

Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 22.Б15-мм

Сбор статистики для системы мониторинга СИМВОЛЬНОГО ИСПОЛНЕНИЯ

Дьячков Виталий Иванович

Отчёт по производственной практике
в форме «Производственное задание»

Научный руководитель:
Кандидат физико-математических наук, Д. А. Мордвинов

Санкт-Петербург
2025

Оглавление

Введение	3
1. Постановка задачи	5
2. Обзор	6
2.1. Символьное исполнение	6
2.2. Инструмент KLEEF	6
2.3. Инструмент USVM	7
2.4. Требования	7
2.5. Используемые технологии	8
3. Архитектура	9
3.1. Общий шаблон работы	9
3.2. Архитектура агента для инструмента KLEEF	9
3.3. Архитектура агента для инструмента USVM	11
4. Реализация	13
4.1. Формат данных	13
4.2. Реализация для инструмента KLEEF	13
4.3. Реализация для инструмента USVM	14
5. Апробация	16
5.1. Описание тестового стенда	16
5.2. Результаты апробации для инструмента KLEEF	16
5.3. Результаты апробации для инструмента USVM	17
Заключение	18
Список литературы	19

Введение

Для разработчика важно отмечать эффективность работы его кода и при этом экономить время. Для того, чтобы иметь возможность в реальном времени отслеживать то, как тестируется его код, необходима система мониторинга символьного исполнения.

Важный этап разработки программного обеспечения – проверка его корректности. Существующие методы тестирования, несмотря на их многообразие, не могут гарантировать полное покрытие всех возможных сценариев и требуют значительных временных затрат. Символьное исполнение, позволяющее анализировать все пути выполнения программы и находить все входные данные, ведущие к ним, предлагает эффективный подход к автоматической верификации. На данный момент есть несколько инструментов, предоставляющих возможности автоматического анализа программ на возможные уязвимости и ошибки, например: KLEE[2], Owi[4], S2E[6] и другие.

Однако, для разработчика критически важна не только сама возможность анализа, но и понимание эффективности процесса. Экономия времени может достигаться не только за счёт автоматизации, но и за счет анализа информации, полученной от символьного исполнителя. Именно поэтому система мониторинга символьного исполнения, собирающая такую информацию, становится удобным подспорьем для разработчика.

Эффективная система мониторинга должна предоставлять разработчику: статистику покрытия кода, количество исполненных инструкций и ветвлений, время выполнения, обнаруженные ошибки и уязвимости, возможность отслеживать изменение статистик в реальном времени.

С помощью такой системы мониторинга разработчик сможет получить всю информацию о ходе символьного исполнения, а также принимать обоснованные решения по оптимизации процесса, ускорению разработки и повышению качества программного обеспечения. Без сбора и анализа статистических данных, символьное исполнение, хотя и мощ-

ное, но остается «черным ящиком», не дающим достаточно информации для эффективного использования.

1. Постановка задачи

Целью работы является сбор статистики для системы мониторинга символьного исполнения. Для этого были поставлены следующие задачи:

1. Выполнить обзор символьного исполнения, инструментов KLEEF, USVM.
2. Разработать архитектуру системы сбора статистики для KLEEF и USVM.
3. Реализовать систему мониторинга для инструментов KLEEF и USVM.
4. Провести апробацию полученных решений.

2. Обзор

Этот раздел представляет собой перечисление основных терминов, необходимых для понимания этой работы, обзор предметной области, существующих решений и более детальный взгляд на KLEEF и USVM.

2.1. Символьное исполнение

Перед тем, как начать погружаться в цель работы, необходимо общее понимание того, зачем эта работа нужна.

Символьное исполнение[7] — это метод анализа программного кода, который выполняется не над конкретными значениями (как при обычном исполнении), а над символьными значениями. Это значит, что, вместо того чтобы подставлять конкретные числа или строки, программа обрабатывает символьные переменные, представляющие любые возможные значения. В процессе символьного исполнения, программа исследует различные пути выполнения кода, генерируя условия пути в виде логических выражений, которые определяют, когда программа идет по определенному пути. Эти условия пути могут затем быть решены с помощью SMT-решателей для получения конкретных входных данных, которые приводят к выполнению данного пути.

У символьного исполнения довольно широкая область применения, например: поиск ошибок в программном обеспечении[8], обнаружение уязвимостей, автоматическая генерация тестов[9], генерация краевых значений, проверка соответствия спецификации, проверка безопасности. Но при этом символьное исполнение обладает своими проблемами, например: взрыв путей, зависимость от окружения, работа с указателями и структурами данных, решение которых ищут и по сей день.

2.2. Инструмент KLEEF

В Стэнфордском университете разрабатывается KLEE[2]. Он предназначен для анализа программ на языке C и C++. KLEE использует технику белого ящика для исследования путей выполнения, то есть он

требует доступа к исходному коду программы. На его основе, командой R&D Toolchain labs[5] был создан KLEEF[3]. KLEEF — усовершенствованная версия классического механизма символьного исполнения с дополнительными функциями, такими как: обратное символьное исполнение, двунаправленное символьное исполнение, интегрированный фазер, ленивая инициализация, массивы символьного размера. В соревновании TestComp24 KLEEF превзошел KLEE, заняв третье место в категории overall[10].

2.3. Инструмент USVM

USVM — символьный движок, также разрабатываемый командой R&D Toolchain labs, который поддерживает различные языки. В рамках этой работы предлагается реализация для версии с языком Java, но в последствии, возможно добавление и в другие языки.

Основные функции, которые предоставляет USVM:

- расширяемая и высокооптимизированная модель символьной памяти;
- оптимизированное управление ограничениями для снижения нагрузки на SMT-решатель;
- улучшенные символические модели для контейнеров;
- символическое исследование в прямом и обратном направлении;
- решение ограничений на уровне типов без использования SMT-решателя.

2.4. Требования

Познакомившись с областью работы, можно сформировать требования к реализации:

- статистика должна собираться правильно;

- статистика должна отправляться на сервер с небольшим периодом;
- сбор статистики не должен влиять на корректность символьного исполнения;
- сбор статистики не должен сильно замедлять процесс символьного исполнения.

2.5. Используемые технологии

Этот раздел представляет собой список технологий и библиотек, который будут использованы во время реализации поставленной задачи.

2.5.1. Используемые технологии для KLEEF

В процессе реализации сбора статистики для KLEEF были использованы следующие технологии: C++, GCC, Nlohmann, Libcurl, CMake.

2.5.2. Используемые технологии для USVM

Во время работы над созданием системы сбора статистики для USVM было использовано следующее: Kotlin, Gradle, Kotlinx.atomicfu, HttpURLConnection, Kotlinx.serialization.

3. Архитектура

Этот раздел представляет собой введение в текущую архитектуру инструментов KLEEF и USVM, описание общего принципа работы, а также разработанную архитектуру, предложенную для реализации.

3.1. Общий шаблон работы

Во время запуска символьного исполнения, должна запускаться система сбора статистики, называемая агентом. Агент должен запускаться в отдельном потоке и раз в какое-то время собирать текущее значение метрик о работе символьного исполнения.

В архитектуре системы символьного исполнения предусмотрен автономный агент сбора статистических данных, функционирующий в режиме параллельного процесса. Данный агент, реализуемый в отдельном потоке исполнения, предназначен для периодической фиксации значений ключевых метрик, характеризующих текущее состояние и прогресс процесса символьного исполнения. С целью сохранения целостности процесса символьного исполнения и минимизации накладных расходов, агент работает обособленно, осуществляя сбор данных через определенные дискретные интервалы времени. Применение такого подхода обеспечивает адекватный и корректный сбор данных, не оказывающий влияния на логику работы символьного движка, позволяя получить объективную картину происходящих процессов.

3.2. Архитектура агента для инструмента KLEEF

Для начала сформируем абстрактное представление об архитектуре инструмента:

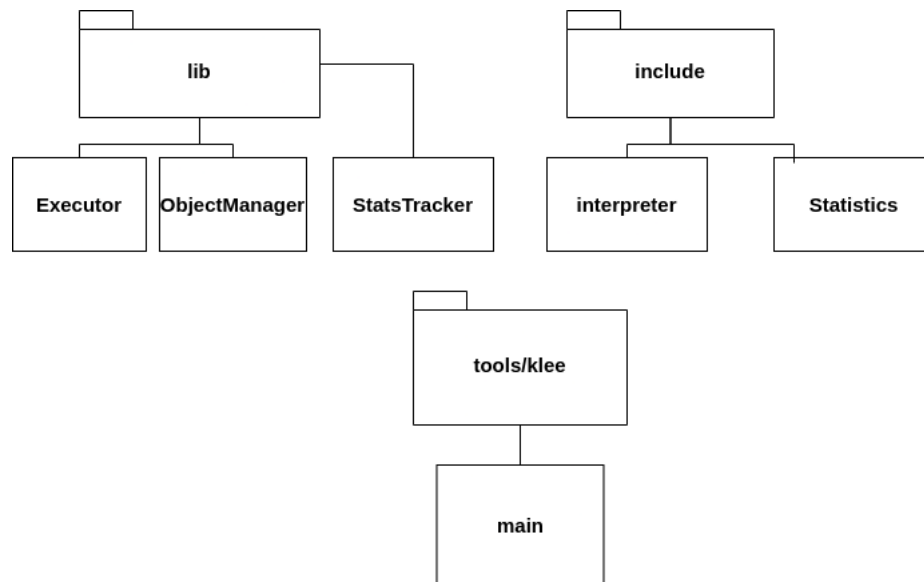


Рис. 1: Абстрактное представление о архитектуре сбора статистики KLEEF

Первый, **lib**, модуль, представляющий собой основной компоненты, необходимые для символического исполнения, например: **Executor**, **ObjectManager**, **StatsTracker** и другие. Второй собирает в себе список классов и методов, которые в последствии будут использоваться в **lib**. Последний, в свою очередь, содержит в себе функцию **main** и является отправной точкой, формирует данные для запуска, инициализирует всё необходимое и запускает непосредственно инструмент.

В KLEEF модуль, отвечающий за сбор статистики, содержит в себе классы **Statistic**, **StatisticRecord** и **StatisticManager**. Где **Statistic** представляет собой базовую единицу и содержит имя статистики и возможность вернуть по нему значение. **StatisticRecord** хранит в себе данные о статистике и её имя, **StatisticManager** хранит в себе все статистики и позволяет с ними взаимодействовать. Основной рабочий модуль - **Executor**, который запускает символическое исполнение, и сохраняет статистику. На данном этапе вывод статистики доступен только после окончания анализа поступающего на вход файла.

В процессе анализа используется класс **CallPathNode**, который содержит в себе информацию о вызываемых функциях и хранит о них статистику, и **CallPathManager**, который собирает в себе все **CallPathNode**

и предоставляет возможности взаимодействия с ними.

Для корректной работы нужно добавить классы, которые отвечают за сбор и сериализацию статистики, отправку её на сервер, а также класс, который будет отвечать за синхронизацию данных между потоками. Исходя из этого, можно сформировать примерное представление о необходимых классах, которое будет выглядеть так:

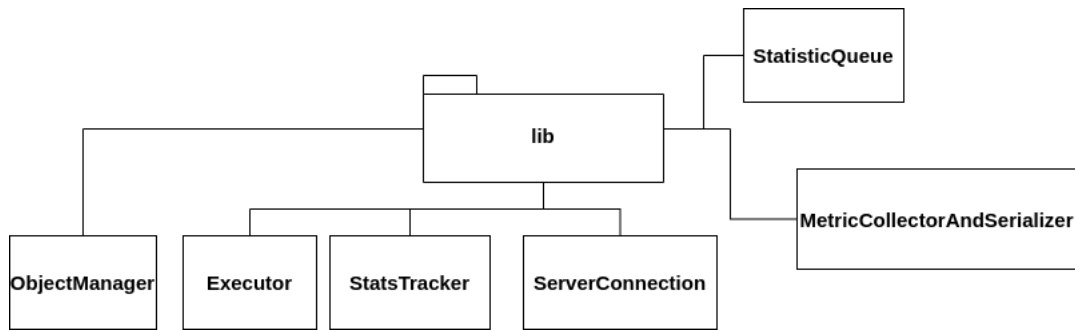


Рис. 2: Список классов, необходимых для добавления

3.3. Архитектура агента для инструмента USVM

Для начала, нам также необходимо образно понять архитектуру USVM. Список основных модулей:

- **usvm-dataflow** — компонент, который выполняет статический анализ программ с использованием IFDS и Tainted[1];
- **usvm-jvm** — реализация символьного анализа для Java;
- **usvm-core** — предоставляет базовую функциональность.

Основные интерфейсы и классы необходимые для реализации работы сбора статистики:

- **UMachine** — интерфейс предоставляющий возможность сбора запуска символьного исполнения;
- **UMachineObserver** — интерфейс дающий возможность реализации некоего слушателя, реагирующего на какие-то события;

- **TimeStatistics** — класс отвечающий за сбор статистики о времени исполнения;
- **StepsStatistics** — класс отвечающий за сбор статистики о количестве исполненных шагов;
- **CoverageStatistics** — класс дающий доступ к текущей зоне покрытия;
- **JcMachine** — реализация **UMachine** для языка Java;

Внутри JcMachine собираются все инстанты классов для сбора статистики и запускаются из метода run. Подобный подход позволяет при небольших временных затратах создать реализацию символьного исполнения для множества других языков.

Для реализации поставленной задачи, необходимо реализовать лишь класс **Agent**, наследованный от **UMachineObserver**, который будет содержать внутри себя все необходимые методы, а именно: метод для сбора статистики, метод для её сериализации, внутренний класс для отправки данных на удаленный сервер, а также логику синхронизации с процессом символьного исполнения:

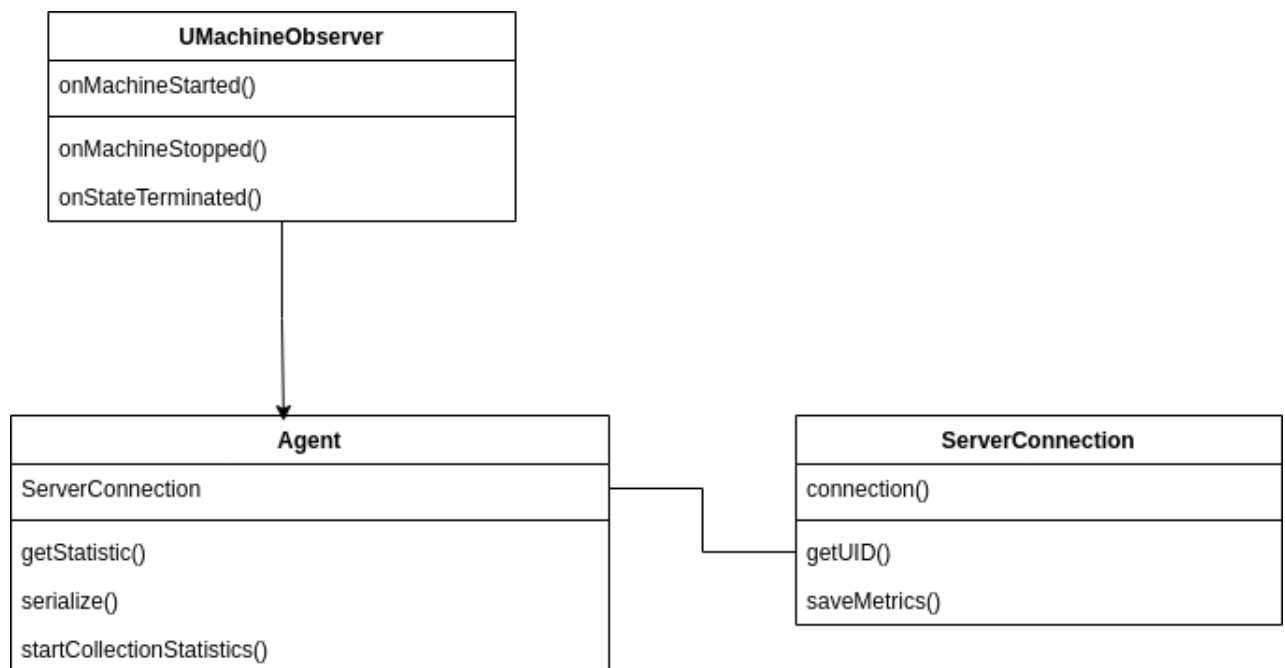


Рис. 3: Архитектура агента

4. Реализация

Этот раздел содержит в себе описание формата данных, отправляемых на сервер, а также детали реализации системы мониторинга в инструментах KLEEF, USVM.

4.1. Формат данных

Конечный сформированный формат данных должен отправляться на сервер с установленным промежутком времени и представлять собой Json, который продемонстрирован на рисунке ниже.

```
{
  "guid": "1234"
  "name": "MetricName",
  "params": {
    "funName": "name",
    "type": "<type>",
    "value": "<some value of the type>"
    "transitive": false
  }
}
```

Рис. 4: Формат данных

Здесь **guid** — уникальный идентификатор сессии, **name** — имя метрики, и вложенный **params** — данные о функции и значение метрики.

4.2. Реализация для инструмента KLEEF

Для корректной работы сбора статистики в реальном времени был изменен основной интерпретирующий цикл в **Executor**, добавлен класс **ServerConnection**, **StatisticQueue** и **MetricCollectorAndSerializer**. Рассмотрим каждое изменение по отдельности:

- **ServerConnection** — класс предоставляющий основные методы работы с http запросами, а именно получение уникального id сессии и сохранение метрик;

- **StatisticQueue** — чтобы избежать гонки данных, появилась необходимость реализации очереди с синхронизацией, для правильного сбора статистики;
- **MetricCollectorAndSerializer** — сущность отвечающая за получение статистики, её накопление и сериализацию;
- **Изменение главного цикла** — был добавлен отдельный поток, в котором происходит сбор статистики и отправлен на сервер, а также проверка сессии на валидность.

Наблюдение за статистическими показателями производится в режиме непрерывного мониторинга, с активацией процедуры сбора данных при изменении состояния системы, преобразуется в необходимый формат и отправляется на сервер в отдельном потоке, при использовании очереди с синхронизацией.

4.3. Реализация для инструмента USVM

Для создания системы сбора статистики был реализован класс **Agent**, который является наследником класса **UMachineObserver**, что обеспечивает ему интеграцию в текущую архитектуру. Внутренняя структура класса **Agent** включает класс **ServerConnection**, ответственный за взаимодействие с удаленным сервером. Этот класс инкапсулирует логику подключения, отправки и получения данных. Для сбора, обработки и отправки метрик в **Agent** созданы методы для получения сырой статистической информации и ее преобразования в подходящий для передачи формат. Данные операции выполняются параллельно, в отдельном потоке, что позволяет избежать блокировки основного потока и замедления работы символьного исполнения.

Структура данных для представления метрик представлена в виде отдельных классов данных. Конфигурационные параметры системы хранятся в объекте **Config**.

Метод **onMachineStopped()**, унаследованный от **UMachineObserver**, инициирует процедуру корректного завершения работы системы сбора

статистики символьного исполнения. Вызов метода **onMachineStarted()** приводит к запуску процесса сбора статистических данных. В свою очередь, метод **onStateTerminated()** реализует функцию сбора текущих метрических данных, отражающих состояние системы на момент завершения анализируемого состояния.

5. Апробация

Этот раздел содержит в себе описание тестового стенда, и в двух главах результаты апробации системы мониторинга символьного исполнения для инструментов KLEEF и USVM.

5.1. Описание тестового стенда

Тестирование производилось на базе операционной системы Ubuntu 23.10. Ниже перечислены технические характеристики тестового стенда.

Процессор: **Intel Core i5-10500H**

Графический процессор: **Intel UHD Graphics**

Оперативная память: **16GB**

5.2. Результаты апробации для инструмента KLEEF

Ниже представлены две таблицы, результаты включенной и выключенной системой мониторинга.

Таблица 1: Результат с включенной системой мониторинга

Параметр	Значение
Время исполнения	42.93
Количество инструкций	26489
Количество разветвлений	191

Таблица 2: Результат с выключенной системой мониторинга

Параметр	Значение
Время исполнения	37
Количество инструкций	26489
Количество разветвлений	191

Система мониторинга незначительно замедляет процесс символьного исполнения, но выдает корректные результаты.

5.3. Результаты апробации для инструмента USVM

В USVM запускается большое количество тестов, время исполнения которых без мониторинга занимает 6 минут и 14 секунд, а с мониторингом тесты проходят за 7 минут и 20 секунд, также выдавая корректные результаты.

Система мониторинга не критически замедляет процесс символьного исполнения, но при этом также выдает корректные результаты, которые можно наблюдать в режиме реального времени. Также предоставлен пример работы системы мониторинга.

Function Name	Forks	Instructions	SolverTime
foo	0	8621	0.0
bar	191	12307	0.04
main	0	395	0.0
klee_div_zero_check	0	5166	0.0

Рис. 5: Пример работы системы мониторинга

Заключение

В ходе выполнения данной работы были достигнуты следующие результаты:

1. Выполнен обзор символьного исполнения, инструментов KLEEF, USVM.
2. Разработана архитектура системы сбора статистики для KLEEF и USVM, а именно, создан список необходимых для работы классов.
3. Реализована система мониторинга для инструментов KLEEF и USVM.
4. Проведена апробация полученного результата.

Результат работы над KLEEF¹ и USVM² можно посмотреть на GitHub.

¹<https://github.com/YazoonDinalt/klee/pull/1>

²<https://github.com/YazoonDinalt/usvm/pull/1>

Список литературы

- [1] All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution. — URL: <https://ieeexplore.ieee.org/document/5504796> (дата обращения: 2025-01-06).
- [2] KLEE. — URL: <https://llvm.org/pubs/2008-12-OSDI-KLEE.pdf> (дата обращения: 2024-12-30).
- [3] KLEEF. — URL: <https://toolchain-labs.com/pdfs/kleef.pdf> (дата обращения: 2024-12-29).
- [4] Owi. — URL: <https://github.com/OCamlPro/owi> (дата обращения: 2024-12-29).
- [5] R&D Toolchain labs. — URL: <https://toolchain-labs.com/> (дата обращения: 2024-12-29).
- [6] S2E. — URL: <https://s2e.systems/> (дата обращения: 2024-12-29).
- [7] A Survey of Symbolic Execution Techniques. — URL: <https://arxiv.org/pdf/1610.00502> (дата обращения: 2025-01-06).
- [8] Symbolic execution and program testing. — URL: <https://dl.acm.org/doi/10.1145/360248.360252> (дата обращения: 2025-01-06).
- [9] A System to Generate Test Data and Symbolically Execute Programs. — URL: <https://ieeexplore.ieee.org/document/1702368> (дата обращения: 2025-01-06).
- [10] Результаты Test Comp 24. — URL: <https://test-comp.sosy-lab.org/2024/results/results-verified/> (дата обращения: 2024-12-29).