Федеральное государственное бюджетное учреждение науки
Институт системного программирования Российской академии наук

На правах рукописи

Мутилин Вадим Сергеевич

Верификация драйверов операционной системы Linux при помощи предикатных абстракций

05.13.11 – математическое и программное обеспечение
вычислительных машин, комплексов и компьютерных сетей

ДИССЕРТАЦИЯ
на соискание ученой степени
кандидата физико-математических наук

Научный руководитель
д. ф.-м. н., проф.
Петренко Александр Константинович

Москва – 2012

Содержание

Введение 4

Глава 1. Обзор работ в области верификации драйверов операционных систем 11

1.1. Ядро ОС Linux 11

1.2. Правила 13

1.3. Инструменты тестирования (динамической верификации) . . . 15

1.4. Общецелевые инструменты статического анализа 15

1.5. Системы верификации драйверов операционных систем 20

1.6. Выводы 26

Глава 2. Метод верификации драйверов ОС Linux 27

2.1. Поток команд 29

2.2. Шаг 1 30

2.3. Шаг 2 33

2.4. Шаг 3 35

2.5. Построение моделей правил 36

2.6. Отделение привязки к интерфейсу ядра 40

Глава 3. Метод генерации окружения целевого драйвера . . . 43

3.1. Сценарии взаимодействия 44

3.2. Основные определения 47

3.3. Формальная модель драйвера и его окружения в π-исчислении 51

3.4. Примеры задания окружения в виде процессов с учетом правил взаимодействия 52

3.5. Трансляция процессов в Си программу 55

Глава 4. Методы оптимизации анализа при помощи предикатных абстракций 57
4.1. Общая информация 58
4.2. Метод CEGAR 58
4.3. Известные ограничения 84
4.4. Выводы 85

Глава 5. Система верификации драйверов ОС Linux 87
5.1. Пользовательский интерфейс системы 87
5.2. Разработка адаптера инструмента верификации 90

Глава 6. Методика выявления и классификации правил корректности 94
6.1. Выбор источника 94
6.2. Методика анализа журнала изменений 95
6.3. Классификация ошибок взаимодействия драйверов с ядром ОС Linux 100
6.4. Аналогичные работы 102

Глава 7. Практические результаты 105

Заключение 107

Литература 108

Приложение А. Приложения к методу генерации окружения целевого драйвера 118
А.1. Трансляция процессов в Си программу 118
А.2. Последовательный случай 130
А.3. Доказательство теоремы 133

Введение

Актуальность темы

В работе ставится задача верификации драйверов операционной системы (ОС) Linux. Обеспечение надежности и, в частности, информационной безопасности программных и программно-аппаратных систем является одной из главных задач современных информационных технологий. Особый вклад в решение этой задачи вносят работы по верификации системного программного обеспечения, в первую очередь операционных систем. Верификация драйверов ОС Linux является критически важной задачей, так как:

- Корректность драйверов является необходимым условием обеспечения надежности и безопасности систем, поскольку драйверы работают с тем же уровнем привилегий, что и остальное ядро.

- Драйверы ОС Linux, входящие в ядро, это большой, стремительно растущий класс программных систем. На данный момент суммарный объем драйверов составляет более 9 миллионов строк исходного кода. За последний год он увеличился на полмиллиона строк.

Для драйверов, которые входят в состав ядра, обеспечение качества проводится силами разработчиков ядра. По данным Linux Foundation, на сегодняшний день в разработке каждой версии ядра, выходящей раз в 2-3 месяца, участвуют более 1000 человек, представляющих более 200 различных организаций. Несмотря на такое большое количество разработчиков, значительное число изменений в уже выпущенных и новых версиях ядра связано с исправлением ошибок в драйверах. Так, например, анализ изменений в драйверах стабильных версий ядра за год разработки с 26.10.10 по 26.10.11 показал, что порядка 80% изменений являются исправлениями ошибок. Происходит это в силу того, что обеспечивать надежность драйверов ОС Linux затруд-

нительно ввиду огромного количества достаточно сложного исходного кода, который должен удовлетворять большому числу разнообразных правил корректности, начиная от общих правил, которым должны подчиняться все программы на Си, и заканчивая специфичными правилами, которые говорят о том, как драйверы должны использовать интерфейс ядра.

Методы общецелевого статического анализа позволяют обнаруживать нарушения общих правил, таких как разыменование нулевых указателей, выход за границу массива. Однако, помимо них остается значительная часть специфичных для ОС Linux ошибок взаимодействия драйвера с интерфейсом ядра. По данным анализа изменений в драйверах стабильных версий ядра за год разработки с 26.10.10 по 26.10.11 количество специфичных ошибок составляет более половины от всех ошибок, являющихся нарушениями правил корректности.

Перспективным направлением является разработка высокоточных инструментов статической верификации при помощи предикатных абстракций. Примерами являются инструменты BLAST, CPAchecker, SLAM. За счет построения предикатных абстракций они позволяют показать не только наличие ошибок, но и их отсутствие для заданного правила.

Ключевым свойством высокоточных инструментов для проверки специфических правил является возможность итеративного уточнения абстракции по методу CEGAR (Counter Example Guided Abstraction-Refinement), что позволяет подстраивать абстракцию как под произвольное правило корректности, так и для конкретного драйвера. Это выгодно отличает такие инструменты от общецелевых, в которых настройка под правило корректности требует реализации соответствующей функциональности в инструменте статического анализа.

Применение инструментов высокоточного статического анализа к драйверам ОС Linux имеет хорошие предпосылки. Высокоточные методы в насто-

ящее время имеют ограничения по размеру анализируемого кода (до 50-100 тыс. строк). В случае драйверов это ограничение приемлемо. Размер драйверов, входящих в состав ядра ОС Linux, не превышает 50 тысяч строк, а в среднем составляет порядка 2-3 тыс. строк кода. Кроме того, важно, что большинство драйверов публикуется вместе с исходным кодом, который является необходимым для большинства инструментов статического анализа.

Для применения высокоточных инструментов требуется подготовка драйвера к верификации. Во-первых, требуется подготовить специальное окружение драйвера, которое должно описывать возможности воздействия на него с учетом ограничений, накладываемых на взаимодействия сердцевины ядра ОС с драйверами данного типа. Во-вторых, требуется формально описать правила корректности в виде, удобном для сведения задачи верификации к доказательству достижимости точки программе. При решении этих задач необходимо учитывать специфику ядра ОС Linux. Одна из главных особенностей состоит в том, что интерфейсы между драйвером и ядром ОС Linux постоянно меняются. Меняются правила взаимодействия, появляются новые, меняется окружение драйвера. Поэтому правила, модели окружения и другие части системы верификации должны быть устроены так, чтобы обеспечивать простоту развития, адаптации к текущему состоянию ядра ОС, правилам взаимодействия между драйверами и ОС. На сегодняшний день не существует методов верификации, учитывающих данные особенности.

Таким образом, задача верификации драйверов ОС Linux является актуальной, для ее решения требуется разработка нового метода верификации.

Цель и задачи работы

Цель работы – разработка метода верификации драйверов устройств операционных систем для проверки выполнения правил корректного взаимодействия драйверов с ядром операционной системы.

Для достижения цели работы были поставлены следующие задачи:

1. Провести анализ существующих методов верификации драйверов;

2. Провести анализ ошибок в драйверах ОС Linux, приводящих к некорректному взаимодействую с ядром ОС;

3. Разработать метод верификации и архитектуру системы верификации драйверов ОС Linux при помощи предикатных абстракций, обеспечивающий:

   - верификацию драйверов в условиях непрерывного развития ядра;
   - возможности расширения (конфигурируемости) системы верификации драйверов за счет пополнения набора правил корректности и набора инструментов верификации.

4. Разработать систему верификации, реализующую метод;

5. Оценить реализацию метода на практике, дать оценку области применимости метода.

Научная новизна работы

Научной новизной обладают следующие результаты работы:

1. Метод построения моделей окружения драйверов устройств ОС Linux;

2. Метод построения конфигурируемой системы верификации;

3. Математическое доказательство адекватности предложенного метода формализации правил корректности в рамках заданного класса правил работы со структурами обработчиков событий;

4. Методы оптимизации предикатной абстракции в BLAST.

Практическая значимость

На основе разработанного метода была создана система верификации драйверов ОС Linux LDV (Linux Driver Verification). Система LDV позволяет находить нарушения правил корректности взаимодействия с ядром ОС для драйверов, входящих в поставку ядер ОС Linux с версии 2.6.31 по 3.4. По состоянию на 25.09.2012 было найдено более 60 ошибок, которые признаны разработчиками ядра и уже исправлены или будут исправлены в последующих версиях.

Результаты работы будут полезны для автоматизации сертификационных процессов для компьютерного обеспечения, которые существуют у многих поставщиков дистрибутивов ОС Linux и ОС, построенных на основе ядра Linux. Также результаты могут быть использованы для проверки качества драйверов ОС Linux, для создания инструментов верификации других видов программного обеспечения, критичного по безопасности. Результаты работы могут быть использованы для сравнения характеристик различных инструментов статического анализа кода.

Доклады и публикации

По теме диссертации автором опубликовано 15 работ [1–15] (из них 5 в изданиях из перечня ВАК [1–5]).

Основные положения работы докладывались на следующих конференциях и семинарах:

- Весенний коллоквиум молодых исследователей в области программной инженерии (SYRCoSE: Spring Young Researchers Colloquium on Software Engineering, г. Санкт-Петербург, 2008 г.);
- Общероссийская научно-техническая конференция “Методы и технические средства обеспечения безопасности информации” (г. Санкт-Петербург, 2008 г.);
- Международный семинар “Принципы и технические средства сертифи-

кации свободного программного обеспечения" (OpenCert: International Workshop on Foundations and Techniques for Open Source Software Certification, г. Милан, 2008 г.);

- Конференция разработчиков свободных программ на Протве (г. Обнинск, 2009 г.);
- Международная конференция памяти академика А.П.Ершова "Перспективы систем информатики" (PSI: Perspectives of System informatics, г. Новосибирск, 2011 г.);
- Международная конференция по инструментам и алгоритмам создания и анализа систем (TACAS: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, г. Таллин, 2012 г.);
- Семинар "День свободного программного и аппаратного обеспечения" (г. Москва, 2012 г.);
- Семинар Института системного программирования РАН (г. Москва, 2012 г.).

Структура и объем диссертации

Работа состоит из введения, семи глав, заключения и списка литературы (55 наименований). Основной текст диссертации (без приложений и списка литературы) занимает 115 страниц.

Основные результаты работы

Основные научные и практические результаты, полученные в диссертационной работе и выносимые на защиту, состоят в следующем:

1. Разработан метод верификации драйверов устройств операционных систем для проверки выполнения правил корректного взаимодействия

драйверов с ядром операционной системы;

2. Разработан метод построения моделей окружения драйверов устройств ОС Linux;

3. Разработан метод построения конфигурируемой системы верификации, обеспечивающий возможность расширения системы за счет пополнения набора правил корректности и набора инструментов верификации;

4. Разработаны методы оптимизации предикатной абстракции в инструменте BLAST;

5. На основе предложенных методов разработана система верификации драйверов ОС Linux.

Система верификации драйверов ОС Linux разработана в рамках проектов отдела Технологий программирования Института системного программирования РАН при непосредственном участии автора в качестве руководителя и участника разработки основных компонентов системы верификации. Автор выражает свою признательность всем участникам данных проектов. Особый вклад в работу внесли А.В.Хорошилов, Е.М.Новиков, П.Е.Швед.

Глава 1

Обзор работ в области верификации драйверов операционных систем

Прежде, чем приступить к обзору различных существующих методов верификации драйверов устройств ОС Linux рассмотрим подробно ядро ОС Linux и организацию его разработки и развития.

1.1. Ядро ОС Linux

Ядро ОС Linux является одним из самых динамично развивающихся и востребованных проектов в мире. Разработку ядра начал Линус Торвальдс в 1991 году. В настоящее время в подготовке каждого нового релиза ядра ОС Linux участвуют более 1000 человек, распределенных по всему миру [16]. Релизы выпускаются в среднем раз в 2-3 месяца. Начиная с ядра версии 2.6.24, выпущенного в начале 2008 года, каждый релиз включает порядка 9-12 тысяч изменений, что соответствует примерно 5 изменениям в час. На сегодняшний день размер исходного кода ядра составляет более 15 млн. строк кода.

Следует отметить тот факт, что помимо официальной, так называемой оригинальной (от англ. mainline) версии ядра ОС Linux, которую выпускает Линус Торвальдс, объединяя разработки сообщества разработчиков ядра со своими, существует большое количество других веток разработки ядра. Например, как правило, разработчики различных дистрибутивов Linux поддерживают свои версии ядра. Об этом говорят, например, разработчики дистрибутивов Red Hat Enterprise Linux, openSUSE и Debian. Данные версии ядра отличаются от оригинальной тем, что в них поддерживается некоторая

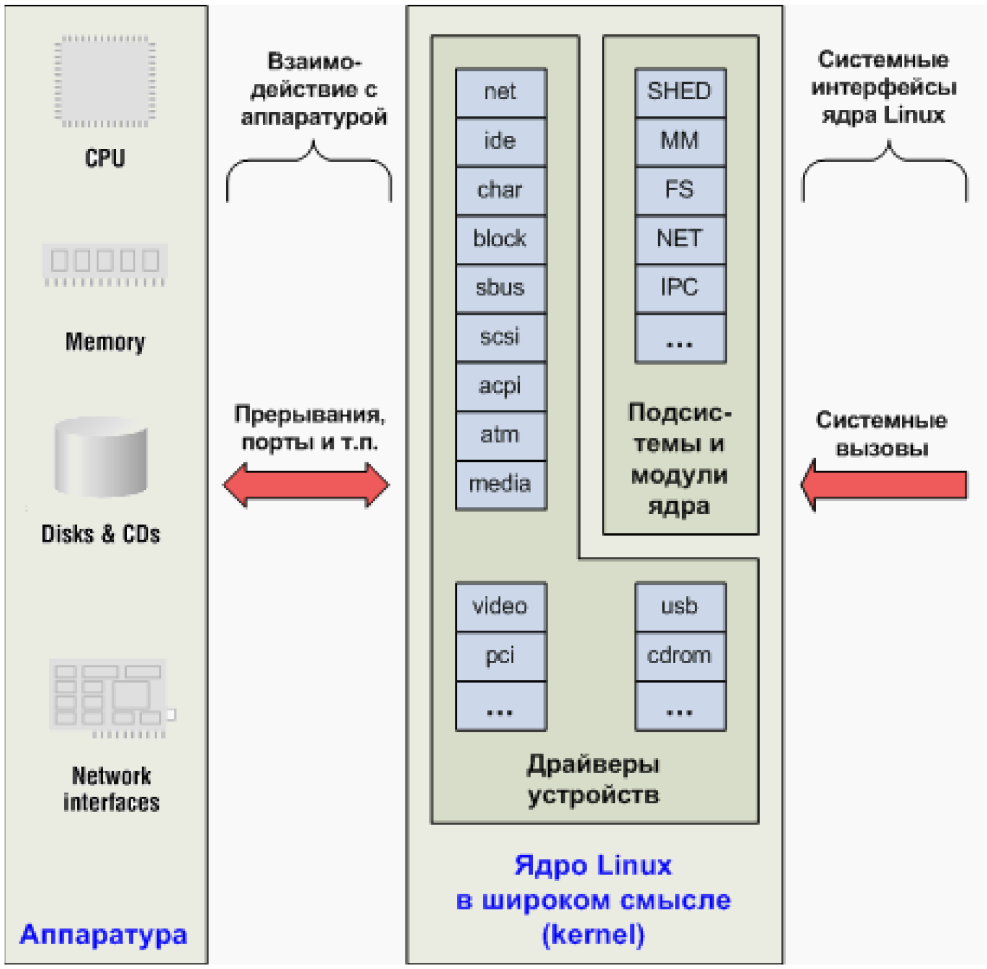

Рис. 1.1. Схема устройства ядра операционной системы Linux

дополнительная функциональность и/или содержатся исправления ошибок. Есть ветки ядра, в которых основы операционной системы реализуются принципиально иначе по сравнению с оригинальной версией [17, 18]. С течением времени изменения, интересные широкому кругу лиц, из различных веток разработки попадают в оригинальное ядро [19, 20].

Схема устройства ядра ОС Linux представлена на 1.1. Ядро состоит из основной части (“сердцевины”) и драйверов устройств. Драйверы устройств участвуют во взаимодействиях:

- с сердцевиной ядра, обрабатывая события и запросы, связанные с соответствующим устройством, и вызывая “библиотечные” функции из сердцевины;

- с аппаратурой, которую они представляют в ядре;
- с пользовательскими приложениями посредством специальных файлов (procfs, sysfs или debugfs).

Драйверы занимают наибольшую часть (до 70%) ядра ОС Linux. В исходном коде драйверов содержится достаточно большое количество различных ошибок, приводящих к некорректной работе всей ОС, зависаниям и падениям. Результаты исследований, которые были проделаны в работах [21, 22] в начале 2000-х годов для ядер версий от 1.0 до 2.4.1, показали, что драйверы содержат до 85% всех ошибок, которые встречаются во всем ядре ОС Linux. Подобное исследование для ядра ОС Microsoft Windows XP в 2006 году также показало, что наибольшее количество ошибок в ядре данной ОС происходит в драйверах устройств [23]. Более поздние исследования, проведенные в 2011 году для ядер ОС Linux версий от 2.6.0 до 2.6.33, продемонстрировали, что хотя количество ошибок в драйверах стало меньше, чем в компонентах ядра, отвечающих за поддержку различных архитектур и файловых систем, их по-прежнему достаточно много [24].

1.2. Правила

Ошибки в драйверах можно условно разделить на типовые и нетиповые. К нетиповым относятся ошибки, которые связаны с некорректным взаимодействием с соответствующими устройствами и нарушениями контрактов интерфейса драйвера. Нетиповые ошибки выделяются среди других ошибок тем, что они затрагивают характерные только для некоторого драйвера константы, ограничения, вычисления и т.д. Особенность типовых ошибок заключается в том, что для них можно выделить соответствующие правила, общие для всех драйверов либо для группы драйверов.

Среди типовых ошибок можно выделить три класса. В первый класс входят общие ошибки в драйверах, как в программах на языке программирования Си (ядро и драйверы ОС Linux разрабатываются на языке программирования Си). Например, разыменование нулевого указателя, превышение максимально возможных значений переменных с целым типом и т.п. Ко второму классу относятся специфичные ошибки, связанные с неправильным использованием интерфейса сердцевины ядра. К числу таких ошибок относятся, например, нарушения правил инициализации переменных, имеющих специфичные типы. Третий класс типовых ошибок включает в себя состояния гонок и взаимные блокировки, которые возникают при параллельном выполнении вследствие неиспользования или неправильного использования механизмов синхронизации.

Для типовых общих ошибок существуют классификации [25]. Правила, вследствие нарушения которых они происходят, практически не меняются со временем (вообще говоря, язык программирования Си и его расширения продолжают развиваться, что приводит к незначительным изменениям в общих правилах). Ошибки, связанные с параллельным выполнением, также достаточно стандартные и мало меняются с течением времени.

Типовые специфичные ошибки не являются стандартными. Правила, соответствующие им меняются, удаляются и появляются достаточно часто ввиду того, что разработка ядра ОС Linux, как уже отмечалось, ведется большими темпами, меняется и интерфейс сердцевины ядра. По заявлению Грега Кроа-Хартмана, одного из ведущих разработчиков ядра ОС Linux, предполагается, что интерфейс основной части ядра является нестабильным [26].

Существуют различные способы выделения правил, о которых написано в главе 6.

Далее рассмотрим техники и инструменты, которые используются для верификации драйверов устройств: техники тестирования (динамической ве-

рификации), общецелевые техники статического анализа и специализированные системы верификации драйверов.

1.3. Инструменты тестирования (динамической верификации)

Данные, необходимые для анализа, в этих инструментах собираются во время исполнения кода драйвера, а поэтому инструменты требуют наличия оборудования, для которого предназначен драйвер, или эмуляции этого оборудования. С другой стороны, исходный код анализируемых драйверов обычно не требуется. Это позволяет анализировать драйверы с закрытым исходным кодом. Инструменты динамической верификации обычно дают меньше ложных срабатываний, чем системы статического анализа, однако проверяют только один путь исполнения в данный момент времени. Инструменты статического анализа, напротив, одновременно проверяют многие (или даже все) пути исполнения в коде драйвера, но ложные срабатывания в них более вероятны, чем при использовании динамического анализа.

1.4. Общецелевые инструменты статического анализа

Общецелевые инструменты статического анализа появились в области разработки компиляторов [27]. Они обладают скоростью работы, сравнимой со скоростью работы компилятора. Эти инструменты используют синтаксические методы, основанные на поиске ошибочных конструкций на деревьях синтаксического вывода, и методы анализа потоков данных, также называемые абстрактной интерпретацией. Последние основаны на том, что некоторое множество значений распространяется по конструкциям программы до тех пор, пока это множество не “насытится”, т.е. не будет меняться при последу-

ющем распространении. Математически это записывается как итеративное применение монотонной функции. Насыщение достигается при достижении неподвижной точки.

Общецелевые инструменты статического анализа крайне полезны для выявления типовых ошибок, общих для языка программирования, для которого они предназначены.

Первые общецелевые инструменты статического анализа кода появились еще в 70-х годах, например, в компании Bell Labs был разработан инструмент Lint для анализа Си программ [28]. Инструмент был способен находить ошибочные конструкции, характерные для языка Си, в том числе, использование неинициализированных переменных, некорректное использование констант внутри условий, вычисления, результат которых не укладывается в результирующий тип. Многие из этих проверок осуществляются непосредственно современными компиляторами.

На сегодняшний день существует большое количество различных общецелевых инструментов статического анализа, которые широко применяются в индустриальной разработке программ. В данном обзоре рассмотрены такие инструменты, для которых имеется опыт применения к ядру ОС Linux.

Инструмент **Sparse**[29] развивается внутри ядра Linux. Отмечается, что его начал развивать Линус Торвальдс, основатель Linux, в 2003 году. В дальнейшем его разработку продолжили разработчики ядра Josh Triplett и Christopher Li. Инструмент основан на синтаксических методах.

Sparse находит следующие виды ошибок:

1. Преобразование типа в ограниченный тип или использование ограниченных типов в неподходящих выражениях (*bitwise, cast-truncate, default-bitfield-sign, enum-mismatch, one-bit-signed-bitfield, ptr-subtraction-blows, typesign*).

2. Использование деклараций: статические переменные (*decl*), декларации переменных по стандарту C89 вне начала блока (*declaration-after-statement*), скрытие деклараций (*shadow*), использование атрибутов transparent-union (*transparent-union*).

3. Использование указателей из неподходящего адресного пространства (*address-space, cast-to-as*).

4. Использование старого синтаксиса инициализации структур (*old-initializer*) или ослаблений GCC при инициализации массивов (*paren-string*).

5. Аннотирование контекста использования функций, в частности, использование синхронизационных примитивов (*no-context*).

6. Отсутствие скобок для блока цикла (*do-while*).

7. Использование нуля в качестве нулевого указателя (*non-pointer-null*).

8. Проверка возвращаемых значений функций (*return-void*).

9. Использование параметров макроса (*undef*).

В проверках инструмент опирается на аннотации в исходном коде ядра с использованием расширений GCC. В приведенных правилах указано как минимум пять разных расширений. Поэтому поддержка данных расширений является особенно важной для инструмента статического анализа нацеленного на ядро ОС Linux, так как данные атрибуты дают дополнительные подсказки статическому анализатору. Из статьи [24] известно, что инструмент Sparse использовался для нахождения ошибок в версиях ядер 2.6.25, 2.6.29, 2.6.30. Было найдено не менее 140 ошибок.

Инструмент **Stanse**[30] умеет находить ошибки, связанные с неправильным использованием функций блокировки, неправильным использованием памяти, потенциально спящих (*sleep*) функций внутри атомарного контекста, недостижимого кода, неправильным использованием функций работы с прерываниями (*irq disable/enable*), подсчета ссылок на *pci, tty* структуры устройств. Инструмент основан на межпроцедурном анализе потоков данных. В 2009-2010 годах инструмент был применен к ядру Linux, было найдено порядка 130 ошибок.

Инструмент **Coverity**[31] находит такие типы ошибок, как наличие недостижимого кода, разыменование нулевых указателей, использование до сравнения, переполнение буфера, утечки ресурсов, небезопасное использование возвращаемых значений, несовпадение размеров типа и выделяемой памяти, чтение неинициализированных переменных и использование памяти после освобождения. Существуют адаптации данных типов ошибок под различные интерфейсы, например, в ядре Linux в качестве функции выделения ресурсов рассматривается функция выделения памяти под структуру *urb* подсистемы *usb*.

Инструмент применялся к ядру ОС Linux в промежутке между 2006 и 2009 годами, в рамках контракта с Департаментом безопасности США [32]. Авторы работы приводят сведения о том, что из всех предупреждений ими было отобрано 2,125 сообщений об ошибках. Все эти сообщения были отправлены разработчикам ядра. Однако лишь 61% из них было принято в обработку. Сведения о количестве ложных предупреждений и исправленных ошибках не приводятся.

Инструмент **Cocinelle**[33] изначально ставил своей целью поддержку изменений в коде, таким образом, что описывался шаблон изменения, и далее этот шаблон применялся ко всему исходному коду. В ядре ОС Linux такими изменениями являются, например, переименование функции, добавление

аргумента функции для которого имеются ограничения по контексту, реорганизация структуры данных. Также инструмент используется для поиска ошибок в ядре, которые описываются такими же шаблонами, как и для поддержки изменений, но вместо применения изменений выводится сообщение об ошибке. Успешность применения Cocinelle к ядру ОС Linux сильно зависит от искомого шаблона ошибки. В работе [34] для некоторых шаблонов количество ложных срабатываний составляло 98% (ошибки переполнения буфера), что недопустимо много. Для шаблона ошибки использование после освобождения количество ложных предупреждений составило 60%, что гораздо лучше, но все еще очень много. В статье [35] авторы выбирали шаблоны с наименьшим количеством ложных срабатываний. В итоге среднее количество ложных срабатываний составило менее 15%. Всего с помощью Cocinelle было найдено 360 ошибок, 30 из которых были исправлены в ядре Linux.

Инструмент **Saturn**[36, 37] реализует межпроцедурный анализ, основанный на выведении аннотаций функций. Для анализа каждой конкретной функции используются высокоточные методы, такие как построение формулы пути и использование SAT решателей. Межпроцедурный анализ использует только аннотации функций.

Авторы уделяют большое внимание распараллеливанию анализа, которое получается естественным образом за счет анализа каждой функции отдельно, за исключением зависимостей между функциями. Запуск на кластере из 40-100 процессоров параллельно дает 80-90% эффективности, в результате анализ ядра ОС Linux занимает несколько часов, что сравнимо с другими общецелевыми подходами не использующими SAT решатели.

В статье [38] применение высокоточных методов расширено до анализа межпроцедурных зависимостей. Приводятся сведения о применении инструмента Saturn к исходному коду ядра ОС Linux версии 2.6.17.1. В частности, в работе показано, что применение более точного анализа позволяет существен-

но снизить количество ложных срабатываний с 1344 до 37 при неизменном количестве ошибок (134).

Инструмент **Locksmith**[39] один из немногих инструментов общецелевого анализа который ищет ошибки связанные с состояниями гонок при параллельном выполнении. В работе приводятся данные о применении инструмента к десятку драйверов ядра Linux, однако ничего не сообщается об обнаруженных ошибках.

Отметим также, что существует также множество других инструментов общецелевого статического анализа, например, коммерческий инструмент **Klocwork Insight**[40], академические инструменты **FindBugs**[41], **Splint**[42], **Svace**[43] и др.

Все общецелевые статические анализаторы направлены на обнаружение общих ошибок, типичных для большинства Си программ. Однако они не учитывают специфику анализа драйверов, так как не обладают знанием о логике взаимодействия драйвера и ядра ОС. По этой причине они не могут выявить ошибки нарушения такой логики или правил взаимодействия драйверов с ядром ОС.

1.5. Системы верификации драйверов операционных систем

Все такие системы предназначены для поиска ошибок связанных с нарушением правил взаимодействия драйверов с ядром ОС. В связи с этим, требуется подготовка специального окружения драйвера, что позволяет, в частности, учитывать ограничения на порядок вызовов обработчика драйвера. Для верификации используются инструменты статической верификации, позволяющие проверять достижимость произвольной точки программы.

1.5.1. Система Static Driver Verifier

Наиболее развитой системой верификации драйверов операционных систем является система Microsoft SDV[44], позволяющая проводить статическую верификацию драйверов операционной системы Microsoft Windows. Она используется в процессе сертификации драйверов и включена в состав Microsoft Windows Driver Developer Kit, начиная с 2006 года. Система Microsoft SDV наглядно демонстрирует возможность применения верификации реальных программ.

В системе SDV для создания окружения от пользователя требуется вручную аннотировать исходный код драйверов, указав в нем роли каждой из функций обработчиков. Аннотации необходимо создавать вручную, однако, положительный эффект от использования SDV компенсирует трудозатраты на аннотирование, поэтому зачастую аннотации пишут непосредственно разработчики драйверов. В рамках подхода SDV в качестве модели окружения генерируется произвольная последовательность вызовов в соответствии с типами функций обработчиков. В генерируемой последовательности учитываются ограничения на порядок вызов функций в соответствии с типами обработчиков. Функции основной части ядра ОС Microsoft Windows моделируются упрощенными аналогами, причем для них, как правило, используются недетерминированные модели. Благодаря этому в дальнейшем инструмент статического анализа кода исследует все ветви недетерминизма, проверяя тем самым все разнообразные варианты работы функций, включая редко встречающиеся.

Проверяемые правила корректности формализуются с помощью языка SLIC[45], в котором связь с исходным кодом драйвера задается с помощью аспектно-ориентированных конструкций, перехватывающих вызовы функций ядра. В рамках данного подхода для разработки модели ошибки предостав-

```
state { int zero_cnt = 0; }
put.entry {
  if ($1 == 0) {
    if (zero_cnt == 4)
      abort "Queue has 4 zeroes!";
    else
      zero_cnt = zero_cnt + 1;
  }
}
get.exit {
  if ($return == 0)
    zero_cnt = zero_cnt - 1;
}
```

Рис. 1.2. Пример спецификации SLIC

ляется относительно простой специальный язык, очень похожий на язык программирования Си. Данный язык нацелен только на моделирование поведения функций, также имеется возможность сохранять глобальное состояние программы. Вообще говоря, из-за этого достаточно сильно страдает выразительная мощь, необходимая для моделирования ошибок. Однако простота языка позволяет использовать его для построения моделей ошибок даже тем пользователям, которые не являются экспертами в области верификации. В настоящее время уже выделен набор из примерно 200 правил, а в исследовательской версии была добавлена возможность добавления новых правил.

Приведенная на Рис.1.2 спецификация на языке SLIC демонстрирует проверку искусственной ошибки. В данном случае, ошибкой является ситуация, когда в очереди больше четырех нулей. Препроцессор SLIC инструментирует исходный код драйверов на основе спецификации. В конечном итоге, инструмент генерирует эквивалентную программу на Си, которая затем проверяется с помощью инструмента статического анализа кода.

К сожалению, SLIC можно использовать только для создания спецификации и инструментации для драйверов ОС Microsoft Windows. Кроме того,

исходный код реализации системы SDV полностью закрыт.

В систему интегрированы инструменты статической верификации SLAM[46, 47] и Yogi[48].

В виду того, что интерфейс между драйвером и ОС Windows изменяется крайне редко, при разработке архитектуры системы не требовалось учитывать возможность постоянного развития интерфейса между драйверами и ядром ОС.

1.5.2. Системы Avinux и DDVerify

Существуют также системы верификации драйверов ядра ОС Linux: Avinux[49], разработка университета города Тюбинген (Германия) и DDVerify[50], разработка Оксфордского университета (Англия). Разработка систем была нацелена во многом на апробацию новых возможностей инструментов верификации и не учитывала ряд особенностей, необходимых для промышленного использования, в настоящее их разработка не ведется.

Система DDVerify

Данная система использует два инструмента высокоточного статического анализа кода: CBMC[51] и SATABS[52]. Особый упор при разработке DDVerify был сделан на проверку ошибок, связанных с некорректным использованием разделяемой памяти при многопоточном окружении, поскольку ошибки данного вида являются одними из самых трудных для обнаружения и понимания.

Для извлечения информации о составе драйверов инструмент DDVerify использует собственные файлы сборки, из-за чего специфика конкретного ядра Linux и его конфигурации не учитывается в полной мере.

Подход к генерации модели окружения, использованный в DDVerify,

предполагает ручное построение модели основной части ядра для каждого типа драйвера. Так, симулируя работу ядра, окружение вызывает функцию инициализации драйвера и ожидает регистрацию драйвером обработчиков прерываний для данного типа драйвера. После регистрации обработчиков окружение осуществляет вызовы с учетом ограничений на основании знаний о типе драйвера. Для некоторых типов разработаны модели, допускающие работу нескольких обработчиков в многопоточном режиме. Следует отметить то, что DDVerify позволяет использовать недетерминированное поведение при моделировании функций основной части ядра.

Таким образом, инструмент DDVerify требует значительных затрат труда при смене версий ядра ОС Linux, поскольку файлы сборки ядра постоянно обновляются (добавляются новые драйверы и подсистемы, меняются конфигурации и т.п.), а также меняются различные типы драйверов (например, могут быть добавлены новые обработчики).

Кроме того, как отмечают авторы работы [50], инструмент требует существенной доработки в отношении скорости выполнения проверки драйверов ОС Linux для его широкомасштабного применения. Для анализа результатов анализа имеется специальный плагин для интегрированной среды разработки Eclipse. В частности, имеется возможность анализа трассы ошибки с одновременным просмотром соответствующего исходного кода драйверов и ядра ОС Linux, что существенно облегчает анализ ложных срабатываний. С помощью DDVerify были обнаружены 2 реальные ошибки в 2 драйверах.

Система Avinux

Система Avinux получает информацию о составе драйверов и настройках их сборки путем встраивания в процесс сборки ядра за счет модификации файлов, описывающих сборку [49]. Однако, данный инструмент предоставля-

ет возможность автоматической работы только с единичными препроцессированными файлами в то время, как драйвер может состоять из нескольких файлов.

В работе [49] отмечено, что Avinux автоматически строит шаблон модели окружения, однако, дальнейшая модификация этого шаблона с учетом ограничений для заданного типа драйвера производится вручную. Следует отметить, что шаблон модели окружения включает в себя генерацию инициализации структур данных, которые используются, как параметры в функциях обработчиках [53]. Данная инициализация необходима для проверки свойств, связанных с обращением к неинициализированным участкам памяти, таких как разыменование нулевого указателя и выход на границу буфера.

Вообще говоря, разработчики инструмента Avinux изначально переносили подход, реализованный в инструменте SDV. В частности, для построения модели ошибки они предложили использовать расширение к языку спецификаций SLICx. С одной стороны, это позволило снять некоторые ограничения, например, стало возможным использовать произвольные выражения языка программирования Си. Однако, с другой стороны, в SLICx отсутствуют некоторые полезные свойства, которые были реализованы в новых версиях SLIC.

Avinux использует инструменты высокоточного статического анализа кода CBMC[51] и BLAST[5, 54].

Результаты применения Avinux для поиска повторных освобождений памяти и использования памяти после освобождения позволило обнаружить 6 уже известных ошибок и 2 новых. Более широкому применению Avinux на большем количестве правил и драйверов, также как и DDVerify, мешает необходимость существенной ручной настройки инструмента для подготовки исходного кода драйверов к проверке.

1.6. Выводы

Верификация драйверов устройств ОС Linux обладает рядом особенностей, которые необходимо учитывать. Одна из главных особенностей состоит в том, что интерфейсы между драйвером и ядром ОС Linux постоянно меняются. Это является принципиальной позицией разработчиков ядра ОС Linux, отмеченной в документации к ядру. Вызвано это тем, что разработчики не ограничивают себя в возможностях усовершенствования интерфейсов взаимодействия, например для ускорения работы системы. Таким образом, ядро Linux постоянно развивается, меняются правила взаимодействия, появляются новые, меняется окружение драйвера. Поэтому правила, модели окружения и другие части системы верификации должны быть устроены так, чтобы обеспечивать простоту развития, адаптации к текущему состоянию ядра ОС, правилам взаимодействия между драйверами и ОС. Эти особенности не позволяют использовать архитектуру системы Microsoft SDV для верификации драйверов ОС Linux.

Время жизни программных инструментов в мире свободного программного обеспечения прогнозировать трудно. Следовательно важно, чтобы система была открыта к изменениям, в частности, была готова интегрировать в себя новые инструменты верификации взамен устаревших или лишившихся поддержки.

Поэтому требуется разработка нового метода и архитектуры системы верификации, учитывающих особенности верификации драйверов ОС Linux.

Глава 2

Метод верификации драйверов ОС Linux

Предлагаемая архитектура системы верификации разработана с учетом особенностей задачи верификации драйверов ОС Linux, рассмотренных выше. Компоненты архитектуры осуществляют действия, соответствующие шагам метода.

Архитектура системы схематично изображена на Рис. 2.1. Компоненты архитектуры изображены в центре. Стрелки указывают порядок, в котором соответствующие компоненты обрабатывают входные данные. Слева изображены входные данные. Справа показан порядок формирования отчета по результатам верификации.

Компонент LDV-Core (шаг 1) осуществляет подготовку ядра ОС Linux (компонент Kernel Manager), извлечение исходного кода драйверов (Build Cmd Extractor) и генерацию модели окружения драйвера (Driver Environment Generator).

Компонент Domain Specific C Verifier (шаг 2) уже не должен учитывать специфику конструкции драйвера, поданный ему на вход программный код может быть и произвольной программой на языке программирования Си. Данный компонент принимает программу на языке программирования Си и передает ее компоненту Rule Instrumentor, чтобы тот встроил в нее код проверок указанных правил корректности. Затем Domain Specific C Verifier, используя Reachibility C Verifier, передает код конкретному инструменту верификации.

Инструмент статической верификации (шаг 3) выносит вердикт. По ходу этого процесса вердикт дополняется отчетами о работе других компонентов, после чего формируется финальный отчет о проверке всего задания, который

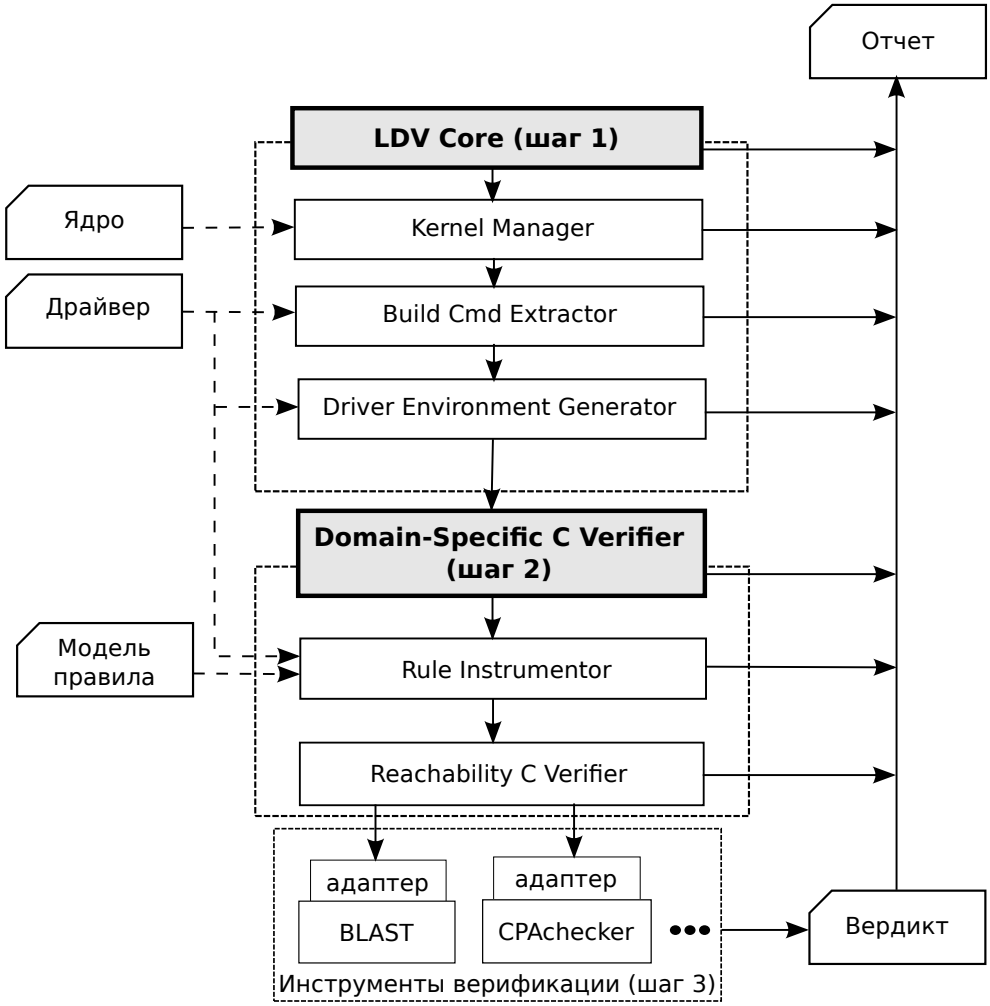

Рис. 2.1. Компоненты архитектуры и шаги метода верификации

затем анализируется.

2.1. Поток команд

Компоненты между собой общаются с помощью *потока команд*. Изначально он является некоторым представлением команд сборки (компиляции и линковки). По мере обработки, поток команд модифицируется каждым компонентом. Компоненты могут изменять опции препроцессора, дописывать мета-информацию и подменять пути к файлам теми, по которым они размещают модифицированные файлы.

Файл потока команд содержит структуру данных в формате XML, описывающую команды, выделенные при сборке драйвера.

Поток команд представляет собой множество описаний CC и LD команд компилятора (теги *cc* и *ld* соответственно), необходимых для компиляции одного и более драйвера. Кроме этого, есть тег *basedir* – путь к ядру Linux, с которым собирался драйвер.

Каждая CC или LD команда содержит:

- opt – Опции компилятора (команда CC) или линковщика (команда LD).
- cwd – Папка, из которой вызывался соответствующий инструмент.
- in – Входной файл. Файл с исходным кодом в случае компилятора. Объектные файлы в случае линковщика.
- out – Выходной файл, который должен получиться, в результате выполнения текущей команды. Если тег помечен атрибутом *check='true'*, значит, результат этой команды соответствует модулю драйвера и должен быть верифицирован.

2.2. Шаг 1

Процесс верификации драйверов начинается с запуска компонента *LDV Core*. Данный компонент вызывает компонент *Kernel Manager*, который создает на диске копию ядра ОС Linux, предоставленного пользователем в виде архива с исходным кодом или репозитория git. Затем *Kernel Manager* модифицирует подсистему сборки ядра, что впоследствии позволяет получить информацию о составе драйверов и настройках их сборки. Метод позволяет верифицировать как драйверы, входящие в предоставляемое им ядро ОС Linux, так и внешние драйверы относительно данного ядра.

Далее *LDV Core* запускает процесс компиляции копии ядра, по ходу которого на основе модифицированной подсистемы сборки *Build Cmd Extractor* читает поток команд компиляции и линковки, определяет зависимости между файлами с исходным кодом и выделяет то, что относится к верифицируемым драйверам.

После того, как получен полный поток команд, специальный подкомпонент *Build Cmd Extractor* разделяет данный поток на несколько небольших частей, каждая из которых соответствует одному модулю ядра. Эта процедура особенно важна для анализа драйверов, входящих в ядро, поскольку сама по себе подсистема сборки ядра не делает такого разделения.

Драйвер ядра ОС Linux, как правило, состоит из одного или нескольких модулей. Для своего вызова драйвер предоставляет ядру функции обработчики событий, такие как операции с файлами, операции с USB интерфейсом и т.п.; а также обработчики прерываний, таймеры, обработчики отложенных задач. Функции обработчики регистрируются при загрузке драйвера в ядро, а затем вызываются по мере поступления соответствующих запросов со стороны пользовательских приложений и оборудования.

Затем компонент *Driver Environment Generator* осуществляет генерацию

окружения драйвера, моделирующего воздействия на драйвер, осуществляемые операционной системой с учетом ограничений, накладываемых на данные взаимодействия. Создание синтезируемого окружения позволяет, с одной стороны, построить замыкание программного кода драйвера (инструменты верификации требуют наличия замкнутого кода) и, другой стороны, нацелить инструменты верификации на поиск ошибок характерных для данного вида драйверов. *Метод генерации окружения целевого драйвера* подробно описан в третьей главе. Метод позволяет осуществлять генерацию моделей окружения полностью автоматически на основе конфигурации типов драйверов, которая описывает ограничения на взаимодействия между драйвером и сердцевиной ядра. Таким образом, конфигурацию требуется изменять, только если меняются ограничения на взаимодействия. Причем, данное изменение осуществляется сразу для всех драйверов данного типа. Это позволяет обеспечивать актуальность системы верификации в условиях непрерывного развития ядра.

В результате этого шага файл потока команд пополняется командами сборки модели окружения, а также данными о точках входа для каждой из сгенерированных моделей окружения. Так *Driver Environment Generator* дописывает опцию препроцессора к CC команде, если сгенерировал точку входа в исходном файле, указанном в тегах *in*. А в LD команду дописывает теги *main* с именами точек входа.

Пример командного файла после работы компонента *Driver Environment Generator* показан на Рис. 2.2. Здесь компонент добавил опцию *'<opt>-DLDV_MAIN0</opt>'* к команде CC. Это значит, что в исходном файле сгенерирован код и функция с именем main, которая включается в сборку указанной опцией. А в LD команде в теге main указано имя этой функции *'ldv_main0'*.

```
<cmdstream>
  <basedir>/path/to/kernel</basedir>
  <cc id="1»
    <cwd>/driver</cwd>
    <in>/driver/main.c</in>
    <opt>-include</opt>
    <opt>include/generated/autoconf.h</opt>
    <opt>-c</opt>
    <out>/driver/main.o</out>
    <opt>-DLDV_MAIN0</opt>
  </cc>
  <ld id="2»
    <cwd>/driver</cwd>
    <in>/driver/main.o</in>
    <out check="true»/driver/main.ko</out>
    <main>ldv_main0</main>
  </ld>
</cmdstream>
```

Рис. 2.2. Пример выходного файла потока команд после компонента *Driver Environment Generator*

2.3. Шаг 2

Domain Specific C Verifier предоставляет интерфейс для статической верификации программ на языке Си, не зависящий от способа описания моделей правил и от используемого инструмента статической верификации. На вход *Domain Specific C Verifier* поступает набор команд сборки, соответствующих анализируемому модулю, наборы точек входа и идентификаторов моделей правил, которые необходимо проверить.

Для каждого идентификатора модели правила *Domain Specific C Verifier* вызывает *Rule Instrumentor*, который создает задание для соответствующего инструмента статической верификации. Взаимодействие с инструментом верификации осуществляет *Reachibility C Verifier* посредством специального адаптера, специфичного для каждого инструмента статической верификации.

Отметим, что информация о верификации, специфичная для драйверов и ядра ОС Linux, полностью скрывается в описании модели правила, поэтому компонент *Domain Specific C Verifier*, вообще говоря, может быть применен и для других предметных областей в том случае, если предоставлены соответствующие описания моделей правил корректности, набор команд сборки и исходный код анализируемой программы. Для примера, уже на сегодняшний день в базе моделей правил описано общее правило, проверяющее, что в программе не нарушается ни один *assert*. Данное правило может быть применено для произвольной программы на языке Си.

Rule Instrumentor – компонент, основное назначение которого – связывать формализованное представление моделей правил корректности с исходным кодом проверяемой программы для его последующей верификации с помощью некоторого инструмента статической верификации.

Помимо исходного кода программы и команд сборки на вход *Rule*

Instrumentor поступает идентификатор модели правила. Информация о моделях хранится в базе данных моделей. Используя идентификатор, *Rule Instrumentor* получает описание соответствующей модели, которое, по сути, состоит из путей к так называемым *аспектным файлам* и вспомогательной информации для инструмента верификации, которая способствует выполнению более качественной и быстрой проверки.

Аспектные файлы пишутся на аспектно-ориентированном расширении языка программирования Си наподобие того, как это делается с помощью SLIC в проектах SDV и Avinux. Данный подход позволяет описывать модели ошибки, независимым от используемого инструмента статической верификации образом, что важно ввиду нацеленности системы верификации *LDV* на использование различных инструментов статической верификации. Кроме того, в данной работе используется реализация аспектно-ориентированного программирования, описанная в статье Е.М.Новикова [55], набор средств которой для моделирования ошибок превосходит аналоги для языка Си.

Для упрощения сопровождения в условиях непрерывного развития ядра написание аспектов осуществляется так, что описания точек соединения, т.е. программных конструкций, к которым осуществляется привязка, отделены от модели, проверяющей правило, таким образом, что при изменении названий функций, атрибутов, параметров, не влияющих на модель, требуется изменять только описания точек соединения, а модель при этом остается неизменной. Если же меняется семантика функций, к которым осуществляется привязка, то изменять требуется только соответствующие модели.

Инструментирование исходного кода драйвера на основе аспектов осуществляется специальным инструментом автоматически. Так как инструментированный исходный код должен обрабатываться инструментами статического анализа кода, на выходе после инструментирования создается программа на Си, которая представляет собой препроцессированный исходный код

драйвера, дополненный описанием модели соответствующего правила.

Более подробно процесс проектирования моделей описан в разделе 2.5.

2.4. Шаг 3

На сегодняшний день для анализа инструментированного исходного кода драйверов ОС Linux используются так называемые *инструменты анализа (верификации) достижимости*, то есть инструменты статической верификации, предназначенные для выявления нарушений правил корректности, выраженных в виде достижимости ошибочной точки в программе (подробнее в главе 4). Тем не менее, в дальнейшем планируется использовать и другие классы инструментов верификации, например, решающих задачу завершаемости, что не потребует внесения значительных изменений в архитектуру системы верификации.

Инструменты верификации достижимости выносят один из трех вердиктов: Safe, Unsafe и Unknown. Вердикт Safe означает, что соответствующий инструмент статической верификации гарантирует отсутствие нарушений проверяемого правила корректности. Вердикт Unsafe говорит об обнаружении нарушения правила и сопровождается более детальной информацией о проблеме. Такой информацией, как правило, является трасса ошибки, которая показывает путь выполнения программы, приводящий к ошибочному состоянию. Вердикт Unknown означает, что инструмент по тем или иным причинам (например, нехватка памяти или времени) не смог найти ответ на поставленный вопрос.

Компонент *Reachibility C Verifier* решает задачу преобразования задачи верификации из представления в виде набора команд сборки и настроек верификации в представление конкретного инструмента верификации. Данный компонент получает на вход инструментированный исходный код драйверов

со сгенерированными точками входа и ошибочными метками. На данном исходном коде *Reachibility C Verifier* вызывает заданный адаптер для соответствующего инструмента верификации достижимости.

В типовом адаптере инструмента статической верификации можно выделить следующие четыре части:

- подготовка входных файлов;
- подготовка обработчика вывода инструмента верификации;
- запуск инструмента;
- обработка результатов.

Подробнее о разработке адаптера написано в главе 5.

На сегодняшний день предложенный интерфейс интеграции со сторонними инструментами верификации был опробован на примере инструментов статической верификации BLAST и CPAchecker. Было выявлено, что предоставленного интерфейса достаточно, чтобы обеспечить нужды этих инструментов по интеграции в систему верификации *LDV*, описанную в главе 5.

Вердикт о результате верификации и другая информация, выдаваемая инструментом статической верификации, обрабатывается всеми компонентами на Рис. 2.1 в обратном порядке. При этом каждый компонент дополняет отчет информацией о своей работе, после чего формируется финальный отчет о проверке всего задания.

2.5. Построение моделей правил

Рассмотрим более подробно, что из себя представляет правило корректности и как представляется его модель.

ID 0077: Выделение памяти с флагом NOIO в контексте usb_lock
ОПИСАНИЕ: Выделение памяти с флагом *GFP_KERNEL* может использовать операции ввода/вывода с устройством хранения, которые могут завершаться с ошибкой, что приводит к необходимости сброса состояния утстройства. Поэтому флаг *GFP_KERNEL* не может использоваться между вызовами *usb_lock_device()* и *usb_unlock_device()*. Вместо него следует использовать флаг *GFP_NOIO*.
ССЫЛКИ: Пример исправления: http://git.kernel.org/?p=linux/kernel/git/stable/ linux-stable.git;a=commitdiff; h=186c74d336e2c1377df9e5dc88f7966b2dd6acf7

Рис. 2.3. Пример правила корректности

Пример правила корректности показан на рисунке Рис. 2.3. Данное правило выделено на основе анализа изменений в стабильных ветках ядра, ссылка на изменение приведена в разделе *ССЫЛКИ*. Методика выделения правил описана в главе 6.

Модель правила описывается в базе моделей, в которой для каждой модели задается:

- идентификатор модели;
- идентификатор правила для которого написана эта модель;
- статус завершенности правила;
- имя метки ошибки (например LDV_ERROR);
- произвольное описание;
- файл аспект, содержащий так называемые рекомендации (advices), которые применяются к каждому файлу драйвера, и общую модель, которая содержит определения модельных функций и переменных состояния;
- подсказки инструменту верификации, такие как опции запуска, дополнительная информация о модели; формат подсказок определяется адаптером к инструменту верификации;

Для построения моделей правил используется подход аспектно-ориентированного программирования (АОП), описанный в работе [55].

Правило формализуется в аспекте – специальном отдельном модуле. Аспект состоит из набора рекомендаций (advice).

Каждая *рекомендация* состоит из *среза* и *тела* рекомендации. В теле указываются действия, которые должны быть выполнены для тех точек соединения в программе, которые задаются с помощью среза (см. далее). Тело

рекомендации записывается с помощью инструкций языка программирования Си, на котором пишутся драйверы. АОП предоставляет средства, позволяющие использовать в теле рекомендации информацию о соответствующей точке соединения, например, имя вызываемой функции, типы ее аргументов и т.п.

Также в рекомендации указывается, должны ли эти действия быть выполнены *до (before)*, *после (after)* или *вместо (around)* выполнения точки соединения программы.

Срез (pointcut) – это набор *точек соединения (join point)*, т.е. программных конструкций, к которым осуществляется привязка. Например, срезом могут быть все вызовы функций выделения памяти (таких как malloc, calloc и т.д.). Срез, который состоит из точек соединения, соответствующих одной программной конструкции, называется *примитивным срезом (primitive pointcut)*. Комбинации срезов, которые можно получить с помощью операторов “или” – объединение, “и” – пересечение, “не” – исключение и оператора группировки (скобок), называются *составными срезами (composite pointcut)*. Поддерживаются следующие *примитивные срезы*:

- *call(function signature)* – описывает точки вызова функций (function signature = return type + function name + arguments types);
- *execution(function signature)* – описывает точку определения тела функции (если тела нет, то рекомендация не применяется).
- *define(macro signature)* – описывает точки макроопределений (обращаться к параметрам в теле рекомендаций можно по тем именам, которые заданы в соответствующем срезе);
- *file("$this")* – текущий файл;
- *set(variable signature)* – точки присваивания значения переменной;

- *get(variable signature)* – точки чтения значения переменной;
- *introduce(type signature)* – точки определения типов;

Именованный срез (named pointcut) представляет собой примитивный или составной срез, который связан с некоторым именем, посредством которого данный срез можно использовать.

pointcut A: pointcut1 || pointcut2 – присваивает имя (в данном случае – "A") вашему срезу.

Для аспектов можно выделять *переиспользуемые блоки* аналогично тому, как это делается с помощью препроцессора языка Си. Поддерживаются директивы: включение аспектного файла, аспектное макроопределение, аспектные #ifdef, #endif.

2.6. Отделение привязки к интерфейсу ядра

Для упрощения сопровождения в условиях непрерывного развития ядра точки соединения в правилах описываются отдельно от основной части модели правила. Для этого используются именованные срезы. Например, для правила 0077 на Рис. 2.3 фрагменты привязки выглядят следующим образом:

```
/* Функции имеющие флаг gfp_t третьим параметром */
pointcut ARG_3: execution(static inline void *kcalloc(..))
    || call(void * krealloc(..))
    || call(struct sk_buff *__netdev_alloc_skb(..))
    || call(void *usb_alloc_coherent(..))
    || call(int mempool_resize(..))
/* Функции захвата и освобождения usb блокировки */
pointcut USB_LOCK : define(usb_lock_device(udev))
pointcut USB_UNLOCK : define(usb_unlock_device(udev))
```

В самой модели можно использовать именованный срез по имени:

```
before: ARG_3
{
  ldv_check_alloc_flags($arg3);
}
around: USB_LOCK
{
  ldv_usb_lock_device()
}
around: USB_UNLOCK
{
  ldv_usb_unlock_device()
}
```

Таким образом, при изменении названий функций, атрибутов, параметров, не влияющих на модель, требуется изменять только точки соединения, а модель при этом остается неизменной.

Рассмотрим пример модели для правила 0077.

В модельном состоянии отражаем, захвачена ли USB блокировка.

```
int ldv_lock = 1;/*в начале свободна*/
```

Проверка условия правила: если USB блокировка захвачена (ldv_lock == 2), то функции выделения памяти должны вызываться с флагами GFP_ATOMIC или GFP_NOIO.

```
void ldv_check_alloc_flags(gfp_t flags)
{
  if (ldv_lock == 2)
    ldv_assert(flags == GFP_NOIO || flags == GFP_ATOMIC);
}
```

Функция захвата USB блокировки:

```
void ldv_usb_lock_device(void)
{
  ldv_lock = 2;
}
```

Функция освобождения USB блокировки:

```
void ldv_usb_unlock_device(void)
{
  ldv_lock = 1;
}
```

Глава 3

Метод генерации окружения целевого драйвера

Как уже упоминалось, для работы статических верификаторов драйвер нужно заключить в некоторое “окружение” с тем, чтобы программный код был замкнутым. Такое окружение будем называть “синтезируемым окружением”. К синтезируемому окружению предъявляются две группы требований. Первая – связана с тем, что оно должно воспроизводить те же сценарии взаимодействия с драйвером, что и реальное окружение драйвера в операционной системе. Вторая группа требований определяет ограничения на конструкцию (структуру) синтезируемого окружения, обусловленную возможностями современных средств статической верификации.

Предлагаемый метод построения состоит из двух шагов. Сначала строится логическая модель реального окружения (с учетом специфики каждого вида драйверов). Эта модель учитывает многопоточность окружения и асинхронность взаимодействия драйвера с окружением. Для формального описания модели используется π-исчисление [56]. Затем строится Си-программа, которая соответствует данной модели, и при этом является однопоточной и отвечает ограничениям инструментов верификации. Поскольку трансформация многопоточной модели в однопоточную программу не является тривиальной, возникает необходимость доказать корректность построения синтезируемого окружения. Дается определение эквивалентности между моделью в π-исчислении и программой на языке Си, в котором учитываются трассы, содержащие только общие действия для процессов драйвера и окружения с ограничением на переключения при выполнении функций-обработчиков драйвера. В диссертации сформулирована и доказана теорема о том, что для каждого драйвера логическая модель окружения эквивалентна синтези-

рованному окружению для того же самого драйвера.

3.1. Сценарии взаимодействия

Одно из требований к генерации окружения состоит в том, что оно должно включать все возможные сценарии взаимодействия, что и реальное окружение, и при этом не содержать тех сценариев, которых нет в реальном окружении. Если в синтезированном окружении какой-то сценарий взаимодействия отсутствует, то это может привести к упущенным ошибкам. Если же в синтезированном окружении имеется сценарий, которого нет в реальном окружении, то это может привести к ложным сообщениям об ошибках.

Сценарии взаимодействия состоят из вызова окружением функций-обработчиков драйвера. Начинается выполнение с вызова функции инициализации драйвера, а заканчивается вызовом функции выхода. Драйвер при этом может вызывать библиотечные функции окружения, обращаться к глобальным переменным. Одним из важных видов библиотечных функций, предоставляемых окружением, являются функции регистрации и дерегистрации структур обработчиков. Структура – это набор функций-обработчиков заданных типов. Например, структрура $file\_operations$ включает в себя такие функции-обработчики как $open$, $release$, $read$, $write$, ... После того как структура обработчиков зарегистрирована драйвер может вызывать входящие в нее функции-обработчики.

На Рис. 3.1 представлены выдержки из кода драйвера. Функция $s21\_init$ является функцией инициализации драйвера, $s21\_exit$ функцией выхода. Функция инициализации в процессе своей работы регистрирует структуру обработчиков $file\_operation$ с функциями обработчиками $open = s21\_open$, $release = s21\_release$, $read = s21\_read$, $write = s21\_write$, вызывая библиотечную функцию $register\_chrdev$. Если регистрация происходит

```
static int s21_open(struct inode *inode, struct file *file)
{
...
}
...
static struct file_operations s21_ops = {
      .open = s21_open,
      .release = s21_release,
      .read = s21_read,
      .write = s21_write
};

static int s21_init(void) {
   int err;
   ...
   err = register_chrdev(INPUT_MAJOR, "input &s21_fops);
   if(err) {
      goto fail;
   }
   ...
   return 0;
   fail:
      return err;
}
...
module_init(s21_init);
module_exit(s21_exit);
```

Рис. 3.1. Выдержки из исходного кода драйвера s21

успешно, то окружение может вызывать функции-обработчики структуры $file\_operations$.

В ядре ОС Linux существует набор правил, которым должно подчинять-ся окружение:

1. Правила, устанавливающие ограничения:
   1.1. на порядок вызова обработчиков одной струкутры;
   1.2. на контекст вызова и передаваемые значения обработчиков структур;
   1.3. на порядок вызова обработчиков нескольких структур;
2. Правила обработки вызовов:
   2.1. функций регистрации/дерегистрации структур;
   2.2. библиотечных функций;

Например, в рамках одной структуры $file\_operations$, окружение может осуществлять вызов функций $read$, $write$, только если был успешный вызов $open$. Также при вызове $read$ должен передаваться тот же параметр $file$, который был успешно открыт $open$. Другой пример, перед вызовом некоторых функций требуется, чтобы был захвачен мьютекс. В качестве примера ограничения на порядок вызова обработчиков нескольких структур, можно привести пример со структурами usb_driver, $file\_operations$, в котором вызов функций из file_opeartions возможен, только после успешной регистрации $file\_operations$ в функции $probe$ из usb_driver. Другой пример, после успешного вызова open окружение должно обеспечить, чтобы драйвер нельзя было выгрузить, т.е. нельзя вызывать функцию выхода.

Для моделирования окружения с учетом данных правил мы будем использовать π-исчисление.

3.2. Основные определения

Определим понятия, неоходимые в π-исчислении. В определении "The Polyadic π-Calculus: a Tutorial, Robin Milner, October 1991"[56] допускается передача любых меток в качестве параметра сигнала. В определении, используемом в данной работе, прямая передача меток запрещена. Для передачи связей между процессами используются динамически связываемые метки приема F и посылки $\bar{F}$. Обозначим $\Phi = F \cup \bar{F}$. Динамически связываемые метки имеют параметры со значениями из множества Π, которые пишутся в квадратных скобках после метки.

Динамически связываемые метки приема сигнала $f[p_1]$ и посылки сигнала $\bar{f}[p_2]$ осуществляют взаимодействие, только если значения параметров меток совпадают. Т.е. процесс получатель принимает сигнал $f[p_1]$, только если значение передаваемого параметра метки p_1 совпадает со значением получателя p_2. Процесс отправитель считает сигнал $\bar{f}[p_2]$ отправленным, только если нашелся получатель с совпадающим значением параметра метки p_1.

Отметим, что введение динамических меток необходимо нам лишь для удобства записи трансляции в Си программу. Передаваемые метки из классического определения $\Delta = D \cup \bar{D}$ могут быть сведены к динамическим введением двух меток приема сигнала f и отправки сигнала $\overline{f}$ с параметрами D.

Множество статических меток приема сигналов обозначим как A. Множество статических меток отправки сигнала обозначим как $\bar{A}$, оно состоит из меток $\overline{\alpha}$, где $\alpha \in A$. Обозначим $\Lambda = A \cup \bar{A}$.

Множество действий включает в себя действия отправки/получения для статических и динамических меток, а также специальное пустое действие τ, $Act = \Lambda \cup \Phi \cup \{\tau\}$

Непустые действия могут иметь параметры, которые записываются в

круглых скобках. Для действий посылки сигнала $\overline{\alpha(e(x))}$ вычисляется значение выражения $e(x)$, на которое заменяется переменная y в действии приема сигнала $\alpha(y)$.

Кроме того, в определении для удобства записи добавлен оператор if-then-else.

Определение 1. *Процессы нижнего уровня:*

$$K(\mathbf{x}) ::= 0 \mid \sum_{i=1}^{n} \alpha_i(y_i).K_i(e_i(\mathbf{x}, y_i))$$

$$\mid \textbf{if } b(\mathbf{x}) \textbf{ then } K_1(\mathbf{x}) \textbf{ else } K_2(\mathbf{x})$$

Процессы верхнего уровня:

$$P ::= E_1 | \cdots | E_k$$

$$E ::= K(\textbf{const}) \mid (\nu p)K(\textbf{const})$$

$$\mid\ !\alpha_K(\mathbf{x}).K(\mathbf{x}) \mid\ !(\nu p)\alpha_K(\mathbf{x}).K(\mathbf{x})$$

где

- 0 – *пустой процесс;*
- $K(\mathbf{x})$ – *задание идентификатора процесса, где* K – *имя процесса,* $\mathbf{x}$ – *вектор переменных;*
- $\sum_{i=1}^{n} \alpha_i(y_i).K_i(e_i(\mathbf{x}, y_i))$ – *выбор, процесс может продолжаться одним из вариантов* $\alpha_i(y_i).K_i(e_i(\mathbf{x}, y_i))$; α – *действие* $\in Act$;

  1. *Метки* $\alpha_i \in \Lambda$.
     - $a(y_i).K_i(e_i(\mathbf{x}, y_i))$ – *прием сигнала;*
     - $\overline{a(e(\mathbf{x}))}.K_i(e_i(\mathbf{x}))$ – *отправка сигнала (в качестве параметра может передаваться значение из* Π*);*

2. *Динамически связываемые метки* $\alpha_i \in \Phi$.

   – $f[p](y_i).K_i(e_i(\mathbf{x}, y_i))$ – *прием динамического сигнала;*

   – $\overline{f[p](e(\mathbf{x}))}.K_i(e_i(\mathbf{x}))$ – *отправка динамического сигнала;*

3. *Метка* $\alpha_i = \tau$. *Процесс продолжается как* $K_i(e_i(\mathbf{x}))$.

- $E_1|\cdots|E_k$ – *параллельная композиция, одновременно выполняются процессы* $E_1, \ldots, E_k$;

- $!\alpha_K(\mathbf{x}).K(\mathbf{x})$ – *создание копии процесса;*

- $(\nu p)K$ – *создание нового значения параметра из* Π *и присваивание его в переменную* p.

Для сокращенного задания идентификаторов процессов нижнего уровня будем использовать запись:

$N_j(\mathbf{x}) \stackrel{\text{def}}{=} \ldots.\langle N_i \rangle$ выражение1 $\ldots$

которая означает означает:

$N_j(\mathbf{x}) \stackrel{\text{def}}{=} \ldots.N_i(\mathbf{x})\ldots$

$N_i(\mathbf{x}) \stackrel{\text{def}}{=}$ выражение1

Вспомогательное отношение структурной эквивалентности $\equiv$ определяется аналогично [56].

Определим отношение редукции $\rightarrow$ над процессами аналогично [56], при этом пометим пары в отношении метками α из $A \cup F \cup \{\tau\}$, т.е. метками приема сигналов и пустой меткой τ.

Определение 2. *Отношение редукции* $\stackrel{\alpha}{\longrightarrow}$ *над процессами – это наименьшее отношение, удовлетворяюее следующим правилам:*

- *Если есть определение процессов* $K(\mathbf{x})$ *и* $N(\mathbf{x})$, *в каждом из которых есть слагаемое, которое может взаимодействовать с другим, то это*

взаимодействие осуществляется с меткой приема и значениями переданных параметров.

$$COMM_1(\alpha \in A): \frac{\begin{array}{c} K(\mathbf{x}) ::= (\cdots + \alpha(y).K_i(e_i(\mathbf{x}, y))) \\ N(\mathbf{x}) ::= (\cdots + \overline{\alpha(e(\mathbf{x}))}.N_j(e_j(\mathbf{x}))) \end{array}}{K(\mathbf{a}) \mid N(\mathbf{b}) \stackrel{\alpha(e(\mathbf{b}))}{\longrightarrow} K_i(e_i(\mathbf{a}, e(\mathbf{b}))) \mid N_j(e_j(\mathbf{b}))}$$

- *Для динамических меток сравниваются также значения параметров ($\mathbf{x}_i$ – i-ый элемент вектора $\mathbf{x}$).*

$$COMM_2(\alpha \in F): \frac{\begin{array}{c} K(\mathbf{x}) ::= (\cdots + \alpha[\mathbf{x}_i](y).K_i(e_i(\mathbf{x}, y))) \\ N(\mathbf{x}) ::= (\cdots + \overline{\alpha[\mathbf{x}_j](e(\mathbf{x}))}.N_j(e_j(\mathbf{x}))) \\ \mathbf{a}_i = \mathbf{b}_j \end{array}}{K(\mathbf{a}) \mid N(\mathbf{b}) \stackrel{\alpha[\mathbf{a}_i](e(\mathbf{b}))}{\longrightarrow} K_i(e_i(\mathbf{a}, e(\mathbf{b})))) \mid N_j(e_j(\mathbf{b}))}$$

- *Переход по τ действию может осуществляться всегда.*

$$TAU: \frac{K(\mathbf{x}) ::= (\cdots + \tau.K_i(e_i(\mathbf{x})))}{K(\mathbf{a}) \stackrel{\tau}{\longrightarrow} K_i(e_i(\mathbf{a})))}$$

- *if-then-else оператор.*

$$IFTHEN: \frac{\begin{array}{c} K(\mathbf{x}) ::= \textbf{if } b(\mathbf{x}) \textbf{ then } K_1(\mathbf{x}) \textbf{ else } K_2(\mathbf{x}) \\ b(\mathbf{a}) = TRUE \end{array}}{K(\mathbf{a}) \stackrel{\tau}{\longrightarrow} K_1(\mathbf{a})}$$

$$IFELSE: \frac{\begin{array}{c} K(\mathbf{x}) ::= \textbf{if } b(\mathbf{x}) \textbf{ then } K_1(\mathbf{x}) \textbf{ else } K_2(\mathbf{x}) \\ b(\mathbf{a}) = FALSE \end{array}}{K(\mathbf{a}) \stackrel{\tau}{\longrightarrow} K_2(\mathbf{a})}$$

- *В параллельной композиции редукция компонентов может осуществляться независимо.*

$$PAR: \frac{K(\mathbf{a_1}) \stackrel{\alpha}{\longrightarrow} K'(\mathbf{a_2})}{K(\mathbf{a_1}) \mid N(\mathbf{b}) \stackrel{\alpha}{\longrightarrow} K'(\mathbf{a_2}) \mid N(\mathbf{b})}$$

- *Создание нового значения.*

$$RES : \frac{K(\mathbf{a_1}) \xrightarrow{\alpha} K'(\mathbf{a_2})}{(\nu x)K(\mathbf{a_1}) \xrightarrow{\alpha} (\nu x)K'(\mathbf{a_2})}$$

- *Редукция для структурно эквивалентных процессов.*

$$STRUCT : \frac{K \equiv N;\ \ N(\mathbf{a_1}) \xrightarrow{\alpha} N'(\mathbf{a_2});\ \ N' \equiv K'}{K(\mathbf{a_1}) \xrightarrow{\alpha} K'(\mathbf{a_2})}$$

Понятие редукции позволяет нам определить множество возможных трасс выполнения π процесса P, обозначаемое как $traces_\pi(P)$. Для редукции $P \xrightarrow{\alpha_1} P_1 \xrightarrow{\alpha_2} P_2 \cdots \xrightarrow{\alpha_n} P_n$, определим трассу как подпоследовательность $\alpha_{i_1}, \ldots, \alpha_{i_k}$ последовательности $\alpha_1, \ldots, \alpha_n$, включающую все непустые действия $\alpha_{i_j} \neq \tau$. Тогда $traces_\pi(P)$ – это множество трасс для всех возможных редукций процесса P. Определим $traces_\pi(P, D)$, где $D \subseteq F \cup A$, как множество трасс, в которых уделены метки не из множества D (т.е. включаем только $\alpha_{i_j} \in D$).

3.3. Формальная модель драйвера и его окружения в π-исчислении

Драйвер предоставляет сердцевине ядра функции инициализации и выхода, представляемые процессами P_{init}, P_{exit}, которые получают сигналы $init(p)$, $exit(p)$ и отвечают на них $\overline{init^{ret}[p](x)}$, $\overline{exit^{ret}[p]}$.

Во время выполения инициализации, выхода, а также при выполнении других функций, драйвер может:

- обращаться к библиотечным функциям сердцевины ядра $\overline{g_1(p_1, params_1)}$, $\ldots$, $\overline{g_l(p_l, params_l)}$ и получать ответы $g_1^{ret}[p_1](result_1)$, $\ldots$, $g_l^{ret}[p_l](result_l)$,

- обращаться к глобальным переменным, с помощью сигналов $\overline{set_{v_i}(x)}$, $get_{v_i}(x)$ к процессам P_{v_i}.

После регистрации структуры, окружение может вызывать функции обработчики через процесс P_{fcall}: $\overline{f(p_i, f_i, ctx_i, params_i)}$, $f^{ret}[p_i](result_i)$. Причем процессы обработчиков копируются при каждом вызове, т.е. $P_{fcall} \stackrel{\text{def}}{=} !f(p_i, f_i, ctx_i, params_i).Q$, где Q – продолжение процесса.

Таким образом, драйвер представляет собой набор процессов, запущенных параллельно:

$$Drv_\pi \stackrel{\text{def}}{=} P_{init} \mid P_{exit} \mid P_{fcall}$$

Окружение предоставляет обработку:

- вызовов библиотечных функций $g_1(p_1, params_1)$, ..., $g_l(p_l, params_l)$;
- обращений к глобальным переменным, с помощью сигналов $\overline{set_{v_i}\langle x\rangle}$, $get_{v_i}(x)$.

Окружение осуществляет вызовы:

- функций инициализации и выхода драйвера $\overline{init(p)}$, $\overline{exit(p)}$;
- обработчиков для зарегистрированных структур $\overline{f(p_i, f_i, ctx_i, params_i)}$.

В общем случае окружение это набор процессов:

$$Env_\pi \stackrel{\text{def}}{=} P_1 \mid \cdots \mid P_m$$

3.4. Примеры задания окружения в виде процессов с учетом правил взаимодействия

Рассмотрим пример процесса S_{fops}, моделирующего структуру $file\_operations$:

S_{fops} ::= $!start_{fops}(id, file, open, release, read, write)$ $.S_1(id, file, open, close, read, write)$

$S_1(id, file, open, release, read, write)$ ::= $stop_{fops}[id].0 + \overline{tmg}$ $.\langle N_1\rangle tmg^{ret}(r_1)$ $.\langle N_2\rangle$ **if** $r_1 \neq 0$ **then** S_1 **else** $\langle N_3\rangle$ $\overline{f(id, open, file)}.\langle N_4\rangle$ $f^{ret}[id](r_2)$ $.\langle N_5\rangle$ **if** $r_2 \neq 0$ **then** S_1 **else** S_2

$S_2(id, file, open, release, read, write) ::= \overline{f(id, read, file)}.\langle N_6\rangle f^{ret}[id](r_3).S_2$ $+$ $\overline{f(id, write, file)}.\langle N_7\rangle f^{ret}[id](r_4).S_2$ $+$ $\overline{f(id, release, file)}.\langle N_8\rangle$ $f^{ret}[id].\langle N_9\rangle\overline{mput}.\langle N_{10}\rangle mput^{ret}.S_1$

Правила взаимодействия 1.1 и 1.2 из подраздела 3.1 учитываем в состояниях процесса S_{fops}.

Ограничения на порядок вызова обработчиков нескольких структур 1.3 учитываются за счет передачи сообщений между ними.

Процесс trymoduleget моделирует захват модуля драйвера, так чтобы его нельзя было выгрузить, т.е. окружение не вызвало функцию выхода $exit$. Процесс, захватывающий модуль, посылает сигнал $\overline{tmg}$, и если получает ответ $tmg^{ret}(true)$, то модуль успешно захвачен. Сигнал $mstop$ нужен для того, чтобы процесс не допускал новые захваты.

$P_{trymoduleget} ::= M(0)$

$M(0)$ ::= $tmg.\langle M_1\rangle\overline{tmg^{ret}(true)}.M(1) + mstop.MD(0) + iszero.\langle M_2\rangle\overline{iszero^{ret}(true)}.M(0)$

$MD(0) ::= mstop.MD(0) + iszero.\langle MD_1\rangle\overline{iszero^{ret}(true)}.MD(0)$

Для $i \geq 1$:

$M(i) ::= tmg.\langle M_3\rangle\overline{tmg^{ret}(true)}.M(i+1) + mput.\langle M_4\rangle\overline{mput^{ret}}.M(i-1) + mstop.MD(i) + iszero.\langle M_5\rangle\overline{iszero^{ret}(false)}.M(i)$

$MD(i)$ ::= $mp.\langle MD_2\rangle\overline{mput^{ret}}.M(i-1) + mstop.MD(i) + iszero.\langle MD_3\rangle\overline{iszero^{ret}(false)}.MD(i)$

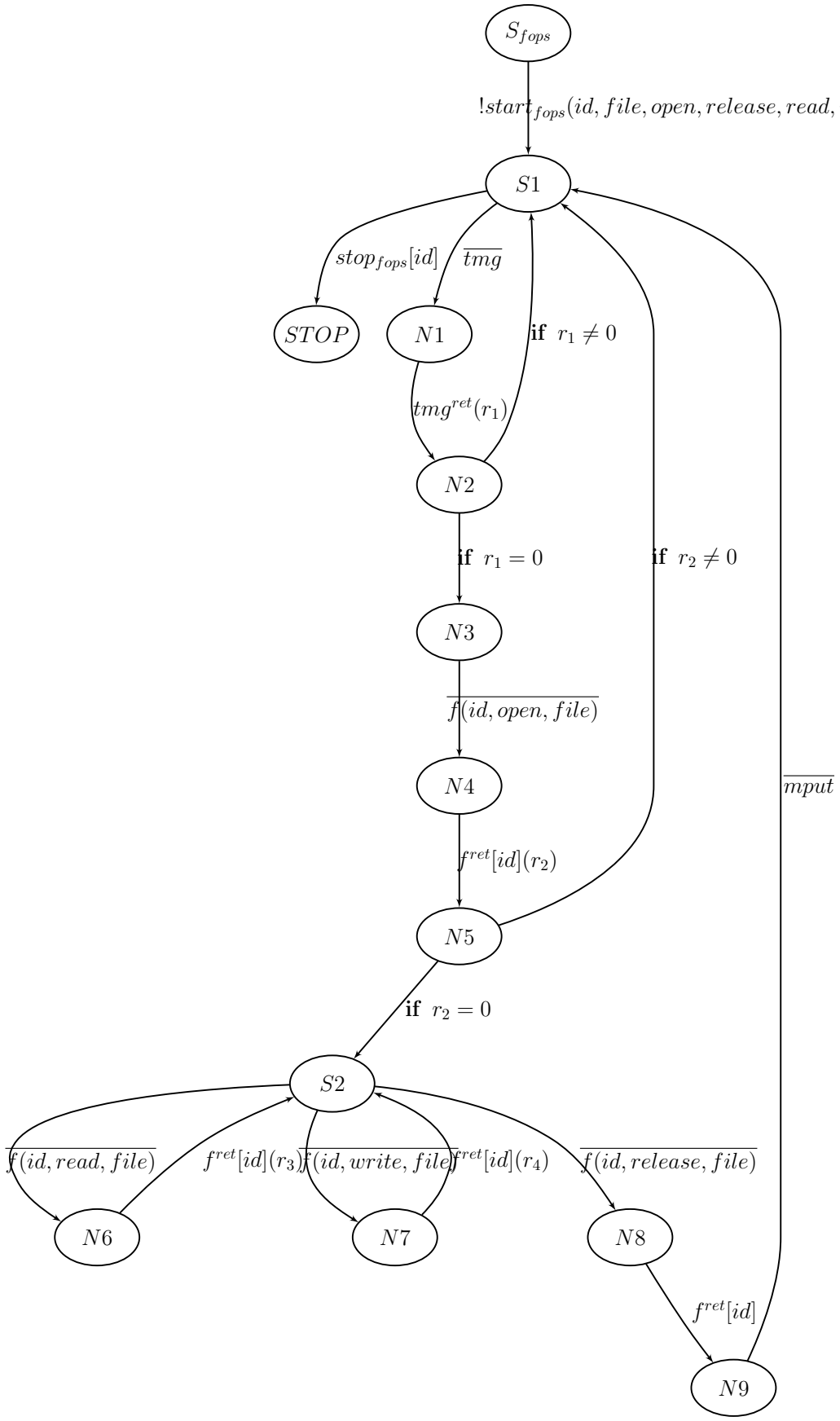

Рис. 3.2. Пример процесса S_{fops}, моделирующего структуру $file\_operations$

3.5. Трансляция процессов в Си программу

Трансляция процессов описана в приложении А.

В разделе А.1 описан общий метод трансляции в многопоточную программу, а в разделе А.2 описывается его модификация для трансляции в последовательную Си программу.

Далее мы покажем, что получающаяся в результате трансляции последовательная Си программа порождает те, и только те трассы взаимодействий, которые были в π-процессах. Так как исходный код драйвера в последовательном случае выполняется в одном потоке, то мы будем требовать, чтобы окружение в виде π-процессов одновременно осуществляло вызов только одной функции драйвера, т.е. после посылки сигнала $\overline{f}$ и до приема сигнала возврата f^{ret}, недопустимо посылать еще один сигнал $\overline{f}$.

Определим понятие трассы выполнения для последовательной Си программы CP. Трасса Си программы определяется событиями вызова функции приема сигнала в сгенерированном коде. Данные функции генерируются в пункте 2.3 приложения А метода трансляции. Функции приема сигналов для каждой метки $a[p](y) \in A \cup F$ имеют вид:

int a(int p, type y)

Функции возвращают значение 0, если сигнал успешно принят, и -1 иначе. Событие $a[p](y)$ добавляется в трассу, если функция $a(p, y)$ завершается с кодом 0. Множество всех трасс обозначим как $traces_C(CP)$. Определим $traces_C(CP, D)$, где $D \subseteq F \cup A$, как множество трасс, в которых удалены метки не из множества D.

Определим понятие трассовой эквивалентности для π-процесса P и сгенерированной Си программы CP.

Определение 3. *π-процесс P трассово-эквивалентен сгенерированной Си программе CP на множестве событий D, обозначим как $P \approx_D CP$, если*

$traces_\pi(P, D) = traces_C(CP, D)$.

Теорема 1. *Пусть D – множество событий приема сигналов между драйвером и окружением $D = \{init, init^{ret}\} \cup \{exit, exit^{ret}\} \cup \{f, f^{ret}\} \cup \{g_1, g_1^{ret} \ldots, g_l, g_l^{ret}\} \cup \{set_{v_i}\} \cup \{get_{v_i}\}$.*

Пусть Sys_π – модель в π исчислении, состоящая из процессов окружения и процессов драйвера, т.е. $Sys_\pi = Env_\pi \mid Drv_\pi$.

Пусть Sys_C – Си код, состоящий из кода, сгенерированного для окружения Env_C (по методу А.2), и кода драйвера Drv_C.

Пусть $Drv_\pi \approx_D Drv_C$, т.е. код драйвера трассово-эквивалентен π модели.

Тогда $Sys_\pi \approx_D Sys_C$.

Таким образом, для любой трассы π процессов существует эквивалентная трасса в Си программе, и наоборот для любой трассы Си программы существует эквивалентная трасса π-процессов.

Доказательство теоремы приведено в разделе А.3 приложения А.

Глава 4

Методы оптимизации анализа при помощи предикатных абстракций

В данной главе описываются предложенные диссертантом методы оптимизации, использованные в инструменте BLAST. Название инструмента BLAST – это сокращение от “Berkeley Lazy Abstraction Software verification Tool”, то есть “Инструмент верификации программ с применением ленивой абстракции, разработанный в Беркли”. Впервые BLAST упоминается в статье [57], датированной 2002-м годом. Инструмент развивался вплоть до июля 2008 (версия 2.5). Начиная с 2009 работа над ним была продолжена в ИСП РАН. Предложенные диссертантом методы оптимизации реализованы в версии 2.7.

Таким образом, в работе рассматриваются версии:

- BLAST 2.5 – Беркли, 2008.
- BLAST 2.7 – ИСП РАН, 2012.

Далее в главе рассматриваются:

- общие принципы и составляющие метода предикатной абстракции;
- особенности реализации этих принципов в инструменте BLAST;
- усовершенствования в инструменте, реализованные в версии BLAST 2.7.

4.1. Общая информация

BLAST распространяется под лицензией Apache 2.0 как в виде исходного кода, так и собранным в бинарном виде под ОС Linux. Хотя BLAST включает в себя компоненты, написанные на различных языках программирования, его основная часть, реализующая алгоритмы верификации, написана на языке OCaml и совместима с версиями OCaml 3.11–3.12. BLAST работает под Linux, но собрать его под Windows также возможно с помощью Cygwin.

OCaml – это язык с автоматическим управлением памятью, поэтому операции выделения памяти и сборки мусора при его использовании могут занимать значительное время. Эта проблема проявилась и в BLAST. Путем подстройки некоторых документированных опций виртуальной машины OCaml, было уменьшено время, затрачиваемое на операции с памятью, что позволило на некоторых программах добиться 20% прироста количества рассмотренных точек проверяемой программы на затраченную единицу памяти.

4.2. Метод CEGAR

Инструмент BLAST реализует метод статического анализа кода CEGAR(Counter Example Guided Abstraction Refinement)[58], основанный на абстракции и уточнении на основе контрпримеров. BLAST реализует CEGAR методом “ленивой абстракции”, который позволяет ему выполнять меньше работы, не уточняя ту часть абстракции, которая не должна измениться после анализа контрпримеров, вместо того, чтобы перестраивать ее всю целиком.

Свойства программы, которые проверяются при помощи CEGAR, формулируются в виде недостижимости заданного множества ошибочных состояний в программе. В примере на Рис. 4.1 ошибочное состояние задается оператором *assert(expr)*, который разворачивается в проверку условия, задаваемо-

```
L0:   void main() {
L1:     int x,y,z;
L2:     x = nondet_int();
L3:     y = nondet_int();
L4:     if(x>y) {
L5:       z=x-y;
        } else {
L6:       z=y-x;
        }
L7:     assert(z>=0);
L8:   }
```

Рис. 4.1. Пример 1

го выражением *expr*, и переходом на метку ERR в случае его невыполнения, т.е.

if(!expr) ERR: goto ERR;

Поставленная перед инструментом задача проверки достижимости в общем случае является алгоритмически неразрешимой, поскольку к ней может быть сведена проблема останова. Инструмент, таким образом, не гарантирует, что его анализ когда бы то ни было завершится. BLAST также может завершиться аварийно, если обнаружит, что его средств недостаточно для корректного анализа программы. У инструмента есть и другие ограничения на проверяемые программы, подробнее о них написано в секции 4.3. В этих случаях BLAST может выдать некорректный результат: найти ложную ошибку (то есть выдать вердикт Unsafe для программы, в которой нет ошибки) и доказать ложную корректность (выдать вердикт Safe для программы, в которой есть ошибка). Однако, если инструмент считает, что в программе есть ошибка, он также печатает и трассу ошибки: путь от точки входа к ошибочной метке, а ее уже может тщательно проверить пользователь на предмет того, действительно ли такое выполнение возможно.

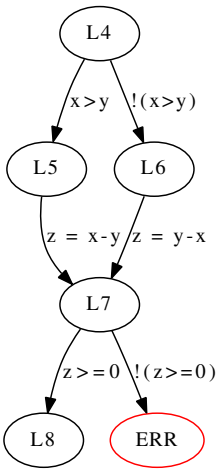

Рис. 4.2. Граф потока управления для примера 1

4.2.1. Представление программы

Перед анализом происходит разбор текста программы во внутреннее представление, в ходе которого для каждой функции в программе строится граф потока управления (ГПУ). Вершины графа соответствуют местам в тексте программы. В примере 1 (Рис. 4.1) места помечены метками $L1, \ldots, L8$. Ребра содержат операторы из текста программы, возможно, преобразованные в последовательность инструкций. Инструкции обращаются к локальным и глобальным переменным программы, к значениям в динамической памяти или к окружению системы.

Граф потока управления для данного примера представлен на Рис. 4.2. В примере каждому ребру соответствует один оператор. Условный оператор представлен двумя ребрами. Первое ребро помечено условием оператора, переход по нему осуществляется, если данное условие выполнено. Иначе переход осуществляется по второму ребру, помеченному отрицанием условия.

Для построения ГПУ BLAST использует синтаксический анализатор CIL, который строит ГПУ для каждой функции в отдельности, а во время анализа программы, как только встречает оператор вызова функции, BLAST автоматически переходит на ГПУ, соответствующий вызываемой подпрограм-

ме.

Поскольку исходный код ядра Linux использует все возможности языка Си и многие его GNU-расширения, у инструментов статического анализа могут возникать проблемы с его синтаксическим разбором. Так, например, BLAST 2.5 не мог разобрать ни одного драйвера в ядре 2.6.31. Для увеличения доли драйверов, которые BLAST может корректно разобрать, в BLAST была интегрирована последняя версия CIL (вместо версии 1.3.1 была встроена версия 1.3.8). После этого изменения, BLAST смог разбирать до 98% драйверов ядра (измерено на версии 2.6.37), и лишь около 2% приводят к ошибкам разбора.

4.2.2. Построение предикатной абстракции

Под *абстракцией* понимается такая модель программы, в которой сохраняются пути исходной программы, в том числе ошибочные. Таким образом, если в этой модели ошибочных путей не будет найдено, то в программе нет искомых ошибок. Но абстракция может содержать также пути, которых нет в исходной программе, так как возможно, что путь в абстрактной модели появился из-за абстрагирования от существенных деталей в коде программы.

Рассмотрим предикатную абстракцию, состояниями которой являются наборы предикатов, описывающих множества конкретных состояний программы. Будем строить абстракцию существования (existential abstraction), в которой при существовании перехода между двумя конкретными состояниями, всегда существует переход между любыми двумя абстрактными состояниями, включающими данные конкретные состояния. Абстракция существования гарантирует сохранение в абстрактной модели достижимых путей в ошибочное состояние. Таким образом, если абстрактная модель не содержит ошибку, то в программе ее тоже нет.

Точность приближения состояний программы зависит от используемых предикатов. Предикаты состояния могут быть выражены в логике высказываний (пропозициональной логике), с высказываниями относительно пропозициональных переменных, логике первого порядка, расширяющей логику высказываний фиксированными функциональными и предикатными символами, кванторами всеобщности и существования или логике более высоких порядков. Функциональные символы и предикаты могут иметь дополнительную интерпретацию в виде теорий.

Примерами теорий являются теории целых и вещественных чисел, теории списков, массивов, битовых векторов и т.д.

Для построения абстракции в примере нам будет достаточно использовать предикаты с целыми переменными с функциональными символами $=, \neq, >, \geq, <, \leq, +, -$ из которых мы будем составлять формулы в теории линейных неравенств.

Например, абстрактное состояние $\varphi \equiv x > 0$ задает множество конкретных состояний, в которых x принимает положительные значения ($x = 1, 2, \dots$), а все остальные переменные принимают произвольные значения.

Построение абстрактных переходов

Вычисление переходов между состояниями требует представления конструкций языка программирования в виде логических формул, и здесь, также как и для состояний, могут использоваться различные логики и теории. При вычислении переходов могут использоваться также дополнительные данные, полученные другими видами анализа, например могут использоваться данные анализа указателей для подстановки потенциально равных значений указателей в конструируемую формулу.

Если текущее абстрактное состояние представляется формулой φ, то

формулу, отражающую семантику перехода по операции *op*, будем обозначать как $SP_{op}(\varphi)$.

Формула $SP_{op}(\varphi)$ представляет множество конкретных состояний, которые достижимы из множества конкретных состояний, задаваемых формулой φ при переходе по дуге *op*.

Например, для проверки условия перехода из метки $L4$ с состоянием φ в $L5$ по оператору $x > y$, обозначим как L4:$\{\varphi\} \xrightarrow{x>y}$ L5:{?}, формула $SP_{x>y}(\varphi) \equiv \varphi \wedge (x > y)$.

Необходимо отметить, что формула состояния и формула перехода должны вводить новые переменные для изменяющихся частей состояния. На практике для этого используется форма SSA (Static Single Assignment), в которой каждой переменной значение присваивается ровно один раз. Например, для этого каждой переменной приписывается индекс, который увеличивается при каждом присваивании.

Рассмотрим пример перехода L5:$\{\varphi\} \xrightarrow{z=x-y}$ L7:{?}, где $\varphi \equiv x > y$. Пусть в формуле φ переменным были приписаны индексы $(x \to 1, y \to 1, z \to 1)$, т.е. $x_1 > y_1$. Тогда переменной z после выполнения перехода приписываем индекс 2. В результате получаем:

$SP_{z=x-y}(x > y) \equiv (x_1 > y_1) \wedge (z_2 = x_1 - y_1)$

В примере вычисление $SP_{op}(\varphi)$ производится во внутреннем представлении, которое затем конвертируется в формат SMTLIB, используемый решателями. Так как запросов к решателю довольно много, то сокрость конвертации играет существенную роль.

Ввиду большого размера формулы и неэффективного, с точки зрения операции конкатенации, представления строк в языке OCaml, эта конвертация занимала много времени. Она была улучшена путем использования более эффективных структур данных для работы со строками, сделав накладные расходы на нее практически незаметными. Полученный результат также де-

монстрирует, что тесная интеграция с решателем (к примеру, составление формул сразу же в его формате или внутреннем представлении) необязательна для эффективно работающего инструмента, реализующего CEGAR.

Использование решателей

Будем писать φ – SAT (φ выполнима), если существуют значения переменных, при которых данная формула истинна (в скобках могут указываться значения переменных, при которых формула истинна), и φ – UNSAT (φ невыполнима), иначе.

Для проверки выполнимости формулы используются специальные инструменты называемые *решателями*. В BLAST версии 2.5 по умолчанию использовался решатель Simplify[59], который был "основан на стеке". В этот стек можно было добавлять (или удалять из него) конъюнкты и проверять выполнимость получившейся конъюнкции всех подформул, которые в текущий момент находятся в стеке. Такой стековый решатель являлся идеальным вариантом для инкрементального анализа формул. Однако Simplify – устаревший проприетарный решатель. После того, как были решены проблемы со скоростью конвертации формул (см. предыдущий подраздел), стало возможным эффективное использование решателей, получающих на вход формулы в формате SMTLIB, и в комплект поставки BLAST был включен свободный решатель CVC3 [60], выпускаемый под лицензией LGPL. Возможность использовать другие решатели при этом была, разумеется, оставлена. Используемый в паре с BLAST решатель помимо результатов *формула выполнима* (SAT) и *формула невыполнима* (UNSAT) может также возвращать результат *неизвестно* (Unknown), который BLAST интерпретирует как формула выполнима (для обеспечения корректности анализа, чтобы избежать выдачи *ложной безошибочности*). При этом доказать невыполнимость формул,

которые генерирует BLAST, как правило, проще чем выполнимость. В качестве компенсации за возможную неточность (результат Unknown), решатель может иногда работать быстрее, если у него есть такая возможность. Наши эксперименты с CVC3 показали, что в “честном” режиме доказательство выполнимости формул могло занимать у него порядка минуты и гигабайта оперативной памяти.

Оказалось, что основная причина такой низкой производительности заключалась в попытках доказательства выполнимости формул с кванторами (которые нужны для отбрасывания некоторых ложных срабатываний) с помощью встроенных эвристик CVC3, связанных с инстанцированием подкванторных выражений. Эти эвристики были активированы в CVC3 по умолчанию в режиме обработки формата SMTLIB. Одной из них была эвристика полного инстанцирования, которая заключалась в многократном инстанцировании каждой аксиомы с кванторами всеми возможными термами до достижения неподвижной точки. На это уходило много времени и памяти, поскольку в генерируемых BLAST формулах, как правило, большое количество переменных.

Была использована специальная опция для отключения эвристики полного инстанцирования подкванторных выражений, кроме того, был установлен меньший лимит для повторных инстанцирований. Это сильно сократило потребляемую решателем память и время. В то же время, это существенно не повлияло на точность верификации, поскольку кванторные аксиомы достаточно редко использовались для доказательства недостижимости в проверяемых нами драйверах.

На сегодня решатель Simplify удален из BLAST, поскольку настроенный CVC3 работает намного быстрее, являясь при этом свободным (открытым) продуктом.

Также было убрано предварительное упрощение формул, реализованное

внутри BLAST, поскольку решатели, написанные на языках более низкого уровня и специально предназначенные для работы с формулами, должны справляться с этим быстрее – и это подтверждено нашими экспериментами.

Декартова абстракция

В инструменте BLAST используется *декартова абстракция*. В декартовой абстракции предикаты могут быть соединены лишь конъюнкцией. Абстракцией является наиболее строгая конъюнкция предикатов, покрывающая состояния программы. Например, рассмотрим предикаты $b1$, $b2$.

b1: x>0,

b2: x=0.

В декартовой абстракции конкретные состояния программы приближаются состояниями:

$\varphi_1 \equiv T$

$\varphi_2 \equiv b1$

$\varphi_3 \equiv b2$

$\varphi_4 \equiv b1 \wedge b2$

$\varphi_5 \equiv F$

Состояние T представляет множество всех состояний программы. Состояние F представляет недостижимое состояние программы, т.е. нет конкретных состояний, ему соответвующих. Заметим, что формула $\varphi_4 \equiv x > 0 \wedge x = 0$ эквивалентна F, так как она ложна во всех состояниях программы.

Для декартовой абстракции вычисление состояния в окончании перехода сводится к проверке истинности каждого предиката в отдельности. Результирующее состояние является конъюнкцией из истинных предикатов. Таким образом, количество вызовов решателя при вычислении декартовой предикатной абстракции равно количеству предикатов в состоянии.

Декартова абстракция φ_C^π для множества предикатов π и состояния $SP_{op}(\varphi)$ при переходе из φ по переходу op это:

1. F – если $SP_{op}(\varphi)$ – $UNSAT$;
2. Конъюнкция из предикатов $p \in \pi$ для которых формула $SP_{op}(\varphi) \wedge \neg p$ – UNSAT, если найдется хотя бы один такой предикат;
3. T – иначе.

Абстрактный граф достижимости

Предикатная абстракция может строиться непосредственно при поиске ошибки, например, при построении абстрактного графа достижимости (АГД).

АГД содержит пути, достижимые из начального состояния программы. Вершинами АГД являются абстрактные состояния, содержащие вершину ГПУ и состояние предикатной абстракции. Ребра АГД помечены переходами из ГПУ. Построение АГД состоит в вычислении конечного состояния φ_C^π по состоянию в текущей вершине φ и переходу op, которым помечено ребро.

Построение АГД происходит одновременно с проверкой свойства достижимости ошибочного состояния, задаваемого как инструкция, помеченная специальной меткой, например ERR. При обнаружении достижимого ошибочного состояния, т.е. состояния в котором $\varphi \neq F$, построение прекращается.

Конечность дерева достижимости обеспечивается конечностью множества абстрактных состояний для конечного множества предикатов π и тем, что пути из абстрактных состояний, покрываемых другими абстрактными состояниями, не перебираются.

Покрытие состояний означает вложенность соответствующих множеств

состояний программы. Для предикатной абстракции покрытие проверяется как импликация, т.е. состояние φ_1 покрывает состояние φ_2, если $\varphi_2 \Rightarrow \varphi_1$. На рисунках покрытие будем обозначать прерывистой стрелкой помеченной словом *stop*.

Построение предикатной абстракции по заранее выбранным предикатам

Для начала рассмотрим построение абстракции по заранее выбранным предикатам, которые каким-то образом нам предоставили.

1. Выбираем предикаты:

   b1: $x > 0$,

   b2: $x = 0$,

   b3: $y > 0$,

   b4: $y = 0$,

   b5: $z \geq 0$.

2. Строим АГД с декартовой абстракцией.

   2.1. Обрабатываем L4:{T} $\xrightarrow{\text{x>y}}$ L5:{?}

   b1: $x_1 > y_1 \wedge \neg(x_1 > 0)$ – SAT ($x_1 = -1$, $y_1 = -2$)

   b2: $x_1 > y_1 \wedge \neg(x_1 = 0)$ – SAT ($x_1 = 2$, $y_1 = 1$)

   b3: $x_1 > y_1 \wedge \neg(y_1 > 0)$ – SAT ($x_1 = 1$, $y_1 = 0$)

   b4: $x_1 > y_1 \wedge \neg(y_1 = 0)$ – SAT ($x_1 = 2$, $y_1 = 1$)

   b5: $x_1 > y_1 \wedge \neg(z_1 \geq 0)$ – SAT ($x_1 = 2$, $y_1 = 1$, $z_1 = -1$)

   Результат L4:{T} $\xrightarrow{\text{x>y}}$ L5:{T}

   2.2. Аналогично L4:{T} $\xrightarrow{\text{!(x>y)}}$ L6:{T}

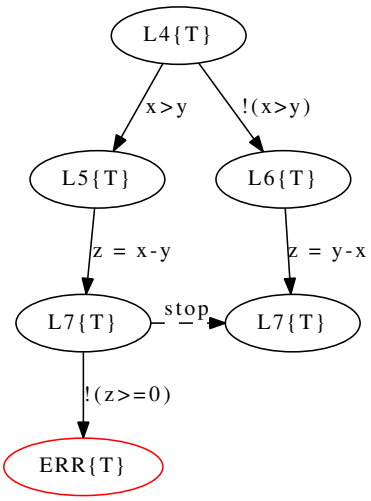

Рис. 4.3. Граф достижимости по заранее выбранным предикатам

2.3. Обрабатываем L5:{T} $\overset{z=x-y}{\longrightarrow}$ L7:{?}

Наши предикаты не содержат взаимосвязей между переменными x и y, поэтому мы не можем установить истинность предиката $z > 0$ после выполнения присваивания $z = x - y$.

$b1, \ldots, b4$ – SAT

b5: $z_2 = x_1 - y_1 \wedge \neg(z_2 \geq 0)$ – SAT ($x_1 = 1$, $y_1 = 2$, $z_2 = -1$)

Результат L5:{T} $\overset{z=x-y}{\longrightarrow}$ L7:{T}

2.4. Аналогично L7:{T} $\overset{z<0}{\longrightarrow}$ ERR:{T}

Достигаем ошибочное состояние.

Результат построения показан на Рис. 4.3.

Подобрать хорошие предикаты заранее затруднительно, так как часто они зависят от конкретной программы.

Использовать одни и те же предикаты на все случаи жизни неэффективно, так как зачастую каждой переменной требуется свой предикат, а остальные не влияют на точность абстракции. Собственно метод подбора предикатов описывается в следующем подразделе.

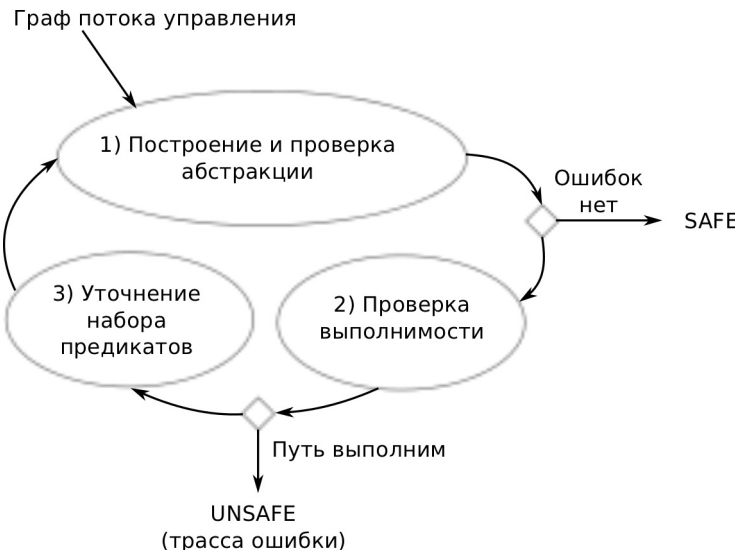

Рис. 4.4. Подход CEGAR (Counter Example Guided Abstraction Refinement)

Уточнение абстракции по контрпримерам

CEGAR[58] – метод статического анализа кода, основанный на абстракции и уточнении на основе контрпримеров.

Идея CEGAR состоит в итеративном уточнении абстракции до тех пор, пока не будет доказана ее корректность или не будет найден ошибочный путь, то есть путь, начинающийся в точке входа в программу и ведущий в ошибочное состояние. Для этого CEGAR предлагает строить такую абстракцию, в которой сохраняются пути исходной программы, и осуществлять поиск ошибочных путей в этой абстракции. Если в абстракции ошибочных путей не найдено, то в программе нет искомых ошибок.

Если в абстракции найден ошибочный путь, потребуется проверить его реализуемость в исходной программе, так как возможно, что ошибочный путь в абстракции появился из-за абстрагирования от существенных деталей в коде программы. Если проверка покажет, что путь реализуем, то это означает, что найдена ошибка в программе. Иначе этот путь в качестве контрпримера будет использован для уточнения абстракции на следующем шаге, так чтобы в уточненной абстракции этого пути уже не было.

В CEGAR выделяют следующие составляющие этапы (Рис. 4.4):

1. Построение и проверка абстракции.

2. Проверка выполнимости.

3. Уточнение набора предикатов.

На первом этапе строится начальная абстракция программы по начальному набору предикатов, например по пустому. Вторая часть первого этапа – проверка абстракции на наличие ошибки. Она может происходить как строго после построения абстракции, так и одновременно. Мы рассмотрим второй случай, в котором строится АГД.

В результате первого этапа либо удается доказать, что ошибок в абстрактной модели нет, либо находится ошибочный путь в абстракции, называемый контрпримером. Так как абстракция включает все ошибочные пути, то отсутствие ошибок в абстрактной модели говорит о том, что ошибки нет в исходной программе, инструмент выдает вердикт Safe. Данный вердикт может оказаться неправильным, т.е. может быть упущена ошибка, например, если программа содержит неподдерживаемые конструкции языка, которые моделируются неточно.

Если обнаруживается контрпример, то он передается на проверку его выполнимости в исходной программе. Для этого строится формула пути, кодирующая выполнение программы по пути к ошибке. Эта формула затем передается решателю. Семантика выполнения кодируется в терминах теорий, поддерживаемых решателями. Таким образом, точность проверки выполнимости зависит как от точности кодирования конструкций языка, так и от возможностей решателя. Неточности проверки выполнимости ведут к ложным срабатываниям, когда инструмент выдает вердикт Unsafe в случае отсутствия реальной ошибки.

Когда путь оказывается невыполнимым, это означает, что абстракция программы нуждается в уточнении. На этапе уточнения набора предикатов, невыполнимый путь используется для конструирования нового набора предикатов, такого, чтобы можно было исключить данный путь к ошибке в уточненной абстракции, а вместе с ним, возможно, и другие невыполнимые пути. Невыполнимый путь к ошибке используется для поиска новых предикатов как с помощью синтаксических методов, основанных на эвристиках, так и с помощью методов интерполяции.

Метод итерполяции состоит в том, чтобы использовать интерполянты Крейга (Craig)[61, 62], позволяющие находить неявные зависимости. Интерполянт существует всегда, когда формулы заданы в логике предикатов первого порядка. В статье [63] показано, что интерполянт может быть построен за линейное время по размеру резолютивного доказательства для формул в теориях равенства с неитерпретируемыми функциями и линейных неравенств.

Для двух формул φ и ψ, конъюнкция которых невыполнима ($\varphi \wedge \psi$ – UNSAT), интерполянтом является формула ρ такая, что

1. Содержит только общие символы из φ и ψ;
2. $\varphi \Rightarrow \rho$;
3. $\rho \wedge \psi$ – UNSAT.

Данные требования означают, что интерполянт – это общая часть двух формул, которая приближает формулу φ (условие 2), т.е. интерполянт может быть проще, но при этом он дает невыполнимость конъюнкции (условие 3). Таким образом, с помощью интерполянта можно показать невыполнимость формулы пути, разрезанной в какой то точке на две формулы, так как интерполянт использует только общие переменные доступные в этой точке (условие 1), должен выводиться по формуле, предшествующей точке разреза (2),

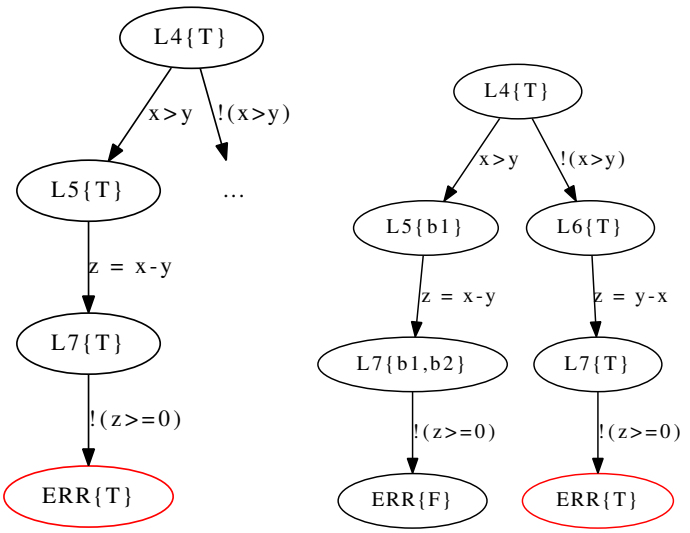

Рис. 4.5. Граф достижимости на начальной итерации(слева) и итерации 2(справа)

и отрезать путь после точки (3). Для получения таких интерполянтов существуют специальные *интерполирующие решатели*.

Рассмотрим применение метода CEGAR на примере 1.

1. Начальный набор предикатов пуст $\pi = \emptyset$, абстрактное состояние всегда {T}.

2. Метка ERR достижима, например по пути (Рис. 4.5)

   L4:{T} $\overset{\text{x>y}}{\longrightarrow}$ L5:{T}

   L5:{T} $\overset{\text{z=x−y}}{\longrightarrow}$ L7:{T}

   L7:{T} $\overset{\text{z<0}}{\longrightarrow}$ ERR:{T}

3. Проверяем данный путь на выполнимость. С учетом семантики выполнения Си программы конвертируем операторы в формулу пути.

   $x_1 > y_1 \wedge z_2 = x_1 - y_1 \wedge z_2 < 0$ – UNSAT

   Наша задача найти предикаты, которые не позволят пройти по данному пути.

4. Получаем новые предикаты на основе интерполянтов Крейга.

В нашем примере рассмотрим две точки разреза: L5 и L7

L5:

$\varphi_1 \equiv x_1 > y_1$,

$\psi_1 \equiv (z_2 = x_1 - y_1) \wedge (z_2 < 0)$

L7:

$\varphi_2 \equiv (x_1 > y_1) \wedge (z_2 = x_1 - y_1)$,

$\psi_2 \equiv z_2 < 0$

Интерполянты для φ_1, ψ_1:

$\rho_1 \equiv x_1 > y_1$

Интерполянты для φ_2, ψ_2:

$\rho_1 \equiv z_2 \geq 0$

5. Новые интерполянты добавляем в множество предикатов $\pi = \{b1, b2\}$

   b1: $x > y$,

   b2: $z \geq 0$.

6. Перестраиваем абстракцию (Рис. 4.5)

   6.1. Обрабатываем L4:{T} $\xrightarrow{\text{x>y}}$ L5:{?}

   b1: $x_1 > y_1 \wedge \neg(x_1 > y_1)$ – UNSAT

   b2: $x_1 > y_1 \wedge \neg(z_1 \geq 0)$ – SAT ($x_1 = 2$, $y_1 = 1$, $z_1 = -1$)

   Результат L4:{T} $\xrightarrow{\text{x>y}}$ L5:{b1}

   6.2. Обрабатываем L4:{T} $\xrightarrow{\text{!(x>y)}}$ L6:{?}

   b1: $\neg(x_1 > y_1) \wedge \neg(x_1 > y_1)$ – SAT ($x_1 = 1$, $y_1 = 2$)

   b2: $\neg(x_1 > y_1) \wedge \neg(z_1 \geq 0)$ – SAT ($x_1 = 1$, $y_1 = 2$, $z_1 = -1$)

   Результат L4:{T} $\xrightarrow{\text{!(x>y)}}$ L6:{T}

6.3. Обрабатываем L5:{b1} $\overset{\text{z=x−y}}{\longrightarrow}$ L7:{?}

Добавляем $x_1 > y_1$ в конъюнкцию как аппроксимацию текущего состояния $\{b1\}$

b1: $x_1 > y_1 \wedge z_2 = x_1 - y_1 \wedge \neg(x_1 > y_1)$ – UNSAT

b2: $x_1 > y_1 \wedge z_2 = x_1 - y_1 \wedge \neg(z_2 \geq 0)$ – UNSAT

Результат L5:{b1} $\overset{\text{z=x−y}}{\longrightarrow}$ L7:{b1,b2}

6.4. Обрабатываем L7:{b1,b2} $\overset{\text{z<0}}{\longrightarrow}$ ERR:{?}

$x_1 > y_1 \wedge z_2 \geq 0 \wedge z_2 < 0$ – UNSAT

состояние недостижимо, представляется как F (FALSE)

Результат L7:{b1,b2} $\overset{\text{z<0}}{\longrightarrow}$ ERR:{F}

Первый путь, на котором мы в первый раз нашли ошибку, оказался недостижимым. В силу того, что в L5 мы показали истинность b1, а в L7 – b2, что согласуется со свойством 2 интерполянта, свойство 3 интерполянта обеспечивает невыполнимость последнего перехода на пути.

6.5. L6:{T} $\overset{\text{z=y−x}}{\longrightarrow}$ L7:{T}

6.6. L7:{T} $\overset{\text{z<0}}{\longrightarrow}$ ERR:{T}

Результирующий граф достижимости показан на Рис. 4.5 справа.

7. Метка ERR: оказывается достижимой по другому пути.

L4:{T} $\overset{\text{!(x>y)}}{\longrightarrow}$ L6:{T}

L6:{T} $\overset{\text{z=y−x}}{\longrightarrow}$ L7:{T}

L7:{T} $\overset{\text{z<0}}{\longrightarrow}$ ERR:{T}

Составим формулу пути:

$\neg(x_1 > y_1) \wedge z_2 = y_1 - x_1 \wedge z_2 < 0$ – UNSAT

Рассмотрим две точки разреза: L6 и L7

L6:

$\varphi_1 \equiv \neg(x_1 > y_1)$,

$\psi_1 \equiv z_2 = y_1 - x_1 \wedge z_2 < 0$.

L7:

$\varphi_2 \equiv \neg(x_1 > y_1) \wedge z_2 = y_1 - x_1$,

$\psi_2 \equiv z_2 < 0$.

Интерполянты для φ_1, ψ_1:

$\rho_1 \equiv x_1 \leq y_1$

Интерполянты для φ_2, ψ_2:

$\rho_1 \equiv z_2 \geq 0$

8. Данные интерполянты добавляем в множество предикатов $\pi = \{b1, b2, b3\}$

   b1: $x > y$,

   b2: $z \geq 0$,

   b3: $x \leq y$.

9. Покажем, как будет пересчитана абстракция на пути к метке ERR.

   9.1. Обрабатываем L4:{T} $\xrightarrow{!(x>y)}$ L6:{?}

   b1: $\neg(x_1 > y_1) \wedge \neg(x_1 > y_1)$ – SAT ($x_1 = 1$, $y_1 = 2$)

   b2: $\neg(x_1 > y_1) \wedge \neg(z_1 \geq 0)$ – SAT ($x_1 = 1$, $y_1 = 2$, $z_1 = -1$)

   b3: $\neg(x_1 > y_1) \wedge \neg(x_1 \leq y_1)$ – UNSAT

   Результат L4:{T} $\xrightarrow{!(x>y)}$ L6:{b3}

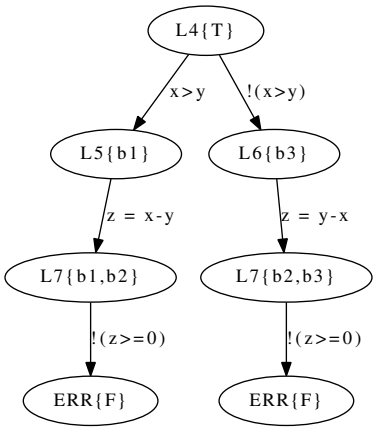

Рис. 4.6. Граф достижимости на итерации 3

9.2. Обрабатываем L6:{b3} $\overset{\text{z=y−x}}{\longrightarrow}$ L7:{?}

b1: $x_1 \leq y_1 \wedge z_2 = y_1 - x_1 \wedge \neg(x_1 > y_1)$ – SAT ($x_1 = 1$, $y_1 = 2$, $z_1 = -1$)

b2: $x_1 \leq y_1 \wedge z_2 = y_1 - x_1 \wedge \neg(z_2 \geq 0)$ – UNSAT

b3: $x_1 \leq y_1 \wedge z_2 = y_1 - x_1 \wedge \neg(x_1 \leq y_1)$ – UNSAT

Результат L6:{b3} $\overset{\text{z=y−x}}{\longrightarrow}$ L7:{b2,b3}

9.3. Обрабатываем L7:{b2,b3} $\overset{\text{z<0}}{\longrightarrow}$ ERR:{?}

$x_1 \leq y_1 \wedge z_2 \geq 0 \wedge z_2 < 0$ – UNSAT

Результат L7:{b2,b3} $\overset{\text{z<0}}{\longrightarrow}$ ERR:{F}

Результирующий граф достижимости показан на Рис. 4.6.

4.2.3. Оптимизация получения новых предикатов

В классическом алгоритме поиска предикатов, описанном выше, для получения интерполянта Крейга используется часть невыполнимой формулы пути до разреза и после разреза.

В BLAST 2.5 его авторами заложены два улучшения этого алгоритма. Во-первых, несколько операторов программы, не включающие операторы перехода и вызова функций объединяются в базовые блоки и не разделяются

на индивидуальные присваивания при выборе точек разреза (см. [57])

Во-вторых, вместо запуска процедуры интерполяции в каждой точке разреза, BLAST предварительно выделяет *полезные блоки* – подмножество операций в трассе, которые и являются своеобразной "причиной" невыполнимости формулы. То есть, вначале определяется минимальный набор базовых блоков, такой, что соответствующая ему конъюнкция подформул невыполнима, а остальная часть формулы пути – выполнима. Возможно выделение нескольких таких наборов. Особенностями этого приема (отличающие его от похожей концепции "ядра недостижимости", "ядра невыполнимости" или "unsatisfiable core"), являются:

- *гранулярность* выбора подформул. Разделение на подформулы происходит на уровне базовых блоков, которые соответствуют операторам исходной программы. При этом весь базовый блок (или условие) либо целиком входит в набор полезных блоков, либо целиком в этот набор не входит. В ядро недостижимости же могут входить и подформулы с меньшей гранулярностью;

- в *полезных блоках* могут участвовать и абстрактные состояния. По построению это некоторая аппроксимация формулы пути до точки, которой это состояние соответствует. Поэтому вместо анализа части трассы, предшествующей этому абстрактному состоянию, можно в *полезные блоки* добавлять само это состояние.

После выделения полезных блоков, BLAST запускает интерполирующий решатель для каждого разреза в каждом наборе *полезных блоков*, считая каждый такой набор как бы маленькой трассой ошибки. Таким образом, снижается нагрузка на интерполирующий инструмент, который по сравнению с SMT-решателем (без интерполяции) более подвержен ошибкам ввиду своей большей сложности.

Чтобы выделить полезные блоки, BLAST 2.5 по очереди соединяет соответствующие предикаты из формулы пути, пока их конъюнкция ни станет невыполнимой. Тогда последний добавленный блок считается полезным. Затем эта процедура повторяется сначала, только теперь выделенный полезный блок добавляется к каждой проверяемой конъюнкции, и таким образом определяется следующий полезный блок. Так происходит до тех пор, пока конъюнкция из всех найденных полезных блоков сама не станет невыполнимой. Время работы такого алгоритма $O(N)$, где N – количество базовых блоков в формуле пути.

Эта процедура естественным образом соответствует работе решателя, основанного на стеке. SMTLIB-решатели, однако, обрабатывают каждую из множества промежуточных формул отдельно. Было замечено, что выполнимость конъюнкции всех блоков от последнего до i-го – это монотонная функция от i (чем больше блоков присоединяется, тем скорее их конъюнкция окажется невыполнимой). Поэтому для получения очередного полезного блока в BLAST 2.7 была реализована процедура двоичного поиска соответствующего ему индекса i. Это существенного уменьшило нагрузки на SMT-решатели. Время работы алгоритма стало $O(log_2 N)$.

Также было реализовано кэширование частей преобразованной в SMTLIB-формат формулы. Как показали эксперименты на примере драйвера *cxausb*, количество вызовов решателя сократилось с 32630 до 831, давая семикратный прирост производительности.

В качестве интерполирующего инструмента BLAST использует CSIsat[64], распространяемый под лицензией GPL. Он использует входной формат, совместимый с другим интерполятором – FOCI[65]. CSIsat предназначен для интерполирования формул в рамках теории линейной вещественной арифметики и неинтерпретируемых функций (LA+EUF). Это, в частности, значит, что целочисленную линейную арифметику он

изначально не поддерживает, и, несмотря на то, что в нем реализованы некоторые эвристики для получения интерполянтов с целыми коэффициентами, все же иногда он может выдавать неравенства вроде $x < 0.1y$ (вместо эквивалентного $10x < y$). В этом случае BLAST игнорирует такие части полученного интерполянта и пытается доказать недостижимость ошибочной метки без соответствующих предикатов.

4.2.4. Анализ программ с указателями

Анализ программ с указателями, реализованный в инструменте BLAST, основан на анализе алиасов. В работе Андерсена [66] алиасом переменной называется одно из возможных обозначений этой переменной в данной точке программы. Например, пусть в программе имеются объявления

int a;

int *p;

Тогда в точке программы, где выполнено условие (p == &a), разыменование *p будет являться алиасом (иначе говоря, синонимом) переменной a. Аналогично можно говорить об алиасах некоторого адресного выражения (lvalue), т.е. выражения, которое может находиться в левой части оператора присваивания, семантически представляющее собой размещение переменной, массива, элемента структуры и т.п. Например, *(q + 1) может являться алиасом для b[1], если (q == b). В BLAST используется не зависящий от потока данных анализ возможных алиасов переменных для более точной верификации в присутствии присваиваний с участием указателей.

Анализ алиасов осуществляется в терминах анализа Андерсена [66]. Для хранения и запросов к информации об алиасах используется анализ BDD и алгоритм, похожий (но не совпадающий) с алгоритмом в [67]. Два символьных выражения считаются возможными алиасами друг друга, если существует

```
L1:    void main() {
L2:      int a,b;
L3:      int *p, *q, **r;
L4:      a = nondet_int();
L5:      b = nondet_int();
L6:      if(b) {
L7:        p = &a;
L8:        *p = 1;
L9:        assert(a==1);
         } else {
L10:       q = &a;
L11:       r = &q;
L12:       **r = 2;
L13:       assert(a=3);
L14:     }
       }
```

Рис. 4.7. Пример 2

значение на стеке (или точка вызова функции выделения памяти), которое является возможным алиасом для каждого из них.

Нечувствительный к контексту анализ алиасов Андерсена в примере 2 на Рис. 4.7 обнаружит, что *p является алиасом a, обозначим $*p \stackrel{\text{a}}{=} a$, так как в операторе, помеченном меткой L7, имеется присваивание p = &a. Аналогично *q является алиасом a, отсюда $*q \stackrel{\text{a}}{=} a \stackrel{\text{a}}{=} *p$. Также $*r \stackrel{\text{a}}{=} q$ и $**r \stackrel{\text{a}}{=} *q \stackrel{\text{a}}{=} a \stackrel{\text{a}}{=} *p$.

В [57] описано, как информация об алиасах используется для формул перехода и формулы пути, а в [62], в секции о программах без вызова функций, этот алгоритм расширен для поддержки структурных типов.

Анализ алиасов позволяет для данного присваивания определить, какие еще выражения (в каждый момент это подмножество всех возможных алиасов данного выражения) меняют свое значение в результате выполне-

ния данного оператора. Обозначим множество алиасов выражения t как $aliases(t) = \{x \mid t \stackrel{\text{a}}{=} x\}$. Обозначим несколько разыменований $\underbrace{*\cdots*}_{k} p$ как $*^k p$. Вспомогательная функция $EqAddr(*^{k_1} x_1, *^{k_2} x_2)$ сравнивает адреса размещений x_1 и x_2 и определяется как $EqAddr(*^{k_1} x_1, *^{k_2} x_2) \equiv *^{k_1} x_1 = *^{k_2} x_2$, если $k_1 = 0$ или $k_2 = 0$ и $EqAddr(*^{k_1} x_1, *^{k_2} x_2) \equiv EqAddr(*^{k_1-1} x_1, *^{k_2-1} x_2)$, иначе. Например $EqAddr(**q, **p) \equiv p = q$. Если в качестве одного из параметров передается переменная a, то вводится специальная переменная $addr\_a$, соответствующая выражению $\&a$, т.е. $EqAddr(*x, a) \equiv x = addr\_a$.

Для оператора присваивания $t = v$ выписываем:

$$SP_{t=v}(\varphi) \equiv \varphi \wedge (t_{new} = v) \bigwedge_{x \in aliases(t)} ite.\ EqAddr(x, t).\ (x_{new} = v).\ (x_{new} = x),$$

где $ite.\ b.\ p1.\ p2$ – это сокращенная запись оператора if b $then$ p1 $else$ p2 в виде логической формулы, x_{new} – это представление переменной с новыми SSA индексами.

Кроме того, в [62] для каждого присваивания осуществляется обновление размещений доступных по разыменованию. Например, если было присваивание $p = q$, то обновляются все выражения, доступные через разыменование p, т.е. $*p = *q$, $**p = **q$, и т.д. Данные присваивания будем записывать в виде функции:

$$Equate(p, q) \equiv \bigwedge_{0 \le k \le N} *^k p = *^k q,$$

где N – параметр формулы. Заметим, что число N может выбираться исходя из типов переменных, если предполагается строгая типизация, отсутствие преобразования типов и операций взятия адреса. В инструменте BLAST 2.5 число N задавалось как входной параметр $cldepth$.

В итоге формула для оператора присваивания имеет вид:

$$SP_{t=v}(\varphi) \equiv \varphi \wedge Equate(t_{new}, v) \wedge \qquad (1)$$

$$\bigwedge_{x \in aliases(t)} \begin{pmatrix} ite.EqAddr(x,t). \\ Equate(x_{new}, v). \\ Equate(x_{new}, x) \end{pmatrix}. \quad (2)$$

Например для оператора $*p = 1$, помеченного меткой L8, будет выписана формула:

$SP_{*p=1}(\varphi) \equiv \varphi \wedge$

выписываем условие присваивания

$\langle *p \rangle_2 = 1 \wedge$

дальше выписываем условия для всех алиасов

$aliases(*p) = \{a, *q, **r\}$

т.е. нужно выписать условия для трех алиасов

$ite.\ p_1 = addr\_a.\ a_2 = 1.\ a_2 = a_1 \wedge$

$ite.\ p_1 = q_1.\ \langle *q \rangle_2 = 1.\ \langle *q \rangle_2 = \langle *q \rangle_1 \wedge$

$ite.\ p_1 = \langle *r \rangle_1.\ \langle *r \rangle_2 = 1.\ \langle *r \rangle_2 = \langle *r \rangle_1 \wedge$

Здесь размещению, соответствующему разыменованию $*p$, присваивается новый индекс 2.

Если таким образом, например, расписать всю формулу пути по меткам $L6 - L9$,$L14$, то мы получим формулу, в которой встречаются обновления, генерируемые по равенству алиасов *q и **r, которые не влияют на выполнимость формулы.

В инструменте BLAST 2.7 формулу для оператора вычисления перехода добавляются только алиасы, которые влияют на выполнимость формулы.

Пусть дана формула пути $\varphi = \varphi_1 \wedge \varphi_2 \cdots \wedge \varphi_n$, где $\varphi_i = SP_{op_i}(\varphi_{i-1})$, $\varphi_0 = true$. Преобразуем данную формулу в $\hat{\varphi}$ следующим образом. Для любой формулы φ_i, если адресное выражение $x \in aliases(t)$ из части формулы (2) не встречается в части (1) формул $\varphi_{i+1}, \ldots, \varphi_n$, то удалим выражение (2) в формуле φ_i для x, т.е. уберем формулы для тех алиасов, которые не меняются

в хвосте формулы.

Справедливо следующее утверждение:

Утверждение 1. *Формула φ выполнима тогда и только тогда, когда выполнима формула $\hat{\varphi}$.*

Справедливость утверждения следует из того, что индексы переменных в формуле все время возрастают, так что обновление индекса делает ее недоступной для предыдущей части формулы. Если с какого-то момента переменная перестает использоваться, на нее не накладывается никаких дополнительных ограничений. Так что, если существовало решение для формулы $\hat{\varphi}$, то решение для φ получается прямым вычислением значений переменных по (2) в последующей части формулы. В обратную сторону утверждение верно, так как $\varphi \Rightarrow \hat{\varphi}$.

Аналогично производится упрощение генерации формул для вычисления предикатной абстракции. Достаточно заметить, что на выполнимость формулы влияют только те алиасы, которые встречаются в предикатах.

Данный новый алгоритм генерации позволяет существенно сократить нагрузку на решатели, так как в формуле существенно сокращается количество дизъюнкций. Кроме того, в новом алгоритме генерация формулы пути не зависит от параметра N, что позволяет гарантировать его корректность для произвольного N.

4.3. Известные ограничения

BLAST не учитывает возможности переполнения целых чисел, не поддерживает вызовы функций по указателю, игнорирует ассемблерные вставки, не осуществляет автоматического определения свойств сложных структур данных (таких как хэши и связные списки), не поддерживает массивы, считая

каждый доступ к массиву отдельной переменной (например, a[i] и a[j] считаются всегда разными переменными), не может использовать неравенство адресов структур на стеке, не всегда правильно обрабатывает короткую логику в выражениях, не разбит на легко заменяемые модули и всегда анализирует исходную программу целиком. Эти ограничения часто являются причиной ложных срабатываний инструмента. В таких случаях выданный им контрпример приходится анализировать вручную. Однако наш опыт применения BLAST для верификации кода драйверов Linux показал, что несмотря на перечисленные ограничения, этот инструмент вполне способен находить реальные ошибки (см. главу 7).

4.4. Выводы

В данной главе были рассмотрены следующие методы оптимизации, реализованные в инструменте BLAST 2.7:

- Настройка автоматического управления памятью в языке OCaml позволила уменьшить время, затрачиваемое на операции с памятью, что дало 20% прироста в количестве посещенных точек проверяемой программы на затраченную единицу памяти.

- В синтаксическом анализаторе CIL, используемым инструментом BLAST, были внесены исправления, которые позволили успешно анализировать до 98% драйверов ядра (измерено на версии 2.6.37), и лишь около 2% приводят к ошибкам разбора. До этого BLAST 2.5 не мог разобрать ни одного драйвера в ядре 2.6.31.

- Была улучшена работа с решателями, получающими на вход формулы в формате SMTLIB, за счет снижения накладных расходов при конвертации формул из внутреннего представления.

- Оптимизирован алгоритм фильтрации формулы пути, который осуществляет удаление частей формулы, не влияющих на получение интерполянта. Время работы алгоритма стало O(log N), вместо O(N).

- Был реализован новый алгоритм анализа алиасов, который позволяет анализировать программы, использующие указатели на базовые типы и структуры.

Данные улучшения позволили инструменту BLAST 2.7 занять первое место в категории DeviceDrivers64 и третье в категории DeviceDrivers32 на международных соревнованиях по верификации программ SV-COMP 2012 [68].

Глава 5

Система верификации драйверов ОС Linux

Система верификации *LDV* (Linux Driver Verification) является реализацией метода верификации, описанного в главе 2.

Разработанная система верификации обладает следующими важными особенностями:

- Система интегрирована с процессом сборки ядра, поэтому вся необходимая информация о составе и настройках сборки драйверов извлекается автоматически.
- Генерация окружения осуществляется полностью автоматически на основе иерархии шаблонов, покрывающей все типы драйверов.
- Система позволяет добавлять новые правила корректности с помощью аспектно-ориентированного расширения языка программирования Си.
- Система поддерживает встраивание внешних инструментов статической верификации с помощью написания адаптеров.
- Для анализа трасс ошибок и сравнительного анализа результатов система предоставляет специальные компоненты с Веб-интерфейсом.

5.1. Пользовательский интерфейс системы

Пользователь взаимодействует с системой верификации *LDV* посредством высокоуровневого интерфейса командной строки *LDV manager*. Данный компонент позволяет проверить некоторый набор драйверов (внутренних или внешних) для некоторого набора ядер по одному или нескольким

правилам корректности. В том случае, когда в ходе работы не происходит критическая исключительная ситуация, на выходе *LDV manager* создает архив, содержащий результаты анализа, информацию о работе компонентов архитектуры *LDV*, трассы ошибок и необходимые для их визуализации файлы с исходным кодом драйверов и ядра ОС Linux. Далее данный архив может быть загружен в базу данных и использован для анализа, например, с помощью *Statistics Server*.

Statistics Server – это компонент, который предоставляет веб-интерфейс для статистического анализа и сравнения результатов верификации.

Statistics Server позволяет анализировать большие объемы данных, получаемых в ходе верификации драйверов различных версий ядра ОС Linux по множеству правил с помощью инструментов статической верификации, запускаемых с разными конфигурациями. Помимо статистки, такой как, например, суммарное количество различных вердиктов для некоторого ядра и правила, компонент позволяет анализировать детальные списки, например, посмотреть все драйверы некоторого ядра, для которых инструмент верификации выдал вердикт Unsafe на некотором правиле корректности.

Система верификации *LDV* изначально проектировалась для использования различной целевой аудиторией такой, как разработчики компонентов, разработчики ядра, разработчики инструментов верификации достижимости и т.д. Как правило, запросы к представлению статистики у этих групп отличаются, поэтому *Statistics Server* предлагает различные заранее подготовленные профили представления данных. Так, например, разработчикам инструментов верификации достижимости помимо статистики по вердиктам предоставляется статистика по времени, затраченном на верификацию; разработчикам компонентов показывается статистика по внутренним проблемам соответствующих компонентов.

Еще одна важная возможность компонента – это возможность сравне-

ния результатов различных заданий верификации. В частности, это оказывается очень полезным и удобным инструментом для сравнения различных инструментов статической верификации, в том числе, различных версий и конфигураций.

Statistics Server интегрирован с *Error Trace Visualizer*, компонентом, который нацелен на упрощение анализа трасс ошибок, которые выдают инструменты верификации достижимости в случае вынесения вердикта Unsafe. Данный компонент позволяет ускорить анализ трасс ошибок, благодаря чему существенно повышается степень автоматизации процесса верификации в целом. *Error Trace Visualizer* подробно описан в статье [69].

Как правило, трасса ошибки представляется в текстовом виде, который имеет весьма специфичный, вообще говоря, сильно зависящий от инструмента верификации, формат. Для некоторых инструментов статической верификации существуют инструменты, позволяющие представить трассы ошибок в более наглядном и удобном для анализа формате. Например, трассу ошибки инструмента статической верификации CPAchecker можно преобразовать в HTML, после чего ее можно открывать в любом браузере. Трассы ошибок SATABS визуализируются посредством специального Eclipse плагина. А, например, инструмент статической верификации BLAST до *Error Trace Visualizer* не имел инструментов, позволяющих упростить анализ трасс ошибок. Подобное многообразие форматов представления трасс ошибок, в конечном итоге, затрудняет их анализ для различных инструментов верификации.

В рамках проекта *LDV* был разработан общий формат представления трасс ошибок. Разработанный формат является в достаточной степени гибким и расширяемым, что позволяет преобразовывать к нему трассы ошибок различных инструментах верификации без больших трудозатрат. Преобразование исходных трасс ошибок к общему формату реализуется на уровне адаптеров инструментов верификации. Для инструментов BLAST

и CPAchecker подобное преобразование было реализовано разработчиками *LDV. Error Trace Visualizer* визуализирует трассы, представленные в общем формате, единообразным образом и показывает результаты с помощью веб-интерфейса.

Важно отметить, что при визуализации наряду с трассой ошибок *Error Trace Visualizer* показывает соответствующий ей исходный код программы, причем, между ними устанавливаются определенные взаимосвязи (например, соответствие строк трассы ошибки строкам исходного кода программы). Также компонент выделяет каждый класс элементов трассы ошибки определенным стилем и цветом. Для показываемого исходного кода программы выполняется синтаксическая подсветка. Имеется возможность скрывать и раскрывать как отдельные элементы, так и целые классы элементов трассы ошибки. Все это существенно облегчает анализ трасс ошибок различных инструментов верификации.

5.2. Разработка адаптера инструмента верификации

Для использования инструмента статической верификации необходим адаптер, в котором можно выделить следующие четыре части:

- подготовка входных файлов;
- подготовка обработчика вывода инструмента верификации;
- запуск инструмента;
- обработка результатов.

Далее рассмотрены данные части по отдельности с указанием того, какие средства предоставляются разработчику адаптера для реализации соответствующих задач.

В рамках одной задачи инструменту верификации необходимо проверить достижимость ошибочной метки из некоторой точки входа, причем анализируемая программа может быть расположена в одном или нескольких файлах. Инструменты верификации могут накладывать ограничения на то, в каком виде должны быть представлены эти файлы. Разработчику адаптера предоставляется возможность применить к каждому из входных файлов стандартный препроцессор языка Си, обработать файлы с помощью инструмента трансформации кода CIL, или соединить все файлы в один с помощью того же CIL [70]. В результате возвращается список файлов, в которых содержится весь необходимый для проверки исходный код.

Обработчику вывода инструмента верификации на вход подаются строчки, выдаваемые инструментом верификации на стандартный вывод и/или на стандартный поток ошибок. Обработчик опционально возвращает набор значений с некоторой информацией, которую он извлек из трассы (например, последние 20 строк или вердикт о наличии/отсутствии в программе ошибок). Каждая функция-обработчик хранит свое внутреннее состояние. Непосредственно перед запуском инструмента верификации адаптер регистрирует такие функции, а *Reachibility C Verifier* применяет их, когда инструмент верификации будет запущен, параллельно с его работой.

Запуск инструмента верификации заключается в вызове библиотечной функции с командной строкой, соответствующей вызову инструмента верификации. Подготовка аргументов осуществляется в индивидуальной для каждого инструмента манере, на основе полученных имен препроцессированных файлов, точки входа и ошибочной метки. Библиотечная функция, через которую адаптер осуществляет вызов инструмента, отличается от стандартной функции вызова внешней программы некоторыми функциональными особенностями, а именно:

- автоматически применяет функции обработчики вывода, зарегистрированные адаптером ранее;
- вывод инструмента, который может быть большим по объему, архивируется и сохраняется на диск;
- инструмент запускается в контролируемом окружении, позволяя лимитировать использование инструментом и его дочерними процессами ресурсов машины, а именно потребления памяти, дискового пространства и процессорного времени;
- по желанию разработчика адаптера, происходит измерение потребления запускаемым инструментом и всеми его дочерними процессами времени; при этом разработчик адаптера может указать, измерение времени в каких именно процессах ему интересно и в какие категории каждый из них следует отнести.

После окончания запуска адаптер получает информацию о причине окончания (нарушение лимита ресурсов, получение сигнала или успешное завершение), коде возврата и номеру завершившего процесс сигнала, а также информацию, собранную функциями обработчиками вывода трассы. Ожидается, что в адаптере разработчик реализует автоматизированную интерпретацию этой информации. Например, разработчик адаптера должен реализовать интерпретацию трассы ошибки соответствующего инструмента статической верификации и ее преобразование в специальный общий формат, используемый для визуализации этой трассы.

Адаптер также может передать произвольную текстовую строку (например, содержащую исключение и трассу стека, выброшенные инструментом при неудачном завершении) и один или несколько файлов для сохранения их в финальном отчете с результатами верификации. Затем эта информация мо-

жет быть использована для последующей обработки и построения статистики с помощью компонента *LDV Statistics Server*.

Разработчик адаптера также может добавить конфигурируемость адаптеров, чтобы проводить эксперименты с различными настройками инструментов верификации без необходимости модифицировать код адаптера каждый раз. Например, для конфигурации разработчик адаптера может использовать переменные окружения.

Глава 6

Методика выявления и классификации правил корректности

6.1. Выбор источника

Для выявления правил корректности достаточно много информации можно найти в документации, в том числе, входящей в состав ядра и в исходный код ядра и драйверов, а также в литературе по разработке драйверов и ядра ОС Linux.

Однако, в этих источниках документированы далеко не все особенности различных подсистем ядра и типов драйверов. Поскольку ядро развивается очень стремительно, данные источники не всегда поддерживаются в актуальном состоянии.

В качестве еще одного источника для выявления правил можно рассмотреть список рассылки ядра Linux (от англ. Linux Kernel Mailing List, LKML). В данном списке рассылки обсуждаются различные актуальные вопросы разработки ядра, в том числе вопросы, связанные с исправлением ошибок. На основании этих обсуждений можно выявить множество общих и специфичных правил, но анализ сообщений в LKML достаточно трудоемок, поскольку сообщений много и они содержат большое количество информации, причем не только технической. Кроме того, много ошибок обсуждается в других списках рассылок, на форумах, системах отслеживания ошибок и т.д.

Правила корректности можно выявлять на основе анализа **журнала изменений**, содержащего изменения, вносимые в драйверы ОС Linux. Данный источник является достаточно актуальным, поскольку рано или поздно среди изменений появляются все важные исправления ошибок, которые об-

суждаются в различных списках рассылок, на конференциях, форумах и т.д. Изменения в драйверах ОС Linux проходят достаточно тщательную предварительную процедуру согласования и проверки, в том числе, у экспертов в соответствующих подсистемах ядра.

Каждое изменение содержит в себе достаточно подробную информацию, включающую сведения об авторе, краткое название изменения, подробное описание (возможно, со ссылками на соответствующие обсуждения), изменение исходного кода драйверов и/или ядра ОС Linux и различную вспомогательную информацию.

В качестве основного источника для выявления правил будем использовать журнал изменений.

6.2. Методика анализа журнала изменений

Первый шаг методики анализа изменения в драйвере ОС Linux заключается в том, чтобы определить, является ли оно исправлением ошибки или оно каким-либо образом расширяет функциональность драйвера (например, добавляет поддержку новых устройств).

Далее из рассмотрения исключаются ошибки, исправления которых связано с требованиями к функциональности конкретного драйвера или ошибки взаимодействия драйвера с конкретным оборудованием. Это так называемые "нетиповые" ошибки. В том случае, если ошибка связана с неправильным использованием специализированных для конкретного драйвера или группы драйверов констант, условий и вычислений, ее следует исключить из дальнейшего рассмотрения, так как для нее нельзя сформулировать правило корректности.

Основным шагом предлагаемой методики анализа изменений в драйверах ОС Linux является классификация ошибок, для которых можно сформу-

лировать правила.

Среди типовых ошибок можно выделить три класса. В первый класс входят общие ошибки в драйверах, как в программах на языке программирования Си (ядро и драйверы ОС Linux разрабатываются на языке программирования Си). Например, разыменовывание нулевого указателя, превышение максимально возможных значений переменных с целым типом и т.п. Ко второму классу относятся специфичные ошибки, связанные с неправильным использованием интерфейса сердцевины ядра. К числу таких ошибок относятся, например, нарушения правил инициализации переменных, имеющих специфичные типы. Третий класс типовых ошибок включает в себя состояния гонок и взаимные блокировки, которые возникают при параллельном выполнении вследствие неиспользования или неправильного использования механизмов синхронизации.

Данный шаг требует осмысления причины, которая привела к ошибке. Дело в том, что с первого взгляда может показаться, что ошибка проявилась вследствие нарушения некого общего правила. На самом деле, причина ошибки может крыться в особенностях параллельного выполнения либо в некорректном использовании интерфейса сердцевины ядра. Кроме того, не всегда легко определить, какую именно общую или специфичную ошибку исправляет рассматриваемое изменение. На данном шаге помимо анализа описания и изменения в исходном коде предполагается более тщательный анализ взаимодействий различных подсистем ядра и драйвера.

Важно отметить, что список классов неизвестен заранее и может быть определен только непосредственно по ходу самого анализа. Предполагается, что, в первую очередь, классы позволят различить общие и специфичные ошибки в драйверах.

```
commit 5fc8fe8e2ea30805bee5a13420817d6ad34ea9ce
Author: Anders Larsen <al@alarsen.net>
Date:   Wed Oct 6 23:46:25 2010 +0200

    USB: cp210x: Add WAGO 750-923 Service Cable device ID

    commit 93ad03d60b5b18897030038234aa2ebae8234748 upstream.

    The WAGO 750-923 USB Service Cable is used for configuration and firmware
    updates of several industrial automation products from WAGO Kontakttechnik GmbH.

    (skipped)

    Signed-off-by: Anders Larsen <al@alarsen.net>
    Signed-off-by: Greg Kroah-Hartman <gregkh@suse.de>

diff --git a/drivers/usb/serial/cp210x.c b/drivers/usb/serial/cp210x.c
index 3ad53bd..9927bca 100644
--- a/drivers/usb/serial/cp210x.c
+++ b/drivers/usb/serial/cp210x.c
@@ -132,6 +132,7 @@ static const struct usb_device_id id_table[] = {
        { USB_DEVICE(0x17F4, 0xAAAA) }, /* Wavesense Jazz blood glucose meter */
        { USB_DEVICE(0x1843, 0x0200) }, /* Vaisala USB Instrument Cable */
        { USB_DEVICE(0x18EF, 0xE00F) }, /* ELV USB-I2C-Interface */
+       { USB_DEVICE(0x1BE3, 0x07A6) }, /* WAGO 750-923 USB Service Cable */
        { USB_DEVICE(0x413C, 0x9500) }, /* DW700 GPS USB interface */
        { } /* Terminating Entry */
 };
```

Рис. 6.1. Описание изменения 5fc8fe8, добавляющего поддержку нового устройства

6.2.1. Примеры применения предложенной методики

Рассмотрим применение предложенной методики на примерах. На основании описания и изменения в исходном коде для изменения в драйвере, представленного на Рис. 6.1 можно сделать вывод, что данное изменение в драйвере не является исправлением ошибки, а добавляет поддержку нового USB устройства.

Изменение в драйвере на Рис. 6.2 демонстрирует исправление нетиповой ошибки, вызванной нарушением контракта драйвера трекпада. Изменение исходного кода задействует специфичные для данного драйвера константы и ограничения.

На Рис. 6.3 представлено изменение в драйвере, которое является исправлением типовой специфичной ошибки. Данная ошибка вызвана неправильным использованием интерфейса сердцевины ядра, а именно, в контексте блокировки вызывается функция выделения памяти, которая может переве-

```
commit 8e6b41c76c5b8a27b2abd7b9f6ed0877987fd11b
Author: Chase Douglas <chase.douglas@canonical.com>
Date:   Tue Jan 11 19:37:50 2011 +0100

    HID: magicmouse: Don't report REL_{X, Y} for Magic Trackpad

    [ Linus' tree commit 6a66bbd693c12f71697c61207aa18bc5a12da0ab ]

    With the recent switch to having the hid layer handle standard axis
    initialization, the Magic Trackpad now reports relative axes. This would
    be fine in the normal mode, but the driver puts the device in multitouch
    mode where no relative events are generated. Also, userspace software
    depends on accurate axis information for device type detection. Thus,
    ignoring the relative axes from the Magic Trackpad is best.

    Signed-off-by: Chase Douglas <chase.douglas@canonical.com>
    Signed-off-by: Jiri Kosina <jkosina@suse.cz>
    Signed-off-by: Greg Kroah-Hartman <gregkh@suse.de>

diff --git a/drivers/hid/hid-magicmouse.c b/drivers/hid/hid-magicmouse.c
index e6dc151..ed732b7 100644
--- a/drivers/hid/hid-magicmouse.c
+++ b/drivers/hid/hid-magicmouse.c
@@ -433,6 +433,11 @@ static int magicmouse_input_mapping(struct hid_device *hdev,
        if (!msc->input)
                msc->input = hi->input;

+       /* Magic Trackpad does not give relative data after switching to MT */
+       if (hi->input->id.product == USB_DEVICE_ID_APPLE_MAGICTRACKPAD &&
+           field->flags & HID_MAIN_ITEM_RELATIVE)
+               return -1;
+
        return 0;
```

Рис. 6.2. Описание изменения 8e6b41c, исправляющего нетиповую ошибку

```
commit 83a9a8034ee98ac21804c376ec90af8e4997790e
Author: John Johansen <john.johansen@canonical.com>
Date:   Wed Jun 8 15:07:47 2011 -0700

    AppArmor: Fix sleep in invalid context from task_setrlimit

    commit 1780f2d3839a0d3eb85ee014a708f9e2c8f8ba0e upstream.

    Affected kernels 2.6.36 - 3.0

    AppArmor may do a GFP_KERNEL memory allocation with task_lock(tsk->group_leader);
    held when called from security_task_setrlimit.  This will only occur when the
    task's current policy has been replaced, and the task's creds have not been
    updated before entering the LSM security_task_setrlimit() hook.

    BUG: sleeping function called from invalid context at mm/slub.c:847
     in_atomic(): 1, irqs_disabled(): 0, pid: 1583, name: cupsd

    (skipped)

    Signed-off-by: John Johansen <john.johansen@canonical.com>
    Reported-by: Miles Lane <miles.lane@gmail.com>
    Signed-off-by: James Morris <jmorris@namei.org>
    Signed-off-by: Greg Kroah-Hartman <gregkh@suse.de>

diff --git a/security/apparmor/lsm.c b/security/apparmor/lsm.c
index ec1bcec..3d2fd14 100644
--- a/security/apparmor/lsm.c
+++ b/security/apparmor/lsm.c
@@ -612,7 +612,7 @@ static int apparmor_setprocattr(struct task_struct *task, char *name,
 static int apparmor_task_setrlimit(struct task_struct *task,
                unsigned int resource, struct rlimit *new_rlim)
 {
-       struct aa_profile *profile = aa_current_profile();
+       struct aa_profile *profile = __aa_current_profile();
        int error = 0;

        if (!unconfined(profile))
```

Рис. 6.3. Описание изменения 83a9a80, исправляющего типовую специфичную ошибку (вызов функции, которая может заснуть, в контексте блокировки)

сти процесс в режим ожидания.

6.3. Классификация ошибок взаимодействия драйверов с ядром ОС Linux

Подробное описание анализа можно найти в статье [1].

В рамках основного анализа рассматривались изменения, которые были сделаны в стабильных ядрах с 2.6.35 по 3.0 в репозитории [71] за время, начиная с 26 октября 2010 года по 26 октября 2011 года. Всего таких изменений насчитывается 3101. Среди данных изменений некоторые являются дубликатами, поскольку одни и те же изменения попадают в различные стабильные ветки. Уникальных изменений за указанный период времени насчитывается 2623 (около 84.6% от общего числа изменений). Для того чтобы отделить изменения в драйверах от остальных изменений в ядре, рассматривались только такие изменения, которые затрагивали файлы из папок: crypto, drivers, sound, security, include/acpi, include/crypto, include/drm, include/media, include/mtd, include/pcmcia, include/rdma, include/rxrpc, include/scsi, include/sound и include/video. Из всех уникальных изменений, сделанных в стабильных ядрах с 26.10.10 по 26.10.11, в драйверах оказалось 1503, т.е. около 57.3% от общего числа уникальных изменений. Сводные результаты основного анализа изменений в драйверах представлены в Табл. 6.1. Подробное описание анализа можно найти в статье [1].

В соответствии с методикой, предложенной в данной работе, была произведена классификация выявленных типовых ошибок. При классификации не рассматривались изменения, которые являются исправлением типовых специфичных ошибок, связанных с некорректным использованием интерфейса группы драйверов (47 изменений). Изменениям, которые включали исправления сразу нескольких ошибок, приписывалось соответствующее количество

Изменения в драйверах (1503)		
Расширение функциональности (321 – 21%)	Исправление ошибок (1182 – 79%)	
	Нетиповые ошибки (786 – 67%)	Типовые ошибки (396 – 33%)

Таблица 6.1. Анализ изменений в драйверах стабильных версий ядра с 2.6.35 по 3.0 за время с 26.10.10 по 26.10.11

Класс		Количество	%
specific	Специфичные	176	50.4%
generic	Общие	102	29.2%
sync	Связанные с параллельным выполнением	71	20.4%
Всего		349	100%

Таблица 6.2. Распределение типовых ошибок в драйверах ОС Linux
(Всего 349 типовых ошибок за вычетом 47 типовых специфичных ошибок, связанных с некорректным использованием интерфейса группы драйверов)

классов. Подклассы ошибок первого уровня с небольшим количеством представителей (не более 3) были объединены в misc.

В Табл. 6.2 приводится общее распределение типовых ошибок по классам.

В Табл. 6.3 представлен список подклассов для специфичных ошибок, для которых можно сформулировать правило корректности взаимодействия драйвера с сердцевиной ядра. Таких ошибок было выделено 176, что составляет 14,9% от общего количества ошибок (1182) и 50.4% от общего количества типовых ошибок.

В разделе 7 показано, что метод позволяет обнаруживать нарушения для всех подклассов правил взаимодействия драйверов с ядром операционной системы из Табл. 6.3.

6.4. Аналогичные работы

Существуют подходы, нацеленные на автоматизированное извлечение правил на основе исходного кода программного обеспечения. Например, в статьях [72] и [73] предложены подходы для автоматического выявления неявных правил, которым должен удовлетворять исходный код анализируемых программ, на основе статистических данных. В отличие от [72], где правила получались только для шаблонов из пар элементов (например, если вызвана некоторая функция, то должна быть вызвана и другая), [73] описывает более общий подход, который затрагивает помимо функций переменные и типы и может выявлять правила для большего количества элементов без каких-либо шаблонов. Применительно к ядру ОС Linux эти подходы позволили выявить тысячи правил и десятки потенциальных ошибок. Однако, данные подходы не могут гарантировать, что найденные с их помощью ошибки не являются ложными срабатываниями. Они используют множество эвристик, в особенности, при различных статистических оценках и ранжировании выявленных правил. Поэтому требуется существенный ручной труд по анализу потенциальных ошибок. Кроме того, эти подходы не могут выявить все возможные правила и соответствующие им ошибки (в [72] задается несколько фиксированных шаблонов; ни в [72], ни в [73] не сообщается о классах и распределении найденных ошибок), что в принципе не позволяет получить полную классификацию и распределение ошибок даже после их ручного анализа.

В статьях [21] и [22] распределение типовых ошибок в ядре ОС Linux было получено автоматизированным образом с помощью инструментов ста-

тического анализа на основе предопределенной классификации. Благодаря данным исследованиям, в частности, было выявлено, что драйверы содержат до 85% всех ошибок, после чего много усилий было потрачено для повышения их надежности. Данным подходам, также как и подходам, описанным ранее, присуще наличие ложных срабатываний. По этой причине результаты, полученные для некоторых классов ошибок, вообще не принимались во внимание разработчиками ядра. Кроме того, подходы в значительной степени зависят от того, как инструменты статического анализа проверяют соответствующие классы. В [21] и [22] были использованы различные инструменты статического анализа. Очень вероятно, что именно это послужило причиной значительного расхождения полученных результатов.

Результаты, полученные в рамках данной работы, существенно отличаются от результатов этих исследований. Прежде всего, это связано с тем, что в предложенном подходе типовые ошибки анализируются вручную на основе изменений, вносимых в драйверы ОС Linux, в то время, как в аналогичных подходах ошибки определяются автоматизированным образом на основе анализа исходного кода ядра. Ручной подход позволяет проанализировать типовые ошибки в драйверах более тщательно. Поэтому на основании его результатов можно получать достаточно полную классификацию и более точное распределение ошибок.

№	Подкласс	Краткое описание подкласса	Количество	% от общего
1	resource	Утечки ресурсов и использование после освобождения	32	18.2%
2	check_params	Нарушение ограничений на входные параметры	25	14.5%
3	context	Вызовы функций в недопустимых контекстах	19	10.8%
4	uninit	Некорректная инициализация специфичных объектов	17	9.7%
5	lock	Некорректное использование механизмов синхронизации	12	6.8%
6	style	Нарушение стиля оформления драйверов ядра ОС Linux	10	5.7%
7	net	Некорректное использование интерфейса сетевой подсистемы	10	5.7%
8	usb	Некорректное использование интерфейса USB подсистемы	9	5.1%
9	check_ret_val	Отсутствие проверок возвращаемых значений	7	4.0%
10	dma	Некорректное использование интерфейса подсистемы прямого доступа к памяти	4	2.3%
11	device	Некорректное использование интерфейса общей модели драйвера	4	2.3%
12	misc	Разное	27	15.3%
Всего			176	100%

Таблица 6.3. Описание подклассов специфичных ошибок взаимодействия в драйверах ОС Linux

Глава 7

Практические результаты

Система верификации выносит один из трех вердиктов: Safe, Unsafe и Unknown. Вердикт Safe означает отсутствие нарушений проверяемого правила корректности. Вердикт Unsafe говорит об обнаружении нарушения правила и сопровождается трассой ошибки, которая показывает путь выполнения программы, приводящий к ошибочному состоянию. Вердикт Unknown означает, что система по тем или иным причинам (например, нехватка памяти или времени) не смогла найти ответ на поставленный вопрос.

Следует отметить, что количество истинных вердиктов Unsafe по мере того, как разработчики их исправляют, в каждой версии неизменно снижается в результате остаются только неисправленные истинные вердикты Unsafe, например, ошибки в неподдерживаемых драйверах или ошибки, для которых на данный момент не имеется приемлемого исправления.

Система верификации предоставляет возможность сохранять результаты анализа вердиктов Unsafe в базе знаний. База знаний позволяет автоматически отображать количество истинных вердиктов Unsafe (True Unsafe), ложных вердиктов Unsafe (False Unsafe) и непроанализированных или не до конца изученных вердиктов Unsafe (Unknown Unsafe).

При верификации новых версий ложные и истинные вердикты Unsafe, проанализированные в предыдущих версиях и сохраненные в базе знаний, автоматически отображаются в новой версии. Совпадение трасс ошибок определяется с точностью до заданного критерия (трасса ошибки целиком, дерево вызовов, стек вызовов).

В качестве иллюстрации приводится фрагмент из отчета по результатам верификации драйверов ядра ОС Linux linux-3.4 в мае 2012 года на моделях

Rule	Total	Safe	Unsafe	Unknown	Verdicts		
					True	False	?
32_7	3441	2998	44	399	4	40	-
39_7	3441	3002	40	399	11	29	-
43_1a	3441	2965	10	466	2	7	1

Рис. 7.1. Результаты верификации драйверов ядра ОС Linux версии 3.4

правил 32_7, 39_7, 43_1a, при ограничении на память 6Gb, ограничении на анализ одного драйвера 15 мин (Рис. 7.1).

В данном запуске системы верификации было проверено 3441 модуля драйверов. Количество положительных вердиктов Safe или Unsafe от 86,5% до 88,4% (в среднем 87,5%). В среднем получено от 11,6% до 13,5% вердиктов Unknown (12,5% в среднем). Количество ложных вердиктов Unsafe относительно количества проанализированных драйверов от 0,2% до 1,1%, что составляет (в среднем 0,7%). Для примера, по сравнению с версией 3.2 ядра ОС Linux, в ядре 3.4 для правила 32_7 появилось лишь 3 новых вердикта Unsafe.

Таким образом, количество ложных вердиктов Unsafe невелико – при анализе очередных новых версий ядра появляются лишь единицы новых ложных Unsafe вердиктов.

Всего по предложенному в работе методу разработано более 40 моделей правил, которые в том числе покрывают все подклассы специфичных ошибок из Табл. 6.3. Хотя в рамках данной работы основные усилия были направлены на разработку системы верификации, было обнаружено более 75 ошибок в драйверах Linux, которые были признаны и исправлены в основной ветке ядра.

Заключение

Основные научные и практические результаты, полученные в диссертационной работе и выносимые на защиту, состоят в следующем:

1. Разработан метод верификации драйверов устройств операционных систем для проверки выполнения правил корректного взаимодействия драйверов с ядром операционной системы;

2. Разработан метод построения моделей окружения драйверов устройств ОС Linux;

3. Разработан метод построения конфигурируемой системы верификации, обеспечивающий возможность расширения системы за счет пополнения набора правил корректности и набора инструментов верификации;

4. Разработаны методы оптимизации предикатной абстракции в инструменте BLAST;

5. На основе предложенных методов разработана система верификации драйверов ОС Linux.

Система верификации драйверов ОС Linux разработана в рамках проектов отдела Технологий программирования Института системного программирования РАН при непосредственном участии автора в качестве руководителя и участника разработки основных компонентов системы верификации. Автор выражает свою признательность всем участникам данных проектов. Особый вклад в работу внесли А.В.Хорошилов, Е.М.Новиков, П.Е.Швед.

Литература

[1] Мутилин В. С., Новиков Е. М., Хорошилов А. В. Анализ типовых ошибок в драйверах операционной системы Linux // Труды Института системного программирования РАН. 2012. Т. 22. С. 349–374.

[2] Мандрыкин М. У., Мутилин В. С., Новиков Е. М., Хорошилов А. В. Обзор инструментов статической верификации Си программ в применении к драйверам устройств операционной системы Linux // Труды Института системного программирования РАН. 2012. Т. 22. С. 293–326.

[3] Мутилин В. С., Мандрыкин М. У. Интерполяция формул с кванторами в CSIsat на основе инстанцирования // Труды Института системного программирования РАН. 2012. Т. 22. С. 327–348.

[4] Мандрыкин М. У., Мутилин В. С., Новиков Е. М. и др. Использование драйверов устройств операционной системы Linux для сравнения инструментов статической верификации // Программирование. 2012. Т. 5. С. 54–71.

[5] Швед П. Е., Мутилин В. С., Мандрыкин М. У. Опыт развития инструмента статической верификации BLAST // Программирование. 2012. Т. 3. С. 24–35.

[6] Mutilin V., Mandrykin M. Instantiation-Based Interpolation for Quantified Formulae in CSIsat // Proceedings of SYRCoSE. 2012. Vol. 1. P. 85–93.

[7] Shved P., Mandrykin M., Mutilin V. Predicate Analysis with Blast 2.7 // Proceedings of TACAS. 2012. Vol. 7214. P. 525–527.

[8] Мутилин В. С., Новиков Е. М., Страх А. В. и др. Архитектура Linux

Driver Verification // Труды Института системного программирования РАН. 2011. Т. 20. С. 163–187.

[9] Shved P., Mutilin V., Mandrykin M. Static Verfication “Under The Hood”: Implementation Details and Improvements of BLAST // Proceedings of SYRCoSE. Vol. 1. 2011. P. 54–60.

[10] Khoroshilov A., Mutilin V., Novikov E. et al. Towards An Open Framework for C Verification Tools Benchmarking // Proceedings of PSI. 2011. P. 82–91.

[11] Khoroshilov A., Mutilin V., Petrenko A., Zakharov V. Establishing Linux Driver Verification Process // Perspectives of Systems Informatics / Ed. by A. Pnueli, I. Virbitskaite, A. Voronkov. Springer Berlin / Heidelberg, 2010. Vol. 5947 of Lecture Notes in Computer Science. P. 165–176.

[12] Мутилин В., Хорошилов А. База правил для верификации драйверов Linux // Тезисы докладов VI Конференции разработчиков свободных программ на Протве. 2009.

[13] Мутилин В. С., Хорошилов А. В., Захаров В. А. Верификация безопасности драйверов ОС Linux // Материалы XVII Общероссийской научно-технической конференции “Методы и технические средства обеспечения безопасности информации”. 2008. С. 106–112.

[14] Khoroshilov A., Mutilin V., Shcherbina V. et al. How to Cook an Automated System for Linux Driver Verification // 2nd Spring Young Researchers’ Colloquium on Software Engineering. Vol. 2 of SYRCoSE 2008. 2008. P. 11–14.

[15] Khoroshilov A., Mutilin V. Formal Methods for Open Source Components Certification // 2nd International Workshop on Foundations and Techniques for Open Source Software Certification. OpenCert 2008. 2008. P. 52–63.

[16] Kroah-Hartman G., Corbet J., McPherson A. Linux Kernel Development: How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It [Электронный ресурс] // Режим доступа: http://go.linuxfoundation.org/who-writes-linux-2012, свободный. 2012.

[17] Ядро дистрибутива Debian [Электронный ресурс] // Режим доступа: http://wiki.debian.org/DebianKernel, свободный.

[18] ОС Linux реального времени [Электронный ресурс] // Режим доступа: https://www.osadl.org/Realtime-Linux.projects-realtime-linux.0.html, свободный.

[19] Corbet J. How to Participate in the Linux Community. A Guide To The Kernel Development Process [Электронный ресурс] // Режим доступа: http://www.linuxfoundation.org/sites/main/files/How-Participate-Linux-Community_0.pdf, свободный. 2008.

[20] Leemhuis T. What's new in Linux 3.3 [Электронный ресурс] // Режим доступа: http://www.h-online.com/open/features/What-s-new-in-Linux-3-3-1466872.html, свободный. 2012.

[21] Chou A., Yang J., Chelf B. et al. An empirical study of operating systems errors // SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles. New York, NY, USA: ACM, 2001. P. 73–88.

[22] Swift M. M., Bershad B. N., Levy H. M. Improving the reliability of commodity operating systems // SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles. New York, NY, USA: ACM, 2003. P. 207–222.

[23] Ganapathi A., Ganapathi V., Patterson D. Windows XP kernel crash analysis // Proceedings of the 20th conference on Large Installation System Ad-

ministration. LISA '06. Berkeley, CA, USA: USENIX Association, 2006. P. 12–12.

[24] Palix N., Thomas G., Saha S. et al. Faults in linux: ten years later // SIGPLAN Not. 2011. Vol. 47, no. 4. P. 305–318.

[25] ISO/IEC TR 24772 // Information Technology – Programming Languages – Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use. 2010.

[26] Kroah-Hartman G. The Linux Kernel Driver Interface [Электронный ресурс] // Режим доступа: `http://www.kernel.org/doc/Documentation/stable_api_nonsense.txt`, свободный.

[27] Ахо А. В., Лам М. С., Сети Р., Ульман Д. Д. Компиляторы: принципы, технологии и инструментарий. 2-е изд. Москва: Вильямс, 2008.

[28] Johnson S. C. Lint, a C program verifier // Bell Laboratories. Murray Hill, New Jersey, 1987.

[29] Страница инструмента Sparse – A Semantic Parser for C [Электронный ресурс] // Режим доступа: `http://www.kernel.org/pub/software/devel/sparse/`, свободный.

[30] Obdržálek J., Slabý J., Trtík M. STANSE: bug-finding framework for C programs // Proceedings of the 7th international conference on Mathematical and Engineering Methods in Computer Science. MEMICS'11. Berlin, Heidelberg: Springer-Verlag, 2012. P. 167–178.

[31] Engler D., Chelf B., Chou A. Checking system rules using system-specific, programmer-written compiler extensions // Proceedings of the 4th conference

on Symposium on Operating System Design & Implementation - Volume 4. 2000. P. 1–16.

[32] Guo P. J., Engler D. Linux kernel developer responses to static analysis bug reports // Proceedings of the 2009 conference on USENIX Annual technical conference. USENIX'09. Berkeley, CA, USA: USENIX Association, 2009. P. 22–22.

[33] Padioleau Y., Hansen R. R., Lawall J. L., Muller G. Semantic patches for documenting and automating collateral evolutions in Linux device drivers // Proceedings of the 3rd workshop on Programming languages and operating systems: linguistic support for modern operating systems. PLOS '06. New York, NY, USA: ACM, 2006.

[34] Stuart H. Hunting bugs with Coccinelle: Ph. D. thesis / University of Copenhagen. 2008.

[35] Lawall J. L., Brunel J., Palix N. et al. WYSIWIB: A declarative approach to finding API protocols and bugs in Linux code // DSN'09 – The 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. 2009. P. 43–52.

[36] Xie Y., Aiken A. Saturn: a SAT-based tool for bug detection // Proceedings of the 17th international conference on Computer Aided Verification. CAV'05. Berlin, Heidelberg: Springer-Verlag, 2005. P. 139–143.

[37] Xie Y., Aiken A. Saturn: A scalable framework for error detection using Boolean satisfiability // ACM Trans. Program. Lang. Syst. 2007. Vol. 29, no. 3.

[38] Dillig I., Dillig T., Aiken A. Sound, complete and scalable path-sensitive analysis // SIGPLAN Not. 2008. — June. Vol. 43. P. 270–280.

[39] Pratikakis P., Foster J. S., Hicks M. LOCKSMITH: Practical static race detection for C // ACM Trans. Program. Lang. Syst. 2011. Vol. 33, no. 1. P. 3:1–3:55.

[40] Сыромятников С. Декларативный интерфейс поиска дефектов по синтаксическим деревьям: язык KAST // Труды Института системного программирования РАН. 2011. Т. 20. С. 51–68.

[41] Ayewah N., Hovemeyer D., Morgenthaler J. D. et al. Using Static Analysis to Find Bugs // IEEE Softw. 2008. Vol. 25, no. 5. P. 22–29.

[42] Evans D., Larochelle D. Improving Security Using Extensible Lightweight Static Analysis // IEEE Softw. 2002. — January. Vol. 19. P. 42–51.

[43] Аветисян А., Белеванцев А., Бородин А., Несов В. Использование статического анализа для поиска уязвимостей и критических ошибок в исходном коде программ // Труды Института системного программирования РАН. 2011. Т. 21. С. 23–38.

[44] Ball T., Bounimova E., Levin V. et al. The Static Driver Verifier Research Platform // Computer Aided Verification. CAV'10. 2010. P. 119–122.

[45] Ball T., Rajamani S. K. SLIC: A specification language for interface checking [Электронный ресурс] // Режим доступа: `http://research.microsoft.com/apps/pubs/default.aspx?id=69906`, свободный. 2001.

[46] Ball T., Bounimova E., Kumar R., Levin V. SLAM2: Static driver verification with under 4% false alarms // Formal Methods in Computer-Aided Design (FMCAD), 2010. 2010. — oct. P. 35 –42.

[47] Ball T., Levin V., Rajamani S. K. A decade of software model checking with SLAM // Commun. ACM. 2011. Vol. 54, no. 7. P. 68–76.

[48] Nori A. V., Rajamani S. K., Tetali S., Thakur A. V. The Yogi Project: Software Property Checking via Static Analysis and Testing // Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,. TACAS '09. Berlin, Heidelberg: Springer-Verlag, 2009. P. 178–181.

[49] Post H., Küchlin W. Integrated static analysis for Linux device driver verification // Proceedings of the 6th international conference on Integrated formal methods. IFM'07. Berlin, Heidelberg: Springer-Verlag, 2007. P. 518–537.

[50] Witkowski T., Blanc N., Kroening D., Weissenbacher G. Model checking concurrent Linux device drivers // ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. New York, NY, USA: ACM, 2007. P. 501–504.

[51] Clarke E., Kroening D., Lerda F. A Tool for Checking ANSI-C Programs // Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004) / Ed. by K. Jensen, A. Podelski. Vol. 2988 of Lecture Notes in Computer Science. Springer, 2004. P. 168–176.

[52] Clarke E., Kroening D., Sharygina N., Yorav K. SATABS: SAT-based Predicate Abstraction for ANSI-C // Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005). Vol. 3440 of Lecture Notes in Computer Science. Springer Verlag, 2005. P. 570–574.

[53] Post H., Küchlin W. Automatic data environment construction for static device drivers analysis // Proceedings of the 2006 conference on Specification and verification of component-based systems. SAVCBS '06. New York, NY, USA: ACM, 2006. P. 89–92.

[54] Beyer D., Henzinger T. A., Jhala R., Majumdar R. The software model checker Blast: Applications to software engineering // Int. J. Softw. Tools Technol. Transf. 2007. Vol. 9, no. 5. P. 505–525.

[55] Novikov E. One Approach to Aspect-Oriented Programming Implementation for the C programming language // Proceedings of SYRCoSE. 2011. Vol. 1. P. 74–81.

[56] Milner R. The Polyadic π-Calculus: a Tutorial. LFCS, Department of Computer Science, University of Edinburgh, 1991. P. 49.

[57] Henzinger T. A., Jhala R., Majumdar R. Lazy abstraction // Symposium on Principles of Programming Languages. ACM Press, 2002. P. 58–70.

[58] Clarke E., Grumberg O., Jha S. et al. Counterexample-Guided Abstraction Refinement // Computer Aided Verification / Ed. by E. Emerson, A. Sistla. Springer Berlin / Heidelberg, 2000. Vol. 1855 of Lecture Notes in Computer Science. P. 154–169.

[59] Detlefs D., Nelson G., Saxe J. B. Simplify: a theorem prover for program checking // J. ACM. 2005. Vol. 52, no. 3. P. 365–473.

[60] Barrett C., Tinelli C. CVC3 // Proceedings of the 19^{th} International Conference on Computer Aided Verification (CAV '07) / Ed. by W. Damm, H. Hermanns. Vol. 4590 of Lecture Notes in Computer Science. Springer-Verlag, 2007. P. 298–302. Berlin, Germany.

[61] Craig W. Linear reasoning // J. Symbolic Logic. 1957. Vol. 22. P. 250–268.

[62] Henzinger T. A., Jhala R., Majumdar R., McMillan K. L. Abstractions from proofs // SIGPLAN Not. 2004. Vol. 39, no. 1. P. 232–244.

[63] Jhala R., Majumdar R. Path slicing // SIGPLAN Not. 2005. Vol. 40, no. 6. P. 38–47.

[64] Beyer D., Zufferey D., Majumdar R. CSIsat: Interpolation for LA+EUF // CAV. 2008. P. 304–308.

[65] McMillan K. L. An interpolating theorem prover // Theor. Comput. Sci. 2005. Vol. 345, no. 1. P. 101–121.

[66] Andersen L. O. Program Analysis and Specialization for the C Programming Language. Københavns Universitet. Datalogisk Institut. DIKU, 1994.

[67] Berndl M., Lhoták O., Qian F. et al. Points-to analysis using BDDs // SIGPLAN Not. 2003. Vol. 38, no. 5. P. 103–114.

[68] 1st International Competition on Software Verification (SV-COMP) held at TACAS 2012 [Электронный ресурс] // Режим доступа: `http://sv-comp.sosy-lab.org`, свободный.

[69] Новиков Е. Упрощение анализа трасс ошибок инструментов статического анализа кода // Труды второй научно-практической конференции «Актуальные проблемы системной и программной инженерии» (АПСПИ-2011). 2011. — 25 Мая. С. 215–221.

[70] Necula G. C., McPeak S., Rahul S. P., Weimer W. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs // Proceedings of the 11th International Conference on Compiler Construction. CC '02. London, UK: Springer-Verlag, 2002. P. 213–228.

[71] Репозиторий стабильных версий ядра ОС Linux [Электронный ресурс] // Режим доступа: `http://git.kernel.org/?p=linux/kernel/git/stable/linux-stable.git;a=summary`, свободный.

[72] Engler D., Chen D. Y., Hallem S. et al. Bugs as deviant behavior: a general approach to inferring errors in systems code // SIGOPS Oper. Syst. Rev. 2001. Vol. 35, no. 5. P. 57–72.

[73] Li Z., Zhou Y. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code // SIGSOFT Softw. Eng. Notes. 2005. Vol. 30, no. 5. P. 306–315.

Приложение А

Приложения к методу генерации окружения целевого драйвера

А.1. Трансляция процессов в Си программу

1. Имеем два списка процессов P_i на верхнем уровне:
   1.1. Статические процессы $T_1(const_i), \ldots, T_M(const_M)$;
   1.2. Процессы с динамически порождаемыми копиями $!create_1(x_1).C_1(x_1), \ldots, !create_L(x_L).C_L(x_L)$.
2. Шаблон генерируемого кода:
   2.1. Декларации типов.

%FOREACH P_i

%FOREACH K_j^i – состояния процессов нижнего уровня

//идентификаторы состояний

enum P_i_states $\{STOP, K_1^i, \ldots, K_n^i\}$;

struct P_i_state {

//поле list для добавления в список состояний

struct list list;

//переменные процесса P_i

enum P_i_states state;

...

}

   2.2. Список состояний процессов.

%FOREACH P_i

struct list P_i_state_list;

2.3. Функции приема сигналов.

2.3.1. Основные функции приема.

```
//Для всех меток приема сигнала
%FOREACH f[p](y) ∈ F
int f(int p, type y) {
  switch(nondet_int()) {
  %FOREACH P_i
  case P_i_label:
    int k = list_size(P_i_state_list);
    if(k==0) {return -1;}
    int i = nondet_int(k-1);
    struct P_i_state *s = list_get(P_i_state_list, i);
    int r = P_i_f(p, s, y);
    if(r==0) return 0;
    else return -1;
  }
}
```

2.3.2. Функции приема сигналов создания копий.

```
%FOREACH C_i
int create_i(type y) {
  //Если указано (νp), то добавляем
  int p=globalId++;//иначе p=-1
  create_C_i(p, y)
  return 0;
}
```

2.3.3. Вспомогательные функции приема сигналов.

%FOREACH P_i

%FOREACH $f[p](y) \in F$, кроме создания копий

//обработка приема сообщений

int P_i_f(int p, struct P_i_state *s, type y) {

int res = -1;

//КОД ПРИЕМА СИГНАЛА 1 . . .

//установить res в 0 если сигнал обработан

return res;

}

Для меток сигналов без параметров генерируем аналогичный код, но без параметров *int p*.

int a(type y)

int $create_i$(type y)

int P_i_a(struct P_i_state *s, type y)

2.4. Функции инициализации начального состояния.

P_i_init(int p, struct P_i_state *s, type y) {

ИНИЦИАЛИЗАЦИЯ;

. . .

}

2.5. Один шаг процесса (возвращает 1 – если переходим в конечное состояние, 0 – иначе).

%FOREACH P_i

int P_i_step(struct P_i_state *s) {

switch(nondet_int()) {

//ПОСЫЛКА СИГНАЛА 1 . . .

```
        case 1:
        КОД ПОСЫЛКИ СИГНАЛА 1
        break;
        ...
    }
    return 0;
}
```

2.6. Создание потока процесса.

```
//Основной цикл процесса
%FOREACH P_i
void P_i_thread(struct P_i_state *s) {
    while(true) {
        lock();
        int should_stop = P_i_step(s);
        if(should_stop) {
        list_del(P_i_state_list,s);
        free(s);
            unlock();
            break;
        };
        unlock();
    }
}
//Создание
%FOREACH P_i
```

void create_P_i(int p, type y) {

struct P_i_state *s = malloc(sizeof(struct P_i_state));

P_i_init(p, s, y);

list_add(&P_i_state_list, &s->list)

pthread_create(P_i_thread, s);

}

2.7. Точка входа.

void main() {

%FOREACH T_i

//Если есть $(\nu p)T_i$, то

int p = globalId++;//иначе p=-1

//Если задан параметр константа $T_i(const_i)$, то

type x = $const_i$;//иначе x=-1

create_T_i(p, x);//Создать поток

}

3. Создание параметра динамически связываемой метки $(\nu p)P_i$.

Добавляем в состояние struct P_i_state:

int p;

Добавляем в инициализацию P_i_init:

s->p = p;

Для копируемого процесса выбор значения p осуществляется в $create_i$ (п. 2.3), для статического в $main$ (п. 2.7).

4. Инициализация состояния.

4.1. Статические процессы $P_i ::= T_i(const_i)$.

Добавляем в состояние struct T_i_state:

type T_i_param;

Добавляем в инициализацию T_i_init:

s->state = T_i;

s->T_i_param = y;

В функции *main* добавляем задание значения (п. 2.7).

4.2. Процессы с динамически порождаемыми копиями
$P_i ::= !create_i(x_i).C_i(x_i)$.

Добавляем в состояние struct C_i_state:

type C_i_param;

Добавляем в инициализацию C_i_init:

s->state = C_i;

s->param = y;

5. Анализируем процессы верхнего уровня P, для каждого процесса разбираем процессы нижнего уровня K.

5.1. Простой процесс нижнего уровня K без параметров задает состояние с идентификатором K (см. enum P_i_states п.2.1).

Для переходов в такое состояние, генерируем код:

s->state = K;

5.2. Процесс с параметром $K(x)$ задает состояние с идентификатором K.

Добавляем переменную в структуру состояния:

type K_param;

Для переходов в такое состояние $K(value)$, генерируем код:

s->state = K;

s->K_param = value;

5.3. Пустой процесс.

$K_1 ::= 0$;

Отождествляет все переходы в состояние K_1 с переходом в конечное состояние $STOP$.

В функции P_i_step п.2.5 генерируем код:

case νlabel:

if(s->state==$STOP$) return 1;

break;

5.4. Недетерминированный выбор.

$K(x) ::= \sum_{i=1}^{n} \alpha_i(x_i).K_i(e_i(x, x_i))$

Добавляем код в зависимости от действия α_i.

5.4.1. Действие посылка $\alpha_i = \bar{f} \in \bar{F}$.

$\overline{f[p_i](e_i'(x)\rangle}.K_i(e_i(x))$

В функции P_i_step п.2.5 добавляем код:

case νlabel:

if(s->state == K) {

//Вызов функции действия с заданным параметром

// и проверка успешности выполнения

int res = f(s->p_i, e_i'(s->K_param));

if(res==0) {

//Переход в состояние K_i

s->state = K_i;

s->K_i_param = e_i(s->K_param);

} else {

//остаемся в текущем состоянии,

//необходимо перевыполнить посылку.

}

}

break;

При наличии условий вида $x \geq 1$ $K(x) ::= \ldots$ или $K(const) ::= \ldots$, к проверке состояния добавляем данные условия:

if(s->state == K && s->K_param ≥ 1)

или

if(s->state == K && s->K_param == const).

В случае, если $\alpha_i = \bar{a} \in \bar{A}$, то генерируем аналогичный код, но без передачи параметра p_i.

5.4.2. Действие прием $\alpha_i = f \in F$.

$K(x) ::= f[p_i](y).K_i(e_i(x, y))$

В функцию приемки сигнала P_i_f п.2.3.3 добавляем код:

if(s->state==K) {

if(s->p_i == p) {

s->state = K_i;

s->K_i_param = e_i(s->K_param, y);

res = 0;

}

}

В случае, если $\alpha_i = a \in A$, то генерируем аналогичный код, но без проверки s->p_i == p.

5.4.3. Действие $\alpha_i = \tau$.

В функцию P_i_step п.2.5 добавляем код:

case νlabel:

if(s->state == K) {

//Переход в состояние K_i

s->state = K_i;

s->K_i_param = e_i(s->K_param);

}

break;

5.5. Условный оператор.

$K(x) ::= \textbf{if}\ bexp(x)\ \textbf{then}\ K_1(e_1(x))\ \textbf{else}\ K_2(e_2(x))$

В функцию P_i_step п.2.5 добавляем код:

case νlabel:

if(s->state == K) {

if(bexp(s->K_param)) {

//Переход в состояние K_1

s->state = K_1;

s->K_1_param = e_1(s->K_param);

} else {

//Переход в состояние K_2

s->state = K_2;

s->K_2_param = e_2(s->K_param);

}

}

break;

6. Связывание кода драйвера со сгенерированным кодом.

Процессы драйвера имеют заданный интерфейс взаимодействия, они принимают сигналы вызова-функций обработчиков f.

Процесс можно представить следующим образом:

P_{fcall} ::=$!(\nu p)f(p_i, f_i, ctx_i, params_i).CALLED(p_i, f_i, ctx_i, params_i)$

$CALLED(p_i, f_i, ctx_i, params_i)$ ::=

$\tau.EXECUTING(p_i, f_i, ctx_i, params_i, result)$

Процесс $EXECUTING$ имеет специальное значение, его выполнение определяется кодом драйвера.

Генерируем код оберток драйвера, как для обычных процессов, кроме внутренних процессов $CALLED$ и $EXECUTING$ для них генерируем специальный код.

Добавляем в функцию P_i_step п.2.5:

```
case νlabel:
    //lock should be held here
    if(s->state==CALLED) {
      s->state=EXECUTING;
      //fptr – вызываемая фукнция драйвера,
      //ctx – контекст вызова
      //params – параметры функции
      fptr = s->fptr;
      ctx = s->ctx;
      params = s->params;
    } else {
      break;
    }
```

unlock();

//EXECUTING: вызов функции fptr

type y = *fptr(ctx, params);

lock();

s->state = RET

s->RET_param = y;

break;

Для возврата RET генерируем код, как для обычных процессов.

$RET(p_i, y) ::= \overline{f^{ret}[p_i](y)}$.

6.1. В процессе выполнения драйвер может вызывать библиотечные функции:

$$EXECUTING(p_i, f_i, ctx_i, params_i, result) ::=$$
$$\overline{g_i(p_i, e^0_{g_i})}$$
$$.\langle G\_SENT\rangle\ g_i^{ret}[p_i](result)$$
$$.\langle G\_RET\rangle\ EXECUTING(p_i, f_i, ctx_i, params_i, result)$$

Отправка сигнала $\overline{g_i}$ в коде функций.

Вызов библиотечной функции: *int result* = $g_i(x)$

int res = -1;

while(res!=0) {

lock();

if(s->state==$EXECUTING$) {

//код для посылки $\overline{g_i}$

res = g_i(s->p,x);

if(res==0) {

```
            s->state = G_SENT;
        }
    }
    unlock();
}
//Ожидание возврата значения
while(true) {
    lock();
    if(s->state==G_RET) {
        result = s->result;
        s->state=EXECUTING;
        unlock();
        break;
    }
    unlock();
}
```

6.2. Может устанавливать значения глобальных переменных:

$$EXECUTING(p_i, f_i, ctx_i, params_i, result) ::= \overline{set_{v_i}(e^0_{set_i})} .\langle SET\rangle\ EXECUTING(p_i, f_i, ctx_i, params_i, result)$$

Генерируется код посылки сигнала аналогично п.6.1.

6.3. Может читать значения глобальных переменных:

$$EXECUTING(p_i, f_i, ctx_i, params_i, result) ::= get_{v_i}(x) .\langle GET\rangle\ EXECUTING(p_i, f_i, ctx_i, params_i, result)$$

Генерируется код приема сигнала аналогично п.6.1.

A.2. Последовательный случай

1. В глобальные переменные п.2.2 добавляем:

   int thread_cnt = 0;

2. Вспомогательная функция.

   int nondet_int(int m) {

   int i = nondet_int();

   if(i<=0) return 0;

   if(i>=m) return m;

   return i;

   }

3. Вспомогательная функция, выбирающая произвольный процесс из списка и осуществляющая шаг.

   int P_i_step_any(void) {

   int k = list_size(P_i_state_list);

   if(k==0) {return 0;}

   int i = nondet_int(k-1);

   struct P_i_state *s = list_get(P_i_state_list, i);

   int stop = P_i_step(s);

   if(stop) {

   list_del(P_i_state_list, s);

```
        free(s);
        thread_cnt - -;
        if(thread_cnt==0) {
          return 1;
        }
      }
    }
```

4. Добавляем генерацию кода в п.2.7.

```
    while(true) {
      switch(nondet_int()) {
      %FOREACH P_i
        case P_i_label:
          lock();
          int stop_loop = P_i_step_any();
          unlock();
          if(stop_loop) { goto break_loop;}
          break;
      }
    }
    break_loop:
```

5. В функции create_P_i п.2.6 заменяем вызов создания потока:

```
    pthread_create(P_i_thread, s);
```

на

thread_cnt++;

6. Отправка сигналов $\overline{g_i}$ в коде функций драйвера.

   Для того, чтобы сигналы библиотечных вызовов могли быть обработаны, необходимо передать управление процессам окружения. Причем по условию теоремы, они должны выполнить свою работу без вызова функций-обработчиков драйвера.

   Заменяем код ожидания сигнала в п.6.1 подраздела А.1:

```
while(true) {
  lock();
  switch(nondet_int()) {
    %FOREACH P_i
    case P_i_label:
      int stop_loop = P_i_step_any();
      //stop должен быть 0;
      break;
  }
  //проверяем получение сигнала:
  if(s->state==G_RET) {
    result = s->result;
    s->state=EXECUTING;
    unlock();
    break;
```

```
    }
    unlock();
  }
```

A.3. Доказательство теоремы

Доказательство теоремы будем строить по индукции, основываясь на том, что при редукции π-процессов и выполнении Си программы сохраняется следующее соотношение между состояниями.

По определению процессов верхнего уровня:

$$P ::= E_1 | \cdots | E_k,$$

мы имеем k процессных выражений, причем каждое E_i может определяться либо как $K(\mathbf{const})$ или $(\nu p)K(\mathbf{const})$, в этом случае будем называть этот процесс статическим (возможна только одна копия), либо как $!\alpha_K(\mathbf{x}).K(\mathbf{x})$ или $!(\nu p)\alpha_K(\mathbf{x}).K(\mathbf{x})$, в этом случае будем называть процесс динамическим (возможно несколько копий).

Соответствие будем обозначать как $E_i \leftrightarrow P_i\_state\_list$. В случае статического процесса, E_i обозначает один процесс нижнего уровня, вызванный с некоторым параметром $K(\mathbf{const})$. Ему соответствует один объект s в списке, у которого:

s->state = K;

s->K_param = **const**;

Если $E_i ::= (\nu p)K(\mathbf{const})$, то дополнительно:

s->p = p;

В случае динамического процесса, количество процессов нижнего уровня может быть произвольным, каждому из которых соответствует объект в списке $P_i\_state\_list$.

А.3.1. Доказательство $\Rightarrow$

Пусть имеется трасса π процессов $\alpha_{i_1}, \ldots, \alpha_{i_k}$, где $\alpha_{i_j} \in D$, полученная по редукции $Sys_\pi = Sys_0 \xrightarrow{\alpha_1} Sys_1 \xrightarrow{\alpha_2} Sys_2 \cdots \xrightarrow{\alpha_n} Sys_n$, где $\alpha \in A \cup F \cup \{\tau\}$. Покажем, что существует эквивалентная ей трасса в Си программе.

Базис индукции

В соответвии с п.2.7, в начале функции *main* происходит создание состояний для статических процессов, состояние каждого из которых заносится в список $P_i\_state\_list$ в функции create$_P_i$ п.2.6. Инициализация состояния происходит в п.4.1. Так что, если $E_i ::= K(\mathbf{const})$, то

s->state = K;

s->K_param = **const**.

Если $E_i ::= (\nu p)K(\mathbf{const})$, то в п.2.7 создается уникальное значение параметра $p = globalId{+}{+}$, а в п.3 оно заносится в состояние:

s->p = p.

Для динамических процессов каждый список $P_i\_state\_list$ пуст, что соответствует тому, что еще не порождено ни одной копии процесса.

Таким образом, в Sys_0 и в начале цикла *while* функции *main* Си программы (п. 4) выполнено $\forall i : E_i \leftrightarrow P_i\_state\_list$.

Шаг индукции

Рассмотрим возможные редукции π. Пусть в Sys_{i-1} и в начале цикла *while* функции *main* выполнено $\forall i : E_i \leftrightarrow P_i\_state\_list$ и имеется редукция $Sys_{i-1} \xrightarrow{\alpha_i} Sys_i$. Покажем, что имеется выполнение в Си программе с трассой α_i, приводящее в начало цикла while, и выполнено $\forall i : E_i \leftrightarrow P_i\_state\_list$.

Рассмотрим правило редукции:

$$COMM_1(\alpha \in A) : \frac{\begin{array}{c} E_1 : K(\mathbf{x}) ::= (\cdots + \alpha_i(y).K_i(e_i(\mathbf{x}, y))) \\ E_2 : N(\mathbf{x}) ::= (\cdots + \overline{\alpha_i(e(\mathbf{x}))}.N_j(e_j(\mathbf{x}))) \end{array}}{K(\mathbf{a}) \mid N(\mathbf{b}) \xrightarrow{\alpha_i(e(\mathbf{b}))} K_i(e_i(\mathbf{a}, e(\mathbf{b})))) \mid N_j(e_j(\mathbf{b}))}$$

Так как $E_2 \leftrightarrow P_2\_state\_list$, то в списке $P_2\_state\_list$ имеется состояние s:

s->state = N;

s->N_param = $\mathbf{b}$.

В соответствии с п.4 операторе switch имеется метка *case* P_2_label. При переходе на которую будет вызвана функция P_2_step_any() (п.3). Так как $P_2\_state\_list$ содержит s, то может быть вызвана функция P_2_step(s) (п.2.5).

В P_2_step(s) в соответствии с п.5.4.1 в операторе switch была добавлена метка case, с кодом:

```
if(s->state == N) {
  //Вызов функции действия с заданным параметром
  // и проверка успешности выполнения
  int res = α_i(e(s->N_param));
  if(res==0) {
    //Переход в состояние N_j
    s->state = N_j;
    s->N_j_param = e_j(s->N_param);
  } else {
    //остаемся в текущем состоянии,
    //необходимо перевыполнить посылку.
  }
}
break;
```

По соответствию состояний условие s->state == N выполнено, поэтому осуществляется вызов функции $\alpha_i(e(\mathbf{b}))$ (п.2.3). Возможен переход на метку *case* P_i_label.

По соответствию состояний $E_1 \leftrightarrow P_1\_state\_list$, в списке $P_1\_state\_list$ имеется состояние s:

s->state = K;

s->K_param = **a**.

Потому из списка $P_1\_state\_list$ в коде α_i возможен выбор состояния s, с которым будет вызвана функция $P_i\_\alpha_i$(s, y) (п.2.3.3), где y=e(**b**).

В соответствии с п.5.4.2 в данную функцию был добавлен код приема сигнала:

```
if(s->state==K) {
  s->state = K_i;
  s->K_i_param = e_i(s->K_param, y);
  res = 0;
}
```

По соответствию состояний $E_1 \leftrightarrow P_1\_state\_list$, имеем s->state==$K$, поэтому новое состояние:

s->state = K_i;

s->K_i_param = e_i(**a**, e(**b**));

Поэтому в Sys_i выполнено $E_1 \leftrightarrow P_1\_state\_list$.

Кроме того, res=0, а значит функция $P_i\_\alpha_i$ возвращает 0, а далее функция α_i также возвращает 0.

Это означает, что прием сигнала успешно обработан, т.е. в трассу Си программы добавляется $\alpha_i(e(\mathbf{b}))$.

В P_2_step(s), так как возвращаемое значение функции α_i равно 0, выполняется код:

```
//Переход в состояние N_j
s->state = N_j;
s->N_j_param = e_j(s->N_param);
```

Что дает нам:

s->state = N_j;

s->N_j_param = $e_j(\mathbf{b})$;

Поэтому в Sys_i выполнено $E_2 \leftrightarrow P_2\_state\_list$.

Возвращаемся в основной цикл while в функции $main$.

Что и требовалось доказать для шага индукции.

Для правила редукции $COMM_2(\alpha \in F)$ доказательство аналогичное с учетом условий на проверку параметра динамической метки.

Для правил $IFTHEN$, $IFELSE$, $STRUCT$ – аналогично.

Для правила TAU не происходит вызов функции приема, действие всегда успешное.

Для доказательства PAR достаточно заметить, что состояние процесса не участвующего во взаимодействии не меняется.

В правиле RES значение параметра метки доступно любому процессу нижнего уровня, что обеспечивается его сохранением в состоянии процесса.

Если в правиле редукции встречается сигнал драйвера, то по условию теоремы $Drv_\pi \approx_D Drv_C$, поэтому имеется необходимое выполнение Си программы, при этом сохраняется соответствие состояний.

A.3.2. Доказательство $\Leftarrow$

Пусть имеется трасса Си программы $\alpha_1, \ldots, \alpha_k$, где $\alpha_i \in D$. Покажем что существует эквивалентная ей трасса в Sys_π.

Базис индукции

Аналогично доказательству $\Rightarrow$ устанавливаем, в Sys_0 и в начале цикла *while* функции $main$ Си программы п. 4 выполнено $\forall i : E_i \leftrightarrow P_i\_state\_list$.

Шаг индукции

Пусть в Sys_{i-1} и в начале цикла *while* функции $main$ выполнено $\forall i : E_i \leftrightarrow P_i\_state\_list$ и имеется выполнение в Си программе с трассой α_i,

приводящее в начало цикла *while*.

Покажем, что имеется редукция $Sys_{i-1} \xrightarrow{\alpha_{j_1}} Sys_{j_1} \cdots \xrightarrow{\alpha_{j_m}} Sys_{j_m} \xrightarrow{\alpha_i} Sys_i$, $\alpha_{j_l} \notin D$, и в Sys_i выполнено $\forall i : E_i \leftrightarrow P_i\_state\_list$.

Тело цикла *while* (п.4) состоит из оператора switch, в котором может быть выполнена одна из меток *case* P_j_label. При переходе на данную метку возможно два варианта:

1. осуществлен возврат с кодом 0, состояния $P_j\_state\_list$ не меняются.
2. будет вызвана функция P_j_step_any() (п.3).

В первом случае редукции π-процессов не происходит, состояния $P_j\_state\_list$ не меняются, поэтому $\forall i : E_i \leftrightarrow P_i\_state\_list$.

Во втором случае вызывается функция P_j_step(s) (п.2.5) для одного из состояний s, соответствующего состоянию процесса N (s->state=N, s->N_param = $\mathbf{b}$, s->p = p_1). Выполнение может продолжиться по одной из меток посылки сигнала в операторе switch.

Код меток генерируется на основании п.5, а именно подпунктов:

- п.5.3 для $N ::= 0$ – осуществляется возврат 1 после чего в функции P_j_step_any состояние s удаляется из списка $P_j\_state\_list$. В новом процессе Sys' удаляем процесс N, при этом $Sys' \equiv Sys$.
- п.5.4.1 для $\overline{\hat{\alpha}[p_i](e'_i(x))}.K_i(e_i(x))$. Возможно два варианта:
  1. Процесс не находится в нужном состоянии, т.е. не выполнено условие на состояние или на параметр динамической метки, в этом случае редукции π-процессов не происходит, состояния $P_j\_state\_list$ не меняются, поэтому $\forall i : E_i \leftrightarrow P_i\_state\_list$.
  2. Вызываем функцию приема сигнала в п.2.3:

     int res = $\hat{\alpha}$(p, $e(\mathbf{b})$);

Возможно два варианта:

2.1. $\hat{\alpha}$ – обычный сигнал (п.2.3.1);

Переходим на одну из меток *case* P_i*_label*.

Возможно два варианта:

2.1.1. Список P_i_state_list пуст.

2.1.2. Список P_i_state_list не пуст.

В первом случае редукции π-процессов не происходит, состояния $P_j\_state\_list$ не меняются, поэтому $\forall i : E_i \leftrightarrow P_i\_state\_list$.

Во втором случае из P_i_state_list читаем состояние s соответствующее процессу K (s->state=K, s->K_param = $\mathbf{a}$, s->p = p_2), с которым вызваем функцию $P_i\_\hat{\alpha}$(p, s, y) (п.2.3.3).

В соответствии с п.5.4.2 в данную функцию были добавлены коды приема сигналов. Переходим на одну из меток приема сигнала, возможно два варианта:

2.1.1. Процесс не находится в нужном состоянии, т.е. не выполнено условие на состояние или на параметр динамической метки, в этом случае редукции π-процессов не происходит, состояния $P_i\_state\_list$ не меняются, поэтому $\forall i : E_i \leftrightarrow P_i\_state\_list$.

2.1.2. Процесс находится в нужном состоянии и выполнено условие на параметр динамической метки. В этом случае происходит изменение состояния s из списка $P_i\_state\_list$.

```
if(s->state==K) {
  if(s->p_i == p) {
    s->state = K_i;
    s->K_i_param = e_i(s->K_param, y);
```

res = 0;

}

}

Кроме того, res=0, а значит функция $P_i\_\hat{\alpha}$ возвращает 0, и далее функция $\hat{\alpha}$ также возвращает 0. Это означает, что прием сигнала успешно обработан, т.е. в трассу Си программы добавляется $\hat{\alpha}(e(\mathbf{b}))$.

Так как возвращаемое значение функции $\hat{\alpha}$ равно 0, в P_j_step(s) выполняется код:

//Переход в состояние N_j

s->state = N_j;

s->N_j_param = e_j(s->N_param);

Что дает нам:

s->state = N_j;

s->N_j_param = $e_j(\mathbf{b})$;

Если $\hat{\alpha} \in A$, то это дает нам правило редукции:

$$COMM_1 : \frac{\begin{array}{c} E_i : K(\mathbf{x}) ::= (\cdots + \alpha_i(y).K_i(e_i(\mathbf{x}, y))) \\ E_j : N(\mathbf{x}) ::= (\cdots + \overline{\alpha_i(e(\mathbf{x}))}.N_j(e_j(\mathbf{x}))) \end{array}}{K(\mathbf{a}) \mid N(\mathbf{b}) \stackrel{\alpha_i(e(\mathbf{b}))}{\longrightarrow} K_i(e_i(\mathbf{a}, e(\mathbf{b})))) \mid N_j(e_j(\mathbf{b}))}$$

Если $\hat{\alpha} \in F$, то правило $COMM_2$.

2.2. $\hat{\alpha}$ – это сигнал создания копии, вызываем функцию $create_i$ (п.2.3.2).

Создаем правило редукции по структурной эквивалентности

$$!\alpha(\mathbf{x}).K(\mathbf{x}) \equiv \alpha(\mathbf{x}).K(\mathbf{x}) \mid !\alpha(\mathbf{x}).K(\mathbf{x})$$

и по взаимодействию $COMM_1$.

3. п.5.4.3 для τ – аналогично.

- п.5.5 условный оператор – создаем редукцию $IFTHEN$, если идем по ветке then, и $IFELSE$ – иначе.

Продолжаем процесс построения редукции до тех пор, пока не встретим $\alpha \in D$, которое должно встретиться в силу предположения индукции.

Если встречается вызов функции драйвера, то по условию теоремы $Drv_\pi \approx_D Drv_C$, поэтому имеется необходимая редукция, при этом сохраняется соответствие состояний.