ДИАГНОСТИКА СИНТАКСИЧЕСКОЙ СОВМЕСТИМОСТИ ПРАВИЛ КОРРЕКТНОСТИ С ЯДРОМ ОС LINUX ПРИ СТАТИЧЕСКОЙ ВЕРИФИКАЦИИ ДРАЙВЕРОВ

Щепетков Илья Викторович
Стажер-исследователь
Институт системного программирования РАН
Тел. +7(495)912-56-59 (доб. 4454)
e-mail: shchepetkov@ispras.ru

Новиков Евгений Михайлович
Стажер-исследователь
Институт системного программирования РАН
Тел. +7(495)912-56-59 (доб. 4454)
e-mail: novikov@ispras.ru

Введение

Ядро ОС Linux – один из самых динамично развивающихся проектов в мире, в котором участвуют тысячи разработчиков. Ядро насчитывает свыше 15 миллионов строк кода, и каждые 2 – 3 месяца выпускаются новые релизы, содержащие 8 – 12 тысяч изменений каждый [1].

Наибольшую часть ядра занимают драйверы (около 10 миллионов строк кода [1]), причем в их исходном коде содержится достаточно большое количество разнообразных ошибок. Так как драйверы работают с тем же уровнем привилегий, что и остальное ядро, то каждая такая ошибка снижает общую стабильность работы операционной системы, а так же может привести к зависанию или падению [2], [3]. Обеспечивать высокую надежность драйверов вручную практически невозможно 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 языке Си, и заканчивая правилами, которые говорят о том, как драйверы должны использовать программные интерфейсы ядра.

Для проверки правил корректного использования программных интерфейсов в исходном коде драйверов была разработана открытая система статической верификации Linux Driver Verification (LDV) [4-6]. Упомянутые выше правила корректности в LDV задаются с помощью спецификаций. Спецификация включает в себя:

- модельное состояние (набор глобальных переменных, отображающий реальное состояние ядра ОС Linux);

- модельную реализацию программного интерфейса (набор функций, выполняющий изменения модельного состояния в соответствии с семантикой моделируемого интерфейса, а также проверяющий корректность использования данного интерфейса);
- привязку точек использования программного интерфейса ядра в драйверах к модельным реализациям интерфейса.

На Рис. 1 проиллюстрирована связь между отдельными компонентами спецификации, а также между спецификацией правила на корректное использование мьютексов и программным интерфейсом ядра.

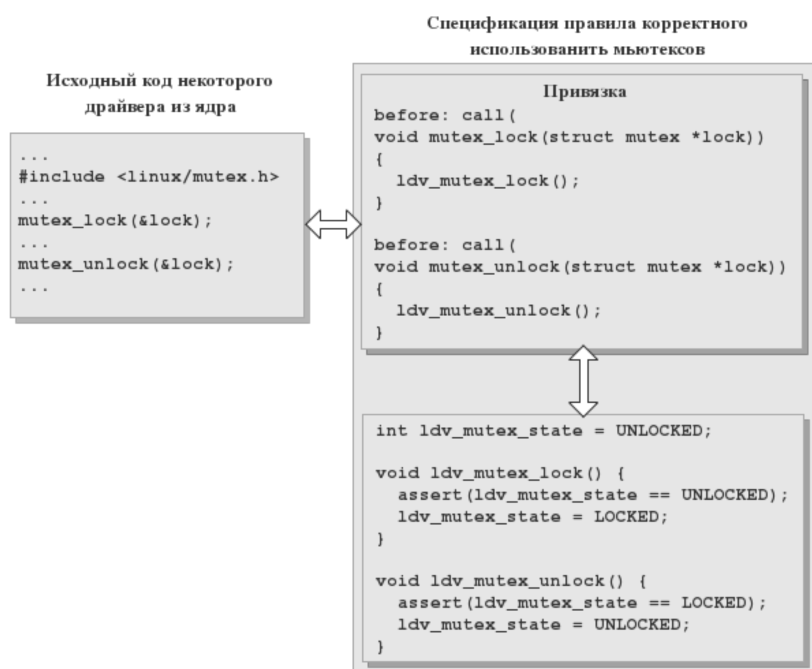

Рисунок 1. Связь спецификации правила на корректное использование мьютексов с программным интерфейсом ядра

Рассмотрим приведённую иллюстрацию подробнее. В исходном коде драйвера присутствует вызов функций *mutex_lock* и *mutex_unlock* из программного интерфейса ядра, определенного в заголовочном файле *mutex.h*. В спецификации определяются модельное состояние *ldv_mutex_state* (которое говорит, что мьютекс захвачен/освобождён) и

модельные реализации проверяемого интерфейса (функции *ldv_mutex_lock* и *ldv_mutex_unlock*). Также в спецификации задается привязка точек вызова функций *mutex_lock* и *mutex_unlock* к модельным функциям *ldv_mutex_lock* и *ldv_mutex_unlock* соответственно. Привязка осуществляется с помощью конструкций аспектно-ориентированного программирования [7].

Однако программный интерфейс ядра ОС Linux постоянно расширяется и не является стабильным [1]. В рамках данной работы было проведено дополнительное исследование, которое показало, что в среднем за год изменяется порядка 5% программных интерфейсов ядра. В результате этих изменений может нарушаться привязка между точками использования программного интерфейса и его модельной реализацией

На текущий момент в LDV формализовано свыше 40 спецификаций правил корректного использования программных интерфейсов. В каждой спецификации обычно содержится несколько модельных функций. На текущий момент количество привязок, за синтаксической совместимостью которых с программными интерфейсами ядра необходимо следить, равняется 325.

Таким образом, для поддержания спецификаций в актуальном состоянии возникает необходимость диагностики синтаксической совместимости программного интерфейса ядра и привязки.

Существующие решения

Наблюдение за изменениями в программных интерфейсах ядра вручную требует много времени и слишком трудозатратно. Также при таком подходе достаточно велика вероятность ошибок.

Для решения проблемы диагностики синтаксической совместимости правил корректности с ядром в рамках проекта LDV было предложено разрабатывать тестовый набор драйверов для каждой спецификации. Каждый тестовый драйвер содержал вызов одной или нескольких функций или подстановок макрофункций из моделируемого в спецификации программного интерфейса ядра, причем вызовы должны быть составлены либо в соответствии с правилом корректного использования, либо нарочито ошибочно. Вердикт системы верификации LDV на каждом тестовом драйвере в соответствии со спецификацией должен быть SAFE (если исходный код драйвера написан в соответствии с моделируемым правилом корректностью), или UNSAFE (если исходный код содержит неправильное использование программного интерфейса ядра). В случае же изменений в программном интерфейсе ядра вердикты изменятся с UNSAFE на SAFE (или наоборот), либо возникнет ошибка на этапе компиляции драйвера.

Такой подход имеет целый ряд преимуществ перед ручным: в долгосрочной перспективе он требует гораздо меньше времени и он менее трудозатратен. Количество возможных ошибок также значительно снижается. Тем не менее, и у него есть серьезные недостатки:

- Для новых версий программных интерфейсов ядра может возникнуть необходимость доработать наборы тестовых драйверов.
- Изменения в компонентах системы верификации LDV могут повлиять на результаты запуска, что в свою очередь не позволяет быть уверенным в достоверности проведенной диагностики.
- Необходимость писать большое количество тестовых драйверов для каждой спецификации. На текущий момент для 33 спецификаций написано порядка 1100 драйверов.
- Запуск системы верификации LDV на всех написанных тестовых наборах длится более 13 часов, и это время увеличивается с ростом количества спецификаций.

Таким образом, возникла потребность разработать новый метод, который был бы лишен описанных выше недостатков. Далее в данной работе предлагается подход, позволяющий решить задачу автоматизации диагностики синтаксической совместимости правил корректности с ядром ОС Linux, и представляются полученные результаты.

Метод диагностики синтаксической совместимости правил корректности с ядром ОС Linux

Предлагаемый подход заключается в сопоставлении деклараций или определений функций/макросов, которые задаются в привязках спецификаций, и деклараций или определений функций/макросов в заголовочных файлах ядра, которые представляют его программные интерфейсы. Для подобного сопоставления предлагается осуществлять поиск по исходному коду ядра ОС Linux. Если результаты поиска будут успешными и среди программного интерфейса ядра будут найдены искомые декларации и определения, то соответствующую этим декларациям/определениям привязку можно считать совместимой с данной версией ядра.

Если же поиск по исходному коду завершился с отрицательным результатом, то в программном интерфейсе ядра произошло одно из следующих изменений:

1. У функции поменяли тип возвращаемого значения.

2. У функции изменились типы аргументов или их количество.
3. Функцию заменили одноименным макросом (или наоборот).
4. У макроса поменялось количество аргументов.
5. Функцию или макрос либо удалили из ядра, либо переименовали.

Варьируя параметры поиска (например, поиск декларации функции без учета типа возвращаемого значения), можно определить тип подобного изменения.

Реализация предложенного метода

Реализация представленного выше подхода основывается на возможностях C Instrumentation Framework (CIF) по осуществлению запросов по исходному коду [7-9]. CIF – это реализация аспектно-ориентированного программирования для языка Си. Он разработан в рамках проекта LDV и основан на компиляторе GCC. В этом проекте возможности CIF используются для формализации правил корректности в виде спецификаций и для подготовки исходного кода драйверов к верификации путем сопоставления привязок и точек использования программных интерфейсов ядра и последующего инструментирования кода.

Запросы по исходному коду помогают обнаруживать искомые фрагменты кода и определять их взаимоотношения друг с другом. Для реализации предлагаемого подхода в рамках работы в CIF была добавлена возможность осуществления запросов по исходному коду на поиск декларации функции или определения макроса.

Алгоритм диагностики синтаксической совместимости с помощью последовательности запросов по исходному коду

Продемонстрируем работу предлагаемого алгоритма на конкретном примере. Пусть в привязке задана связь между функцией *«int func(int a)»* и соответствующей модельной функцией. Требуется провести диагностику синтаксической совместимости с некоторой версией ядра ОС Linux. Тогда алгоритм действий будет следующим:

1. Сначала выполняется запрос по исходному коду на декларацию функции, точно соответствующей заданной сигнатуре:

```
info: declare_func(int func(int a))
{
  $fprintf<"results.txt", "Matched function signature">
  $fprintf<"results.txt", "%s", $signature>
}
```

Если такая функция среди программного интерфейса ядра найдена, то будет выполнен заключенный в фигурные скобки код: в файл results.txt запишется сообщение с сигнатурой обнаруженной в ядре декларации.

2. Если же предыдущий запрос завершился неудачно, то делается запрос, в котором в сигнатуре тип возвращаемого значения заменен на "$" - шаблон, позволяющий задавать произвольный тип возвращаемого значения (предполагается, что у функции изменился тип возвращаемого значения):

```
info: declare_func($ func(int a))
{
  $fprintf<"results.txt", "Another return type">
  $fprintf<"results.txt", "%s", $signature>
}
```

3. Далее, опять же в случае неудачи предыдущего запроса, предполагается, что у функции изменились аргументы, а не тип возвращаемого значения, и выполняется запрос, в котором аргументы функции заменены на ".." – шаблон, позволяющий задавать любое количество аргументов любого типа:

```
info: declare_func(int func(..))
{
  $fprintf<"results.txt", "Another set of parameters">
  $fprintf<"results.txt", "%s", $signature>
}
```

4. Предполагается изменение как типа возвращаемого значения, так и аргументов, и делается соответствующий запрос:

```
info: declare_func($ func(..))
{
  $fprintf<"results.txt", "Another return type and set of parameters">
  $fprintf<"results.txt", "%s", $signature>
}
```

5. Предполагается превращение функции в одноименный макрос и делается запрос, тип которого изменён с запроса на обнаружение декларации функции на запрос на обнаружение определения макроса:

```
info: define(func(a))
{
  $fprintf<"results.txt", "Matched similar macro signature">
  $fprintf<"results.txt", "%s", $signature>
}
```

6. В случае неудачного завершения всех предыдущих пунктов в файл с результатами печатается следующее сообщение:

```
$fprintf<"results.txt", "Nothing was found">
```

Соответствующий алгоритм для макроса несколько проще, но в целом аналогичен.

Архитектура разработанного инструмента

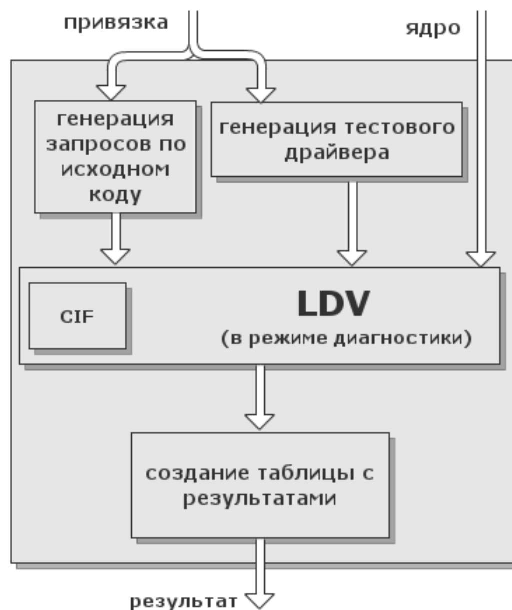

Рисунок 2. Архитектура инструмента, реализующего предложенный подход.

На Рис. 2 приведена архитектура инструмента *test-compatibility*, реализующего предложенный подход. На вход инструменту подаётся привязка и ядро ОС Linux, с которым планируется осуществить диагностику совместимости. Первый этап работы – генерация файла с последовательностями запросов по исходному коду для каждой функции и макроса из привязки.

Второй этап – генерация тестового драйвера. Тестовый драйвер необходим для того, чтобы конкретизировать расположение деклараций и определений используемых в привязке функций и макрофункций в программном интерфейсе ядра. Благодаря тестовому драйверу, запуск диагностики осуществляется не на всём ядре, а только на отдельных заголовочных файлах. Разработчик спецификации должен заранее задать список заголовочных файлов, в которых определены используемые в привязке функции и макрофункции, в специальном файле, и на его основе инструмент генерирует тестовый драйвер, состоящий только из последовательности директив *#include* <...>. Данный специальный файл – общий для всех спецификаций. Более того,

формат файла позволяет задавать подобный список заголовочных файлов отдельно для каждой требуемой версии ядра.

Последний этап – основной, на нём производится диагностика совместимости и по её завершении получаются результаты. Диагностикой занимается сама система верификации LDV, запущенная в специальном режиме. Этот режим позволяет избежать запуска ненужных для диагностики компонентов LDV. На вход LDV подаётся ядро, с программным интерфейсом которого планируется произвести диагностику; привязка и тестовый драйвер. После запуска LDV возвращает текстовый файл с результатами, который затем для большей наглядности преобразуется инструментом в таблицу.

Результаты

Для получения результатов были выбраны 6 различных версий ядер ОС Linux и 38 привязок спецификаций. 5 ядер были выбраны как версии с “поддержкой в течение длительного периода” – они до сих пор обновляются и соответственно актуальные. Шестая версия ядра – последняя на момент написания статьи 3.9-rc8.

Так как система тестов используется уже достаточно давно, ожидалось, что существующие привязки уже совместимы с предлагаемым набором ядер, а изменений стоит ожидать лишь при последующих запусках на новых версиях ядер. Однако было обнаружено достаточно значительное количество ошибок и несовместимостей.

В процессе анализа были выявлены ошибки в 17 привязках из 38 – т.е. затронута почти половина. Чуть более конкретная статистика для каждой версии ядра:

- 2.6.32.60: Проблемы с совместимостью есть у 13 привязок. Всего проблем: у 19 функций/макрофункций.
- 2.6.34.14: Проблемы с совместимостью есть у 10 привязок. Всего проблем: 11.
- 3.0.74: Проблемы с совместимостью есть у 8 привязок. Всего проблем: 11.
- 3.2.43: Проблемы с совместимостью есть у 8 привязок. Всего проблем: 12.
- 3.4.41: Проблемы с совместимостью есть у 9 привязок. Всего проблем: 14.
- 3.9-rc8: Проблемы с совместимостью есть у 11 привязок. Всего проблем: 16.

Большей частью проблемы связаны с изменениями в программном интерфейсе ядра, однако некоторые возникли из-за невнимательности разработчиков соответствующей спецификации.

Анализ запусков также показал, что удалось добиться значительного снижения времени проведения диагностики. Диагностика синтаксической совместимости всех 38 привязок с одной версией ядра длится всего несколько минут, тогда как аналогичный запуск системы тестов продолжается свыше десяти часов.

Заключение

В работе рассмотрен подход к диагностике синтаксической совместимости правил корректности с ядром ОС Linux при статической верификации драйверов. Эффективность диагностики существенно возросла за счёт уменьшения времени ей проведения (с нескольких часов до нескольких минут) и максимального упрощения необходимых действий для её запуска. В настоящее время инструмент, реализующий предложенный подход, активно применяется на практике в проекте LDV.

Несмотря на уже достигнутые успехи, имеется несколько направлений дальнейшего развития подхода, которые позволят ещё более упростить диагностику совместимости правил корректности с ядром:

1. Добавить информацию о конкретном местоположении найденной декларации или объявления функции/макрофункции в исходном коде в таблицу с результатами. Это позволит быстрее анализировать полученные в результате диагностики данные.
2. Реализовать возможность подробного сравнения результатов двух запусков для последующего отслеживания регрессий в спецификациях.

Инструмент *test-compatibility*, реализующий предложенный подход, доступен на сайте разработки проекта статической верификации драйверов ОС Linux *http://forge.ispras.ru/projects/ldv*.

Литература

[1]. J. Corbet, G. Kroah-Hartman, A. McPherson. Linux Kernel Development. http://go.linuxfoundation.org/who-writes-linux-2012, March 2012.

[2]. Chou. An Empirical Study of Operating System Errors. Proc. 18th ACM Symp. Operating System Principles, ACM Press, 2001.

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

[4]. В.С. Мутилин, Е.М. Новиков, А.В. Страх, А.В. Хорошилов, П.Е. Швед. Архитектура Linux Driver Verification. Труды Института системного программирования РАН, том 20, стр. 163-187, 2011.

[5]. A. Khoroshilov, V. Mutilin, E. Novikov, P. Shved, A. Strakh. Towards an Open Framework for C Verification Tools Benchmarking. Proceedings of the Eighth International Andrei Ershov Memorial Conference «Perspectives of Systems Informatics» (PSI 2011), pp. 82-91, 2011.

[6]. Проект статической верификации драйверов Linux Driver Verification. http://linuxtesting.org/project/ldv.

[7]. E. Novikov. One Approach to Aspect-Oriented Programming Implementation for the C programming language. Proceedings of the 5th Spring/Summer Young Researchers' Colloquium on Software Engineering, Yekaterinburg, pp. 74-81, 12-13 May, 2011.

[8]. Реализация аспектно-ориентированного программирования для языка Си C Instrumentation Framework. http://forge.ispras.ru/projects/cif.

[9]. Е.М. Новиков, А.В. Хорошилов. Использование аспектно-ориентированного программирования для выполнения запросов по исходному коду программ. Труды Института системного программирования РАН, т. 23, стр. 371-386, 2012.