими данными, для предыдущего образа процесса счётчик ссылок уменьшается на 1.
2. При получении информации о вызове `fork` или `clone` в хэш-таблицу процессов добавляется новая запись, хранящая в себе указатель на образ родительского процесса, счётчик ссылок для соответствующего образа увеличивается на 1;
3. При остановке на `PTRACE_EVENT_STOP`, соответствующему в данном случае запуску нового процесса, порожденный процесс останавливается, если соответствующий `PTRACE_EVENT_FORK` еще не был получен (и будет разблокирован, когда идентификатор его родителя станет известным).
4. В момент открытия файла трассировщику передаётся информация о пути файла и режиме открытия файла, и в зависимости от режима информация о файле добавляется в соответствующий список файлов для текущего образа процесса, подсчитывается его хэш-сумма;
5. В момент окончания работы процесса из хэш-таблицы процессов удаляется запись и декрементируется счётчик ссылок соответствующего образа;
6. Если счётчик ссылок образа процесса после его уменьшения равен нулю, то

добавляется новая запись в базу данных компиляции в формате JSON.

В рамках исследования были проведены замеры времени сборки библиотек `musl` и `qtbase` с помощью компиляторов `gcc` и `clang` в трех различных условиях: с использованием трассировщика с хэшированием, с использованием трассировщика без функции хэширования, а также без применения трассировщика. Библиотека `musl` была выбрана как проект с большим количеством небольших по размеру единиц трансляции, а `qtbase` — как представитель класса крупных библиотек, написанных на языке C++. Эксперимент позволил оценить влияние процесса трассировки и хэширования на общую производительность сборки. Результаты представлены в таблице ниже.

Табл. 2. Результаты измерения производительности однопоточного трассировщика
Table 2. Performance evaluation results of a single-threaded tracer

Библиотека, компилятор и количество потоков	**Без трассировщика, сек**	**Трассировка без хэширования, сек**	**Трассировка с хэшированием, сек**
`musl`, `gcc`, 1 поток	20.03	24.7	27.23
`musl`, `clang`, 1 поток	33.14	39.38	41.66
`musl`, `gcc`, 8 потоков	4.01	7.35	9.21
`musl`, `clang`, 8 потоков	5.69	7.97	10.31
`qtbase`, `gcc`, 8 потоков	252.61	285.73	294.73

Для снижения процента замедления от использования трассировщика было принято решение оптимизировать процесс, реализовав многопоточный режим.

В многопоточном режиме вместо того, чтобы считать хэш-сумму файла сразу и хранить ее, в хэш-таблицу файлов передаётся порядковый номер хэша, подсчёт хэш-суммы запускается в новом побочном потоке, и работа трассировщика продолжается. После того, как хэш-сумма была подсчитана, она передаётся в общий для всех потоков массив хэш-сумм по индексу, равному порядковому номера хэша.

Так как текущая реализация трассировщика позволяет записывать данные в базу данных только в последовательном режиме, а в новом многопоточном режиме хэш-суммы могут быть неизвестны в момент записи информации в базу данных компиляции, было принято решение вместо хэш-сумм записывать в базу данные-заполнители в формате, соответствующем хэш-сумме, для последующего удобства подстановки в места, определенные заполнителями. Для сохранения мест, которые будут использованы для подстановки, был определен массив смещений в буфере, в котором для каждого файла хранится позиция заполнителя, соответствующего ему.

В момент, когда в базу данных компиляции выводится информация об открытых файлах для образа процесса, в общий массив смещений добавляется текущая позиция в буфере, хранящем в себе базу данных, и порядковый номер в формате, совпадающем с размером хэш-суммы. Полученный механизм позволяет после окончания трассировки подставить все хэш-суммы в необходимые места за один полный проход по массиву смещений.

Для оценки эффективности многопоточного режима были проведены повторные замеры времени сборки тех же библиотек в модифицированных условиях. Результаты представлены в таблице ниже.

Табл. 3. Результаты измерения производительности для многопоточного трассировщика
Table 3. Performance evaluation results of a multi-threaded tracer

Библиотека, компилятор и количество потоков	**Без трассировщика, сек**	**Трассировка с параллельным хэшированием, сек**
`musl`, `gcc`, 1 поток	20.03	25.28

Библиотека, компилятор и количество потоков	**Без трассировщика, сек**	**Трассировка с параллельным хэшированием, сек**
`musl`, `clang`, 1 поток	33.14	39.59
`musl`, `gcc`, 8 потоков	4.01	8.45
`musl`, `clang`, 8 потоков	5.69	8.55
`qtbase`, `gcc`, 8 потоков	252.61	288.87

В результате внедрения многопоточного хэширования в процесс трассировки было достигнуто ускорение. Различия во времени выполнения с трассировщиком между однопоточной и многопоточной сборками демонстрируют, что многопоточность эффективно сокращает время компиляции, но преимущества этого снижаются из-за однопоточной обработки системных вызовов механизмом `ptrace`. На однопоточной сборке выбранных проектов разработанный инструмент демонстрирует замедление от 19% до 26%, в то время как на многопоточной сборке происходит замедление от 14% до 110% в худшем случае.

Одной из ключевых проблем, выявленных при реализации трассировщика, является преждевременный подсчёт хэш-сумм содержимого выходных файлов. Проблема заключается в том, что при остановке процесса на системном вызове семейства `open`, когда файл открывается на запись, его содержимое ещё не успело быть записано. В результате, если хэширование начнётся в этот момент, оно приведёт к некорректным результатам, поскольку содержимое файла ещё не сформировано.

Для решения этой проблемы был внедрён новый механизм вычисления хэш-сумм содержимого выходного файла. При получении информации об открытии выходного файла, его данные заносятся в таблицу файлов с флагом `hash_calculated`, установленным в значение `false`. Если позже тот же файл появляется как входной, это указывает на завершение всех операций над ним, и, соответственно, можно приступать к вычислению его хэша.

Хэш-суммы, которые не были рассчитаны в процессе, считаются отложенными и будут вычислены в функции `calculate_pending_hashes`. В этой функции обрабатываются две категории файлов: артефакты сборки (исполняемый файл, полученный в процессе сборки и побочные выходные файлы) и временные файлы, которые не были прочитаны и затем удалены (например, `.res` файл при компиляции программы на языках Си/C++ компилятором GCC без флага `-flto`). Поскольку функция `calculate_pending_hashes` вызывается после завершения трассировки, то хэш-суммы артефактов сборки могут быть безопасно рассчитаны, в то время как непрочитанные временные файлы могут быть проигнорированы.

5.3. Постпроцессор

Трассировщик в его текущей реализации не позволяет генерировать множество входных файлов во время работы, а также не предоставляет гибкости как для опционального множества входных файлов, так и для получения информации об используемых утилитах. В связи с этим было принято решение использовать дополнительный скрипт, который получает на вход промежуточный файл и обрабатывает его, генерируя оставшиеся поля.

Постпроцессор и трассировщик связаны между собой следующим образом: по завершению работы трассировщика создаётся канал (`popen()`), запускается скрипт постпроцессора, и полученный в процессе трассировки JSON-файл передаётся в созданный канал. Постпроцессор, в свою очередь, получает этот JSON-файл через стандартный ввод (`stdin`) и обрабатывает его.

Первым шагом обработки JSON-файла, полученного из трассировщика, является удаление выходных файлов с неподсчитанной хэш-суммой: на данном этапе удаляются записи о временных файлах, которые не были прочитаны в процессе сборки.

Далее происходит генерация множества входных файлов: по умолчанию множеством входных файлов считаются все файлы, которые были открыты на чтение и не были открыты на запись в процессе трассировки.

Последним шагом в работе постпроцессора является сбор информации об использованных инструментах: по умолчанию в базу данных компиляции вносится информация о пути к исполняемому файлу инструмента и хэш-сумма его содержимого, однако в программном коде скрипта представлены функции, в данный момент не содержащие в себе механизм извлечения какой-либо информации, но позволяющие в дальнейшем поместить в базу данных еще версию инструмента и информацию об его конфигурационных файлах.

6. Заключение

В рамках данной работы были исследованы методы идентификации реквизитов сборки и существующие инструменты для генерации базы данных компиляции. Был реализован инструмент для идентификации реквизитов сборки с помощью отслеживания системных вызовов. Инструмент состоит из двух компонентов: трассировщика системных вызовов и постпроцессора, обрабатывающего промежуточную базу данных компиляции, полученную из трассировщика. Чтобы снизить процент замедления процесса сборки при использовании трассировщика, был реализован механизм многопоточного подсчета хэш-сумм содержимого файлов. Были подобраны проекты для оценки времени работы их сборочного процесса в обычной ситуации и с использованием трассировщика.

Разработанный инструмент предоставляет информацию о каждом этапе компиляции, командах компиляции, зависимостях и выходных файлах, при этом на выбранных для эксперимента проектах замедляет процесс сборки в процентном соотношении от 19% до 26% в случае однопоточной сборки, а при многопоточной сборке — от 14% до 110% в зависимости от объема проекта.

Список литературы / References

[1]. ГОСТ Р 71206-2024 «Защита информации. Разработка безопасного программного обеспечения. Безопасный компилятор языков С/С++. Общие требования». М., Российский институт стандартизации, 2024, 20 с.

[2]. Clang Static Analyzer, Available at: https://clang-analyzer.llvm.org/, accessed 28.04.2024.

[3]. Baloglu B. How to find and fix software vulnerabilities with coverity static analysis //2016 IEEE Cybersecurity Development (SecDev). – IEEE, 2016. – С. 153-153.

[4]. А.А. Белеванцев, А.О. Избышев, Д.М. Журихин. Организация контролируемой сборки в статическом анализаторе Svace. Системный администратор, выпуск 6-7 (176-177), 2017, стр. 135-139.

[5]. Build EAR, Available at: https://github.com/rizsotto/Bear, accessed 28.04.2024.

[6]. Compilation Database Generator, Available at: https://github.com/nickdiego/compiledb, accessed 28.04.2024.

[7]. Guo P. CDE: A tool for creating portable experimental software packages //Computing in Science & Engineering. – 2012. – Т. 14. – №. 4. – С. 32-35.

[8]. JSON Compilation Database Format Specification, Available at: https://clang.llvm.org/docs/JSONCompilationDatabase.html, accessed 28.04.2024.

[9]. Баев Р.В., Скворцов Л.В., Кудряшов Е.А., Бучацкий Р.А., Жуйков Р.А. Предотвращение уязвимостей, возникающих в результате оптимизации кода с неопределенным поведением. Труды ИСП РАН, том 33, вып. 4, 2021 г., стр. 195-210. DOI: 10.15514/ISPRAS–2021–33(4)–14./ Baev R.V., Skvortsov L.V., Kudryashov E.A., Buchatskiy R.A., Zhuykov R.A. Prevention of vulnerabilities arising from optimization of code with Undefined Behavior. Trudy ISP RAN/Proc. ISP RAS, vol. 33, issue 4, 2021. pp. 195-210 (in Russian). DOI: 10.15514/ISPRAS–2021–33(4)–14.

[10]. GCC, Available at: https://gcc.gnu.org/, accessed 28.04.2024.
[11]. Vangoor B. K. R., Tarasov V., Zadok E. To FUSE or not to FUSE: Performance of User-Space file systems //15th USENIX Conference on File and Storage Technologies (FAST 17). – 2017. – С. 59-72.
[12]. Love R. Kernel korner: Intro to inotify //Linux Journal. – 2005. – Т. 2005. – №. 139. – С. 8.
[13]. Keniston J. et al. Ptrace, utrace, uprobes: Lightweight, dynamic tracing of user apps //Proceedings of the 2007 Linux symposium. – 2007. – Т. 1. – С. 215-224.
[14]. Padala P. Playing with ptrace, Part I //Linux Journal. – 2002. – Т. 103. – №. 5.
[15]. Padala P. Playing with ptrace, part II //Linux J. – 2002. – Т. 104. – С. 4.
[16]. Ptrace Notes, Available at: https://shachaf.net/tmp/ptrace-notes.txt, accessed 28.04.2024.

Информация об авторах / Information about authors

Артемий Максимович ГРАНАТ — Лаборант Института системного программирования им. В.П. Иванникова Российской академии наук, студент 4 курса бакалавриата Саратовского государственного университета. Сфера научных интересов: компиляторы, операционные системы, компьютерные сети.

Artemiy Maksimovich GRANAT — Laboratory technician at Ivannikov Institute for System Programming of the Russian Academy of Sciences, 4th year Bachelor's student at Saratov State University. Research interests: compilers, operating systems, computer networks.

Павел Дмитриевич ДУНАЕВ — Старший лаборант Института системного программирования им. В.П. Иванникова Российской академии наук, студент 2 курса магистратуры Саратовского государственного университета. Сфера научных интересов: компиляторы, операционные системы, дискретная математика.

Pavel Dmitrievich DUNAEV — Senior Laboratory technician at Ivannikov Institute for System Programming of the Russian Academy of Sciences, 2nd year Master's student at Saratov State University. Research interests: compilers, operating systems, discrete mathematics.

Артем Александрович СИНКЕВИЧ — Старший лаборант Института системного программирования им. В.П. Иванникова Российской академии наук, студент 1 курса магистратуры Саратовского государственного университета. Сфера научных интересов: компиляторы, дискретная математика, нейронные сети.

Artem Aleksandrovich SINKEVICH — Senior Laboratory technician at Ivannikov Institute for System Programming of the Russian Academy of Sciences, 1st year Master's student at Saratov State University. Research interests: compilers, discrete mathematics, neural networks.

Инна Александровна БАТРАЕВА — кандидат физико-математических наук, доцент, заведующая кафедрой технологий программирования. Сфера научных интересов: дискретная математика, теория автоматов, теория формальных языков и грамматик, информационные системы в теоретической и прикладной лингвистике.

Inna Aleksandrovna BATRAEVA — Candidate of Science in Physics and Mathematics, Associate Professor, Head of the Department of Programming Technologies. Research interests: discrete mathematics, automata theory, theory of formal languages and grammars, information systems in theoretical and applied linguistics.

Дмитрий Юрьевич ПЕТРОВ — кандидат технических наук, старший научный сотрудник лаборатории системных проблем управления и автоматизации в машиностроении; доцент кафедры системного анализа и автоматического управления Саратовского национального исследовательского государственного университета имени Н.Г.Чернышевского. Сфера научных интересов: системный анализ, системотехника на основе моделей, функциональное и имитационное моделирование, авионика и автоматизированные системы управления.

Dmitriy Yurievich PETROV — Candidate of Science in Engineering, Senior Researcher of the Laboratory of System Problems of Control and Automation in Mechanical Engineering; Associate Professor of the Department of System Analysis and Automatic, Saratov State University. Research interests: system analysis, model-based system engineering, functional and simulation modeling, avionics and automated control systems.