МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**
Кафедра математической кибернетики и компьютерных наук

# ИДЕНТИФИКАЦИЯ РЕКВИЗИТОВ СБОРКИ ЧЕРЕЗ ОТСЛЕЖИВАНИЕ СИСТЕМНЫХ ВЫЗОВОВ

АВТОРЕФЕРАТ БАКАЛАВРСКОЙ РАБОТЫ

студента 4 курса 451 группы
направления 09.03.04 — Программная инженерия
факультета КНиИТ
Граната Артемия Максимовича

Научный руководитель
зав. каф. техн. пр., доц., к. ф.-м. н. _____ И. А. Батраева

Заведующий кафедрой
доцент, к. ф.-м. н. _____ С. В. Миронов

Саратов 2024

## СОДЕРЖАНИЕ

# 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 С/С++. Общие требования» [**?**], направленный на установление единых требований к безопасному компилятору языков Си и С++. Одним из пунктов стандарта является функция журналирования процесса трансляции, с сохранением такой информации, как параметры компиляции и хэш-суммы входных и выходных файлов, при этом для ведения базы данных компиляции должен использоваться текстовый формат обмена данными JSON (RFC 8259).

Несмотря на то, что стандарт ориентирован на языки Си и С++, 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.

Целью данной работы является реализация инструмента для идентифика-

ции реквизитов сборки, оценка времени его работы и сравнение с имеющимися решениями.

В результате написания работы должны быть решены следующие задачи:

- исследование и описание методов идентификации реквизитов сборки;
- реализация инструмента для идентификации реквизитов сборки с помощью выбранного метода;
- реализация методов оценки времени работы инструмента в различных режимах;
- оценка времени работы сборочного процесса на выбранных проектах.

**Структура и объём работы.** Бакалаврская работа состоит из введения, двух разделов, заключения, списка использованных источников и двух приложений. Общий объем работы — 77 страниц, из них 44 страницы — основное содержание, список использованных источников информации — 21 наименование.

# 1 Журналирование процесса трансляции

## 1.1 Описание проблемы

### 1.1.1 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, Coverity или Svace, 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.

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.

### 1.1.2 Обзор существующих инструментов

Для сбора параметров компиляции и генерации файла, содержащего информацию о командах компиляции, применяемой в различных инструментах анализа кода, используется утилита `Bear`. Она перехватывает вызовы компилятора во время сборки проекта и создаёт соответствующий файл. Помимо `Bear`, также применяется, например, `compiledb`, подходящая для сборочных систем, основанных на Makefile.

Кроме того, сборочные системы, такие как `CMake`, `Ninja`, `Qt Build System`

и qmake, также способны автоматически генерировать файлы конфигурации, содержащие информацию о параметрах компиляции.

CDE 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. В безопасном компиляторе «SAFEC», разработанном в Институте системного программирования РАН на основе GCC, также добавлена функциональность генерации базы данных компиляции.

Однако, одними из основных ограничений генерации базы данных с помощью внедрения дополнительной функциональности в компилятор являются сложность отслеживания всех внешних зависимостей при использовании сложных сценариев сборки или нестандартных инструментов, а также привязка к определенным языкам программирования.

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

### 1.1.3 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.
2. inotify — это механизм ядра Linux, который позволяет отслеживать изменения в файловой системе.
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.

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.

## 1.2 Системные вызовы

Системный вызов — это интерфейс для обращения прикладного программного обеспечения к ядру операционной системы, который позволяет приложениям выполнять различные операции, требующие привилегий ядра ОС.

Системные вызовы обычно вызываются не напрямую, а через API, которые предоставляют различные языки программирования и инкапсулируют системные вызовы в более высокоуровневый интерфейс. Например, стандартная библиотека языка программирования C позволяет открыть файл через системный вызов `open()` напрямую, так и через обёртку `fopen()`.

Системные вызовы можно разделить на несколько категорий:

1. Управление процессами;
2. Работа с файловой системой;
3. Управление памятью;
4. Управление устройствами;
5. Управление сетевыми операциями.

Системный вызов идентифицируется именем соответствующей ему операции, помимо этого, у каждого из них есть свой номер, который передается в момент вызова через определённый регистр.

## 1.3 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` — сама функция для системного вызова и перечисление возможных запросов `__ptrace_request`, необходимых для взаимодействия с трассируемым процессом, получением необходимой информации и управления его выполнением.

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*);
3. Трассировщик перезапускает отслеживаемый процесс одним из способов перезапуска (PTRACE_CONT, PTRACE_SYSCALL, PTRACE_SINGLESTEP).

### 1.4 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 в виде 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.

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.

## 2 Разработка инструмента sbom-trace

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.

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

— Рабочая директория сборки;
— Полный список аргументов командной строки команды сборки;
— Массив записей, содержащих данные о каждой единице трансляции, состоит из:
  – Список всех аргументов команды;
  – Список, состоящий из имён и хэш-сумм каждого включаемого файла;
  – Список, состоящий из имён и хэш-сумм выходных файлов.

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

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

### 2.1 Архитектура трассировщика

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

Таблица 2.1 – Отслеживаемые системные вызовы

| 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 |

### 2.2 Инициализация трассировщика и обработка сигналов

Трассировщик при запуске принимает на вход три опции:

— `postprocessor` — обязательная опция, параметр — путь до постпроцессора;
— `dump-sbom` — необязательная опция, параметр — директория для выходного JSON-файла;
— `filter-subtree` — необязательная опция, запускает постпроцессор для генерации множества входных файлов, параметром является директория, внутри которой находятся входные файлы сборки.

После парсинга аргументов командной строки трассировщик отделяет команду сборки и вызывает функцию `fork()` для исполнения полученной команды в новом процессе.

Для отслеживания системных вызовов семейства `open` в процессе, исполняющем команду сборки, используется `seccomp`-фильтр.

В трассировщике запускается цикл, продолжающий свою работу до момента полной остановки работы процесса и всех его детей. Для отслеживания количества детей используется счетчик `child_counter`, увеличивающийся на 1 при получении одного из системных вызовов `fork`, `vfork`, `clone`, и уменьшающийся при смерти процесса. Также обрабатываются все случаи, помимо остановки на системных вызовах.

### 2.3 Обработка системного вызова execve

Получение события `PTRACE_EVENT_EXEC` означает, что была исполнена одна из команд сборки, что в свою очередь означает создание нового образа процесса.

Функция `tracer_handle_execve` получает идентификатор процесса и считывает исполняемую команду из файла `/proc/PID/cmdline`, содержащего в себе аргументы командной строки, разделенные между собой нулевым символом, запускает обработчик образа процесса `tracer_handle_image_init`.

Также для дальнейшей обработки информации об используемых инструментах необходимо сохранять путь до исполняемого файла. Данный путь хранится в файле `/proc/PID/exe`, что позволяет считать путь с помощью вызова `readlink` на этот файл.

### 2.4 Обработка открытия файлов

Событие `PTRACE_EVENT_SECCOMP` означает, что из `seccomp`-фильтра было получено оповещение о системном вызове, указанном в фильтре `BPF`. В данном

случае это означает, что был получен один из системных вызовов open, openat, openat2.

В функции tracer_handle_syscall трассировщик получает значения регистров с помощью запроса PTRACE_GETREGS после завершения системного вызова, получает номер системного вызова из регистра orig_rax и код возврата из регистра rax. Если возвращаемое значение системного вызова меньше нуля, в данной ситуации это означает, что в процессе сборки произошла попытка открытия несуществующего файла, а значит, этот вызов можно пропустить. В случае, если был открыт файл, вызывается функция tracer_handle_openat.

Системные вызовы семейства open возвращают номер открытого файлового дескриптора в качестве возвращаемого значения. Информацию о файле можно получить с помощью вызова stat по пути /proc/PID/fd/NUMBER. Если открытый файл является не обычным, а, например, сокетом или символьной ссылкой, то файл игнорируется.

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.

### 2.5 Обработка создания процесса

Далее в главном цикле обработки ptrace-остановок обрабатываются события PTRACE_EVENT_FORK/VFORK/CLONE.

Функция tracer_handle_fork работает следующим образом:

1. Запрос PTRACE_GET_EVENTMSG с указанием идентификатора родительского процесса возвращает идентификатор нового процесса;
2. Так как при вызовах fork, vfork и clone новый процесс хранит в себе образ родительского процесса, то при создании записи в таблице процессов указатель на образ процесса приравнивается к указателю на образ процесса родительской записи, увеличивается счетчик ссылок на родительский

образ.

Когда трассируемый процесс создает новый процесс, который в дальнейшем должен исполнить команду, происходит следующее:

1. Родительский процесс вызывает `fork`, `vfork` или `clone`, посылает событие `PTRACE_EVENT_FORK/VFORK/CLONE` трассировщику;
2. Дочерний процесс останавливается, посылает трассировщику событие `PTRACE_EVENT_STOP`, ожидая ответа от трассировщика, после получения ответа, возможно, вызывает `execve()` и в случае, если системный вызов был успешным, посылает трассировщику `PTRACE_EVENT_EXEC`.

### 2.6 Вывод информации об инструментах и вызов постпроцессора

После окончания трассировки в промежуточный JSON-файл выводится путь до каждого использованного инструмента, а также его хэш-сумма. Аналогично с механизмом вывода зависимостей, был реализован вывод уникальных путей до инструментов.

После вывода информации об используемых инструментах, трассировщик в зависимости от полученных аргументов командной строки формирует команду запуска постпроцессора и вызывает её с помощью функции `popen`.

### 2.7 Многопоточный режим трассировщика

В рамках исследования были проведены замеры времени сборки библиотек `musl` и `qtbase` с помощью компиляторов `gcc` и `clang` в трех различных условиях. Замеры проводились на виртуальном сервере с операционной системой Ubuntu 24.04 LTS, 8-ядерным процессором с частотой 3.7 ГГц и 16 ГБ оперативной памяти. Результаты измерений представлены в таблице ниже.

Таблица 2.2 – Результаты измерения производительности однопоточного трассировщика

| Библиотека, компилятор и количество потоков | Без трассировщика, сек | Трассировка без хэширования, сек | Трассировка с хэшированием, сек |
|---|---|---|---|
| `musl`, `gcc`, 1 поток | 20,03 | 24,70 | 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 с использованием библиотеки `pthread`.

Для оценки эффективности многопоточного режима были проведены повторные замеры времени сборки тех же библиотек в модифицированных условиях. Результаты представлены в таблице 2.3.

Таблица 2.3 – Результаты измерения производительности для многопоточного трассировщика

| Библиотека, компилятор и количество потоков | Без трассировщика, сек | С трассировщиком, сек |
|---|---:|---:|
| `musl`, `gcc`, 1 поток | 20,03 | 25,28 |
| `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.

## 2.8 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 работы постпроцессора является инициализация класса `Postprocessor`. Проверяется существование директорий с именами, переданными через флаги `-d` и `-f`, если они были переданы, далее выставляются значения полей класса и из потока `stdin` считывается JSON-файл:

Далее начинается обработка JSON-файла, полученного из трассировщика. Первым её шагом является удаление выходных файлов с неподсчитанной хэш-суммой: на данном этапе удаляются записи о временных файлах, которые не были прочитаны в процессе сборки. Данный тип файлов можно идентифицировать по его хэш-сумме в JSON-файле — если хэш не был подсчитан, то в промежуточном файле она будет обозначена как последовательность, состоящая только из нулей.

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. В случае, если переменная `filter_subtree_dir` имеет значение, отличное от `None`, то к множеству входных файлов добавляется еще одно условие — файл должен находиться в директории `filter_subtree_dir` или в любой из её поддиректорий.

Следующим 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, что происходит на этапе работы трассировщика, 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.

Последний шаг в работе постпроцессора — сохранение базы данных компиляции в файл, названием файла в данном случае является хэш-сумма базы данных компиляции, подсчитанная с помощью инструмента командной строки `rhash` и хэш-функции, определенной государственным стандартом ГОСТ Р 34.11-2012.

# 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.

В ходе написания работы были решены следующие задачи:

- изучены и описаны методы идентификации реквизитов сборки;
- реализован инструмент для идентификации реквизитов сборки с помощью выбранного метода;
- реализованы методы оценки времени работы и качества поиска;
- оценено время работы сборочного процесса на выбранных проектах.

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.