Формальная верификация библиотечных функций ядра Linux[1]

[1]Д.В. Ефремов <defremov@hse.ru>
[2]М.У. Мандрыкин <mandrykin@ispras.ru>
[1]НИУ Высшая школа экономики,
101000, Россия, г. Москва, ул. Мясницкая, д. 20
[2]Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. В статье авторами рассматриваются результаты дедуктивной верификации набора из 26 библиотечных функций ядра ОС Linux с помощью стека инструментов AstraVer. В набор включены преимущественно функции, работающие с данными строкового типа. Целью верификации является доказательство свойств функциональной корректности. В статье рассматриваются аналогичные работы по верификации, сравниваются полученные результаты, рассматривается ряд проблем, с которыми сталкивались авторы предыдущих работ, в том числе проблемы, с которыми удалось справится в рамках данной работы и те, которые все ещё препятствуют успешной верификации. Также предлагается методология разработки спецификаций, примененная для рассматриваемого набора функций, которая включает некоторые шаблонные приёмы разработки спецификаций. Авторам удалось доказать полную корректность двадцати пяти функций. В статье приведены результаты доказательства полученных условий верификации каждой функции с помощью нескольких современных SMT-солверов.

Ключевые слова: статический анализ; формальная верификация; дедуктивная верификация; функции стандартной библиотеки.

DOI: 10.15514/ISPRAS-2017-29(6)-3

Для цитирования: Ефремов Д.В, Мандрыкин М.У. Формальная верификация библиотечных функций ядра Linux. Труды ИСП РАН, том 29, вып. 6, 2017 г., стр. 49-76. DOI: 10.15514/ISPRAS-2017-29(6)-3

1. Введение

Ошибки и дефекты в критически важных компонентах операционной системы могут привести к полной её компрометации. Одним из средств повышения

[1] Эта работа поддержана грантом РФФИ 15-01-03024.

уровня доверия к системе является дедуктивная верификация. Она позволяет для отдельных типов дефектов доказать их отсутствие в исследуемом исходном коде.

Код ядра Linux написан на языке Си. В коде ядра широко используется перетипирование указателей, указатели на функции, битовые операции, нестандартные расширения и другие возможности языка Си, осложняющие дедуктивный анализ соответствующих фрагментов Си-кода. Инструменты дедуктивной верификации, как правило, либо не поддерживают такие возможности языка Си, либо не позволяют доказывать для использующих их участков кода соответствие нетривиальным спецификациям функциональной корректности. Большинство инструментов дедуктивной верификации Си-кода налагают явные или неявные ограничения на синтаксис языка и общий стиль кодирования [1].

При верификации кода ядра Linux приходится сталкиваться со многими ограничениями инструментов дедуктивной верификации, их недостаточной зрелостью для полноценной поддержки подобного кода. Большинство ограничений может быть снято в том случае, если разрешена модификация кода. В таком случае возможно заменить неподдерживаемые конструкции или же участки кода, плохо поддающиеся дедуктивной верификации, семантически эквивалентным кодом, который легко обрабатывается инструментами.

В проектах по верификации это считается приемлемым и желательным, так как стоимость доработки инструментов многократно превышает стоимость переписывания части кода проекта.

Однако при верификации, например, модуля ядра Linux невозможно снять все подобные ограничения, так как модуль, являясь частью самого ядра, основывается на его структурах данных, интерфейсе и наследует общий стиль кодирования ядра.

По этой причине в данной работе авторы уделяют особое внимание тому, чтобы доказывать код в максимально неизменном виде.

Одной из основных целей проекта AstraVer [2] является доработка инструментов дедуктивной верификации для работы с кодом ядра Linux. Несмотря на существенные продвижения [3, 4] в разработке инструментов, на момент начала данной работы не существовало цельного и репрезентативного набора тестовых примеров, наглядно отражающего текущее состояние прогресса в направлении достижения целей этого проекта. Данная работа призвана восполнить этот пробел.

Хотя выбранные для анализа библиотечные функции не полностью покрывают сложные для анализа конструкции, используемые в коде ядра Linux, рассматриваемый в работе набор достаточно репрезентативен и позволяет как производить сравнение с аналогичными работами [5, 6], так и наглядно показывать основные сложности моделирования Си-кода с помощью инструментов дедуктивной верификации. Данный набор позволил выявить целый ряд ошибок и недостатков в инструментах AstraVer, которые на

текущий момент уже были исправлены. Также были выявлены некоторые фундаментальные ограничения, присущие используемым в инструментах моделям сущностей языка Си, таких как значения целочисленных типов данных, указатели, адреса, выделенные блоки памяти и др. Эти ограничения не позволяют доказать некоторые свойства реализаций алгоритмов и подробно рассматриваются в данной статье.

Основные результаты данной работы состоят в следующем:

- Были выявлены и подробно рассмотрены основные ограничения методов моделирования целых чисел и указателей в инструментах дедуктивной верификации AstraVer.
- Был разработан и применён на практике ряд приёмов спецификации и доказательства корректности, облегчающий применение инструментов дедуктивной верификации к коду ядра ОС Linux. В том числе использование дублирующих спецификаций, формализующих поведение целевых реализационных функций как с помощью постусловий, так и с помощью модельных функций, а также разработка спецификаций функций на основе их реализаций и контекстов использования.
- Была успешно доказана корректность 26 функций ядра Linux, 23 из которых были доказаны без какого-либо изменения их исходного кода.

Также в рамках работы была предложена, реализована и применена на практике новая композитная модель целых чисел для модуля дедуктивной верификации AstraVer, облегчающая моделирование операций с беззнаковыми целыми и преобразований между различными целочисленными типами данных.

Насколько известно авторам данной статьи, библиотечные функции ядра Linux ранее не анализировались методом дедуктивной верификации систематически. Основной практический результат данной работы — разработанные спецификации библиотечных функций ядра Linux и доказательство соответствия реализаций функций разработанным спецификациям с помощью инструментов дедуктивной верификации AstraVer.

Код функций, спецификации и протоколы доказательств выложены в открытом доступе вместе с инструкциями по воспроизведению результата [7]. Спецификации с исходным кодом могут в дальнейшем служить тестовым набором для инструментов дедуктивной верификации и солверов.

В статье приводится сравнение результатов данной работы с работами по дедуктивной верификации стандартных библиотечных функций klibc [5] и OpenBSD [6].

Статья построена следующим образом: в секции 2 даётся описание применяемых в работе инструментов верификации; в секции 3

рассматриваются аналогичные проекты по разработке спецификаций; в секции 4 кратко описан использованный язык спецификаций ACSL; в секции 5 описываются изменения, которые были внесены в инструменты верификации в ходе данной работы; в секции 6 описываются подходы к разработке спецификаций, которые были выработаны и применены в процессе работы; в секции 7 описываются проблемы, которые не удалось решить; в секции 8 представлены результаты работы.

2. Инструменты верификации

Существуют различные инструменты для дедуктивной верификации программ, в том числе и для верификации кода на языке Си. Инструмент FRAMA-C [8] является фреймворком, позволяющим реализовывать и комбинировать разные виды статического анализа. Для аннотирования кода FRAMA-C использует специальный язык ANSI/ISO C Specification Language [9]. Это язык описания поведения. Он поддерживает написание контрактов функций (на уровне пред- и постусловий), а также инвариантов циклов и аксиоматических теорий. FRAMA-C интегрирует спецификации и код в единое синтаксическое дерево, с которым работают плагины. Для FRAMA-C существует несколько плагинов для дедуктивной верификации кода: WP [8], JESSIE [10] и ASTRAVER [2]. Последний плагин является форком JESSIE. Одна из основных целей, которая заявлена его авторами — это доработка инструмента для верификации кода ядра Linux. Для доказательства функций авторы использовали плагин ASTRAVER.

Плагин дедуктивной верификации ASTRAVER (как и JESSIE) транслирует внутреннее представление FRAMA-C в модель программы на языке WhyML [11], на основе реализованных в нем моделей памяти и операций с числами.

Инструмент WHY3 генерирует условия верификации для программы на языке Why3ML и преобразует их во входные задания для солверов. Среди поддерживаемых WHY3 — такие солверы, как ALT-ERGO, CVC3, CVC4, Z3, SPASS, EPROVER, SIMPLIFY и большое количество других. WHY3 также поддерживает ряд трансформаций для условий верификации, например, подразбиение на отдельные условия верификации по конъюнкциям.

3. Аналогичные работы

Так как инструменты дедуктивной верификации WP и JESSIE являются достаточно зрелыми, ранее они уже успешно применялись для верификации реального кода. Так, в работе [6] доказывается корректность 12 стандартных функций строкового типа, реализованных в OpenBSD. Авторами используется JESSIE в качестве плагина дедуктивной верификации. Полностью корректность (валидность всех условий верификации) удалось доказать для 7 функций, для остальных 5 функций несколько условий верификации остались

недоказанными. Особенностью работы является то, что автор для каждой функции делал три итерации по разработке спецификационного контракта: первую — на основе стандарта и опыта самого автора, вторую — на основе документации (`man`), третью - на основе документации и кода функции. Последняя версия спецификации практически в каждом случае сильно отличалась от первых двух. Это показывает, что разработать формальную спецификацию на существующую реализацию, без доступа к самой реализации, достаточно сложно. Однако подобный итерационный подход позволил автору найти неточности в документации к нескольким функциям, а также отсутствие полноты описания поведения функции в ряде случаев.

Для некоторых функций автору потребовалось внести изменения в их код. Это было связано с двумя конкретными ситуациями. В первой указатели на тип `char` в функциях `strcmp` и `strncmp` приводились к указателям на `unsigned char`. Во второй в функции `strlcat` итератор цикла переполнялся на последнем шаге итерации за счет постфиксного декремента, что приводило к невозможности доказать условие верификации на целочисленное переполнение, хотя и не вело к ошибке при выполнении кода функции. Для доказательства условий верификации использовались солверы Alt-Ergo (0.7.3), Simplify (1.5.4) и Z3 (2.0).

В статье [5] авторы используют Frama-C с плагином дедуктивной верификации WP для верификации кода функций библиотеки klibc. Авторам удалось полностью доказать корректность 14 функций, работающих с данными строкового типа, для 12 функций не удалось доказать часть условий верификации, ещё на 4 функциях проявили себя ошибки в инструментах, воспрепятствовавшие их полноценному запуску. Помимо функций для строкового типа (из `string.h`) в работе также анализировались функции из `stdio.h`. Как отмечается самими авторами, практически все функции из данного заголовочного файла используют системные вызовы, что в конечном итоге ведёт к тому, что спецификации для них получаются слабыми (weak). При работе с кодом функций авторы вносили в него изменения, которые позволяли обходить ограничения применяемых инструментов или упрощать результирующие условия верификации.

Так, авторы заранее изучили проблемы с моделированием перетипирования указателей (type casts), например, `unsigned char*` в `char*`, и старались изменить код, для того чтобы исключить подобные операции.

Помимо этого, проблемы вызвал повторяющийся шаблон кода, где в цикле `while` осуществляется постфиксное уменьшение значения переменной беззнакового типа. Выход из цикла в таком случае происходит, когда переменная равна нулю, но после сравнения всё равно осуществляется уменьшение значения переменной на единицу. В случае беззнакового типа это приводит целочисленному переполнению. В реальности, однако, целочисленное переполнение не ведёт к ошибке в коде функции, так как после

выхода из цикла переменная нигде не используется. Но в такой ситуации не удаётся доказать условие верификации, требующее отсутствия переполнения.

Для доказательства условий верификации использовались солверы ALT-ERGO (0.95.1), CVC3 (2.4.1), Z3 (4.3.1).

Самым развёрнутым документом по разработке спецификаций на языке ACSL является ACSL by Example [12]. В нем доказывается корректность функций из стандартной библиотеки C++. Функции перед разработкой спецификаций переписываются с обобщённой реализации на шаблонах в функции языка Си, работающие на массивах элементов типа `int`. Разработчики регулярно обновляют отчёт добавлением новых функций со спецификациями к ним, исправляют ошибки в прошлых спецификациях и перерабатывают их. Проект длится с 2009 года. Документ содержит большое количество полностью доказанных функций. В качестве солверов используются ALT-ERGO, CVC3, CVC4, Z3, EPROVER. Используется плагин дедуктивной верификации WP.

Отчёт компании GrammaTech [13] содержит описание типовых проблем, с которыми столкнулись авторы при разработке спецификаций к реализации стандартной библиотеки GT libc. Они использовали FRAMA-C с плагином дедуктивной верификации WP. Среди прочего авторами описываются проблемы моделей памяти, возникающие при верификации кода с перетипированием указателей, а также со сравнением указателей.

4. Язык спецификаций ACSL

ACSL является языком спецификации поведения интерфейсов (BISL, Behavioral Interface Specification Language) [14], реализованным во FRAMA-C. ACSL разработан специально для спецификации свойств Си-программ и подходит для написания контрактных спецификаций (пред- и постусловий), формализации свойств безопасности (предикатов на достижимые состояния программы), а также для задания дополнительных спецификаций, необходимых инструменту верификации для проверки контрактных спецификаций и свойств безопасности. ACSL включает средства выражения специфичных для языка Си аспектов управления памятью, таких как адреса и длины выделяемых блоков памяти, преобразования типов указателей, доступность областей памяти для чтения/записи и др. Многие же высокоуровневые логические средства спецификации состояния памяти и поведения программ, такие как сепарационная логика, разрешения (permissions) или поддержка наследования поведения (например, уточнения контрактов функций), непосредственно в язык ACSL не включены. Некоторые из них иногда могут быть выражены с использованием возможностей самого ACSL. При этом уровень поддержки языка ACSL со стороны инструментов верификации, реализованных на основе платформы FRAMA-C может различаться.

Рассмотрим пример контрактной спецификации на языке ACSL. В листинге 1 приведён код функции `strnchr` из ядра Linux. Функция `strnchr`

осуществляет поиск символа c в строке `s`, которая ограничена длиной `cnt`. В спецификации функции требуется, чтобы указатель на строку `s` адресовал валидный участок памяти, размером `min(strlen(s), cnt) + 1`. Это условие является предусловием функции и указано в первой строке спецификации. Функция `strnchr` является чистой, то есть не имеет побочных эффектов, что описывается второй строкой спецификации.

```
1  /*@ requires valid_strn(s, count);
2      assigns \nothing;
3      behavior exists:
4        assumes ∃char *p;
5          s ≤ p < s + strnlen(s, count) ∧ *p ≡ (char %) c;
6        ensures s ≤ \result ≤ s + strnlen(s, count);
7        ensures *\result ≡ (char %) c;
8        ensures ∀ char *p; s ≤ p < \result ⇒ *p ≢ (char %) c;
9      behavior not_exists:
10       assumes ∀ char *p;
11         s ≤ p < s + strnlen(s, count) ⇒ *p ≢ (char %) c;
12       ensures \result ≡ \null;
13     complete behaviors;
14     disjoint behaviors;*/
15 char *strnchr(const char *s, size_t count, int c) {
16  //@ ghost char *os = s;
17  //@ ghost size_t ocount = count;
18  /*@ loop invariant 0 ≤ count ≤ ocount;
19      loop invariant os ≤ s ≤ os + strnlen(os, ocount);
20      loop invariant s - os ≡ ocount - count;
21      loop invariant valid_strn(s, count);
22      loop invariant strnlen(os, ocount) ≡ s - os + strnlen(s, count);
23      loop invariant ∀ char *p; os ≤ p < s ⇒ *p ≢ (char %) c;
24      loop variant count;
25   */
26  for (; count-- /*@%*/ && *s != '\0'; ++s)
27   if (*s == (char)/*@%*/c)
28    return (char *)s;
29  return NULL;
30 }
```

Листинг 1. Функция `strnchr`. Ядро Linux 4.12, файл lib/string.c
Listing 1. Linux kernel 4.12, file файл lib/string.c

Далее спецификация подразбивается на два случая. Первый — когда в строке существует искомый символ, второй — когда он отсутствует. Для разбиения спецификаций поведения функции на несколько различных случаев в языке ACSL есть соответствующее средство — поведения (behaviors). В отличие от некоторых языков спецификаций, где поведения вводятся поверх основного языка спецификаций как синтаксическое расширение (например, в JML), в ACSL поведения являются базовой сущностью языка спецификаций и

практически все спецификации, как в контракте, так и в теле функции могут быть отнесены к одному или нескольким ее поведениям. Для доказательства постусловий с помощью инструмента дедуктивной верификации сформулированы инварианты (loop invariants) на внутренний цикл функции и оценочная функция (loop variant) для него.

Реализация функции `strnchr` содержит в себе в явном виде приведение типа с потерей старшей части значения в строке 27, а также случай с целочисленным переполнением в итераторе цикла из-за постфиксного декремента (строка 26).

Табл. 1. Использование спецификационных конструкций
Table 1. Use of specification structures

Функция	Wrap-around +	Wrap-around Cast
_parse_integer.		
check_bytes8		
kstrtobool		
memchr	✓	✓
memcmp		
memcpy	✓	
memmove	✓	
memscan		
memset	✓	✓
skip_spaces		
strcasecmp		
strcat		
strchr	✓	✓
strchrnul	✓	✓
strcmp		✓
strncmp	✓	
strcpy		
strcspn		
strlcpy		
strlen		
strnchr	✓	✓
strnlen	✓	
strpbrk		
strrchr		✓
strsep		
strspn		

Чтобы указать инструментам верификации, что приведение переменной с типа `int` к типу `char` с потерей части значения является намеренным поведением, используется специальная конструкция `/*@%*/`. Ровно такая же спецификация

используется для обозначения переполнения переменной `cnt`. Семантика этих спецификаций обсуждается в следующем разделе.

В таблице 1 отмечено, в каких функциях из числа верифицированных в рамках данной работы, встречались подобные ситуации с переполнением беззнакового итератора цикла (Wrap-around +) и явным приведением типа с потерей части значения (Wrap-around Cast).

Рассмотрим теперь некоторые проблемы, связанные с поддержкой семантики некоторых конструкций языка ACSL в инструментах дедуктивной верификации на примере подходов к моделированию указателей и машинных целых в инструменте верификации JESSIE, которые были унаследованы инструментом ASTRAVER.

5.Проблемы инструментов верификации

5.1 Блочно-байтовая модель памяти JESSIE

Существует по крайней мере несколько способов логического представления указателей и выделенных блоков памяти в генерируемых условиях верификации. В JESSIE реализована так называемая *блочно-байтовая модель памяти* (*byte-level block memory model*), в которой указатели представлены логически как пары вида (l, o), а блоки памяти — как тройки вида (l, a, s). Здесь метка l уникально идентифицирует блок памяти, o обозначает смещение указателя относительно начального адреса a блока l, а s соответствует размеру блока. Использование уникальных меток блоков позволяет проверять, что доступ к памяти за пределами выделенного блока не происходит даже в том случае, если адресуемая соответствующим указателем область памяти также является выделенной. Несмотря на то, что такой доступ не нарушает сегментирование памяти (и, соответственно, не приводит к ошибкам времени выполнения), соответствующее поведение не допускается стандартом языка Си [15]. (секция 6.5.6, абзац 8 классифицирует создание указателей за пределы выделенных блоков как неопределенное поведение, кроме указателей на область памяти, непосредственно следующую за последним элементом массива). Как объяснено в кандидатской работе [10], описывающей теоретическую основу и архитектуру инструмента JESSIE, блочно-байтовая модель памяти в принципе позволяет выражать семантику часто используемых на практике приемов программирования на языке Си, выходящих за рамки стандарта, таких как реализация функции `memmove`, сохраняя при этом возможность обнаруживать ошибки управления памятью, такие как доступ после освобождения (use-after-free), а также возможные переполнения указателей.

Реализация модели памяти в инструменте JESSIE, однако, имеет ряд отличий от соответствующего относительно простого теоретического описания и налагает дополнительные ограничения на поддерживаемое подмножество языка Си.

Во-первых, указатели реализованы в соответствующей теории JESSIE (на языке WhyML) как значения абстрактного типа $pointer$ с четырьмя соответствующими абстрактными операциями:

$$sub\_pointer : pointer \times pointer \to int,$$
$$shift : pointer \times int \to pointer,$$
$$same\_block : pointer \times pointer \to bool \text{ и}$$
$$address : pointer \to int.$$

Размеры выделенных блоков памяти представлены в теории JESSIE неявно с помощью так называемых *таблиц аллокации* (*allocation tables*), индексированных состоянием программы (то есть изменяемых) значений абстрактного типа $alloc\_table$ с двумя аксиоматически заданными функциями:

$$offset\_min : alloc\_table \times pointer \to int \text{ и}$$
$$offset\_max : alloc\_table \times pointer \to int.$$

Эти функции выражают минимально и максимально допустимое смещение указателя, не выводящее его за пределы соответствующего выделенного блока памяти. Для уникальных меток выделенных блоков и их начальных адресов явное представление в теории JESSIE также отсутствует. Условия верификации, генерируемые инструментом для операций динамического выделения и освобождения памяти (вызовы функций `kmalloc` и `kfree` обрабатываются в JESSIE специальным образом), упоминают только таблицы аллокации и функции $sub\_pointer$, $shift$ и $same\_block$. Это делает соответствующую аксиоматизацию заведомо неполной. В частности, функция $address$ не только никак не упоминается в текущей аксиоматизации теории JESSIE, но и в принципе не может иметь в текущей реализации инструмента достаточно полную аксиоматизацию. В частности, рассмотрим следующее свойство этой функции: "*два валидных указателя, адресующие объекты из различных выделенных блоков памяти не могут иметь одинаковый адрес*". Это свойство не может быть выражено в виде логического утверждения в текущей теории JESSIE, потому что его формализация требует использования условия существования элемента во множестве всех *достижимых* состояний соответствующей таблицы аллокации:

$$\forall p_1, p_2. \big(\exists s_a \in Reachable(s_a). offset\_min(s_a, p_1) \le 0 \land offset\_max(s_a, p_1) \ge 0 \land offset\_min(s_a, p_2) \le 0 \land offset\_max(s_a, p_2) \ge 0 \land \neg same\_block(p_1, p_2)\big) \Rightarrow address(p_1) \neq address(p_2).$$

Так как задача получения явного представления предиката $Reachable(s_a)$ в общем случае является алгоритмически неразрешимой, инструмент может использовать неявное представление, например с помощью выражения соответствующих свойств адреса в каждой точке выделения памяти:

$$\forall p. offset\_min(s_a^*, p) \le 0 \land offset\_max(s_a^*, p) \ge 0 \Rightarrow address(p) \neq address(p^*).$$

Здесь p^* — адрес начала выделяемого блока памяти, s_a^* — состояние таблицы аллокации в точке выделения памяти. Невозможность точной формализации функции $address$ не позволяет генерировать соответствующие условия верификации для обнаружения возможных переполнений указателей и использовать более гибкую формализацию операций сравнения и вычитания указателей (для верификации таких функций, как `memmove`).

Кроме этого, в теории JESSIE смещение и разность указателей, используемые функциями $shift$ и $sub\_pointer$, измеряются в единицах, равных размеру типов адресуемых указателями значений (в соответствии с семантикой адресной арифметики в языке Си), а не в байтах или машинных словах. В частности, выражение `p + 1`, где `p` имеет тип `int *`, транслируется как $shift(p, 1)$, а не как $shift(p, s_{int})$, где s_{int} — константа, равна размеру типа `int` (обычно, 4 байта). Такая трансляция изначально не позволяет выражать многие широко распространенные сочетания перетипирования указателей с адресной арифметикой, включая использования макроса `container_of` (из ядра ОС Linux). Это нетрудно увидеть, рассмотрев два указателя: `p + 1` и `((char *) p) + 1`, где `p` имеет тип `int *` и указывает на начало некоторого выделенного блока памяти. В блочно-байтовой модели памяти со смещениями, измеряемыми в размерах типов, эти указатели имеют одинаковое представление $(l, 1)$, хотя их реальные адреса не могут быть равны (они должны различаться хотя бы на 1, обычно на 3). Это противоречит функциональной консистентности функции $address$. В том числе для того, чтобы не допустить возникновение подобного противоречия (но в основном, по другим причинам, см. [15, 16]) в текущей реализации JESSIE используется два отдельных приема. Во-первых, вводятся специальные дополнительные (логические) *таблицы тегов*, содержащие точные динамические типы объектов в выделенной памяти. Эти таблицы позволяют включать в условия верификации необходимые проверки, ограничивающие использование адресной арифметики (более подробно об этом в [17, 3]). Во-вторых, применяются несколько *нормализующих* преобразований кода, которые трансформируют вложенные структуры и адресуемые поля простых типов в указатели на отдельно выделенные структуры или значения соответствующих типов (эти трансформации описаны в [16]). Это позволяет адресовать вложенные объекты в модели памяти JESSIE. Однако сочетание двух этих приемов приводит к возникновению ряда других существенных ограничений. В частности, объединения, содержащие вложенные структуры в качестве своих полей, не могут быть представлены в модели памяти инструмента. Это связано с невозможностью точного статического выделения среди указателей, получаемых, например, как параметры функции, указателей на структуры, вложенные в объединения. Запись в поля таких структур в соответствии с моделью JESSIE должна транслироваться в *сильные обновления* (*strong coercions*) [17] соответствующих объемлющих объединений с возможным обновлением таблиц тегов и логического представления других

интерпретаций (перетипирований) соответствующей памяти (соответствующей другим полям объединения).

Для преодоления этих и других ограничений текущей модели памяти JESSIE в [4] была предложена новая модель памяти. Эта модель, однако, предполагает использование простой байтовой модели указателей (без привязки их к соответствующим выделенным блокам). Но в силу предполагаемой обычно произвольности стратегии выделения памяти такое моделирование на практике не должно приводить к пропуску случаев нарушения ограничений, налагаемых стандартом языка Си, в случаях попыток разыменования валидной памяти из других выделенных блоков памяти. В таких случаях обычно по крайней мере один из возможных вариантов выделения памяти приводит к разыменованию невалидного указателя и, таким образом, не остается возможности доказать корректность соответствующего разыменования в общем случае. Модель памяти, предложенная в [4], однако, еще не была реализована в инструменте верификации. Поэтому в данной работе была использована существующая реализация модели памяти JESSIE.

Единственное существенное изменение, сделанное в ходе работы в инструменте верификации, было связано с трансляцией неравенств указателей. Так как существующая реализация модели памяти не обеспечивает достаточную выразительность для представления семантики произвольных операций сравнения указателей, мы ограничили поддержку сравнения указателей, разрешив сравнение лишь между указателями на объекты одного выделенного блока памяти, добавив генерацию соответствующих условий верификации и изменив трансляцию соответствующих предикатов вида $p_1 \diamond p_2$ на $sub\_pointer(p_1, p_2) \diamond 0 \wedge same\_block(p_1, p_2)$. Это позволило сократить многие спецификации, так как часто встречающееся дополнительное условие $same\_block(p_1, p_2)$ стало задаваться неявно для всех операций сравнения указателей.

5.2 Модели целых чисел, комбинированная модель целых чисел и аннотации для модульной арифметики

Изначально в JESSIE были реализованы три логические модели машинных целых различного размера и знаковости. Наиболее простая модель, называющаяся `math` (или моделью *неограниченных* целых), предполагает представление всех машинных целых как математических целых. Эта модель не поддерживает проверку возможных арифметических переполнений и не моделирует модульную арифметику ни для каких типов машинных целых (по стандарту языка Си модульная арифметика определена только для беззнаковых целых). В этой модели в принципе возможно моделирование некоторых побитовых операций над неограниченными целыми с помощью соответствующей аксиоматизации, но на практике такое моделирование обычно не эффективно. Другая, наиболее часто используемая модель целых

называется `defensive` (или моделью *ограниченных* целых) и имеет два отличия от модели `math`:

- для целочисленных операций в коде в этой модели генерируются соответствующие условия верификации, предотвращающие возможные арифметические переполнения;
- ограниченные целые в спецификациях моделируются абстрактными типами со специальными функциями отображения (такими как $int32\_of\_integer$ и $integer\_of\_int32$), аксиоматизация которых определяет результат преобразования в ограниченное целое только значений, попадающих в соответствующий диапазон.

Модель `defensive` проста и эффективна и подходит для большинства случаев, за исключением тех, когда требуется точное моделирование машинной арифметики или побитовых операций. Для этих целей в JESSIE реализована модель целых `modulo`, которая предполагает точное моделирование значений машинных целых как битовых векторов.

К сожалению, в JESSIE модель целых чисел может быть выбрана лишь один раз для всей верифицируемой программы с использованием соответствующей прагмы. На практике, однако, желательно иметь возможность выбирать подходящий вариант моделирования семантики операций над целыми для каждого отдельного участка кода, вплоть до отдельно взятых операций. Рассмотрим следующий пример (см. Листинг 2):

```
1  int strncasecmp(const char *s1,
2     const char *s2, size_t len) {
3   unsigned char c1, c2;
4   if (!len) return 0;
5   do {
6     c1 = *s1++;
7     c2 = *s2++;
8     if (!c1 || !c2) break;
9     if (c1 == c2) continue;
10    c1 = tolower(c1);
11    c2 = tolower(c2);
     12    if (c1 != c2) break;
13  } while (--len);
14  return (int)c1 - (int)c2;
15 }
```

Листинг 2. Функция strncasecmp. Ядро Linux 4.12, файл lib/string.c
Listing 2. Function strncasecmp. Linux kernel 4.12, file lib/string.c

Здесь в строках 6 и 7 уместно применение модели машинных целых modulo, так как приведение типа `char` к типу `unsigned char` может привести к переполнению, и в данном случае такое поведение соответствует намерениям программиста. Однако желательно также, чтобы возможное переполнение было обнаружено в случае замены типа возвращаемого значения функции на

`char`. Поэтому для моделирования вычитания в строке 13 уместно применение модели `defensive`.

Для поддержки такой точной спецификации модели целых на уровне отдельных арифметических операций мы реализовали в инструменте поддержку расширения языка ACSL *аннотациями для модульной арифметики*. Были введены следующие новые виды аннотаций:

- для арифметических операций: `+/*@%*/`, `-/*@%*/`, `*/*@%*/`, ...
- для составных присваиваний (compound assignments): `+=/*@%*/`, `-=/*@%*/`, `/=/*@%*/` ...
- для префиксных и постфиксных операторов: `++/*@%*/`, `--/*@%*/`;
- для явных приведений типа: `(unsigned char)/*@%*/`, ...
- для модульной арифметики в спецификациях: `+%`, `-%`, `*%`, ...

Модель машинных целых, используемая для моделирования как обычных арифметических операций (без аннотаций), так и аннотированных операций модульной арифметики является комбинированной, в которой машинные целые моделируются как битовые векторы с аксиоматически заданными функциями отображения в/из математических целых. Использование такого комбинированного моделирования целых и аннотаций для модульной арифметики позволило значительно упростить спецификацию и верификацию многих функций в рамках данной работы.

6. Формальные спецификации

При разработке спецификаций мы руководствовались несколькими приёмами: использованием избыточных спецификаций (явные спецификации и спецификации, устанавливающие соответствие логической функции), разработкой спецификаций на основе кода реализации и контекста

```
/*@ requires cnt ≥ 0;
requires \valid(s+(0..cnt-1));
assigns \nothing;
behavior bigger:
 assumes ∀ ℤ i; 0 ≤ i < cnt ⇒ s[i] ≢ 0;
 ensures \result ≡ cnt;
behavior smaller:
 assumes ∃ ℤ i; 0 ≤ i < cnt ∧ s[i] ≡ 0;
 ensures \result ≤ cnt;
complete behaviors;
disjoint behaviors;*/
size_t strnlen(const char *s, size_t cnt);
```

использования функции.

Листинг 3. Контракт для функции `strnlen`. Проект по верификации klibc
Listing 3. Contract for function `strnlen`. Verification project klibc

Результаты, описанные в работе [6], показывают, что разработать полный контракт функции, отталкиваясь от документации крайне затруднительно: почти всегда, при доказательстве приходится переписывать спецификацию «от реализации». Подобный подход также объясняется тем, что в данной работе спецификации разрабатываются к готовому коду, а не код пишется в соответствии с некоторым набором данных спецификаций. Также в ядре для многих функций отсутствует специальная документация в коде. Мы намеренно не стали руководствоваться стандартной документацией к подобным функциям, так как реализация их в ядре может отличаться от других (например, от реализации в стандартной библиотеке), а документация быть не полной и содержать неточности [6].

```
/*@ predicate valid_strn(char *s, size_t cnt) =
 (∃ size_t n;
 (n < cnt) ∧ s[n] ≡ '\0' ∧ \valid(s+(0..n))) ∨
 \valid(s+(0..cnt));

requires valid_strn(s, cnt);
assigns \nothing;
ensures \result ≡ strnlen(s, cnt);
behavior null_byte:
 assumes ∃ ℤ i; 0 ≤ i ≤ cnt ∧ s[i] ≡ '\0';
 ensures s[\result] ≡ '\0';
 ensures ∀ ℤ i; 0 ≤ i < \result ⇒ s[i] ≢ '\0';
behavior cnt_len:
 assumes ∀ ℤ i; 0 ≤ i ≤ cnt ⇒ s[i] ≢ '\0';
 ensures \result ≡ cnt;
complete behaviors;
disjoint behaviors;*/
size_t strnlen(const char *s, size_t cnt);
```

Листинг 4. Контракт для функции `strnlen`*. Проект по верификации библиотечных функций ядра Linux*

Listing 4. Conntract for function `strnlen`*. The Linux library functions verification project*

Как следствие подобного подхода, спецификации к ряду функций обладают несколько более детализированным видом: так для функции вида `strn*` (см. Листинги 4, 6) на уровне спецификаций мы не требуем обязательного наличия маркера конца строки. В случае `strnlen` (см. Листинг 4) в предусловии предполагается что строка должна быть валидной до минимума из длины строки (если существует маркер её конца) и второго аргумента функции `strnlen`, а возвращаемый результат в постусловии задается явным образом. В случае `strncmp` (см. Листинг 6) также не вводятся ограничения на то, что входные строки должны содержать нулевой байт. Это приводит к тому, что на уровне спецификаций приходится явно описывать поведение функции, когда входные строки, имеющие конец, различаются по длине. Мы старались максимально ослабить предусловия и усилить постусловия для того, чтобы

протестировать инструменты дедуктивной верификации, выразительность языка ACSL, а также возможности солверов.

```
/*@ predicate valid_string{L}(char *s) =
 0 ≤ strlen(s) ∧ \valid_range(s,0,strlen(s));

requires valid_string(s1);
requires valid_string(s2);
requires n < INT_MAX;
assigns \nothing;
ensures n ≡ 0 ⇒ \result ≡ 0;
ensures (n > 0 ⇒ (∀ ℤ i;
 0 ≤ i ≤ minimum(n-1, minimum(strlen(s1), strlen(s2))) ∧
 s1[i] ≡ s2[i])) ⇒ \result ≡ 0;
ensures \result < 0 ⇒ (n > 0 ∧ ∃ ℤ i;
 0 ≤ i ≤ minimum(n-1, minimum(strlen(s1), strlen(s2))) ∧
 s1[i] < s2[i] ⇒
 (∀ ℤ k; 0 ≤ k < i ⇒ s1[k] ≡ s2[k]));
ensures \result > 0 ⇒ (n > 0 ∧ ∃ ℤ i;
 0 ≤ i ≤ minimum(n-1, minimum(strlen(s1), strlen(s2))) ∧
 s1[i] > s2[i] ⇒
 (∀ ℤ k; 0 ≤ k < i ⇒ s1[k] ≡ s2[k]));*/
int strncmp(const char *s1, const char *s2, size_t n);
```

Листинг 5. Контракт для функции `strncmp`*. Проект по разработке спецификаций для OpenBSD*

Listing 5. Contract for function `strncmp`*. The project to develop specifications for OpenBSD*

6.1 Логические функции

Для некоторых функций спецификации намеренно избыточны и фактически дважды по-разному описывают то, как функция должна работать. Например, одна из таких функций — `strlen`. В её спецификации есть обычные функциональные требования и есть требование на соответствие возвращаемого результата вызову логической функции под тем же названием `strlen`. Подобный подход мотивирован тем, что логическую функцию `strlen` удобно использовать в спецификациях других функций, например, `strcmp` (а логическую функцию, описывающую поведение функции `strcmp` — при описании функциональных требований к `strcpy`). При этом все основные свойства логической функции задаются с помощью аксиом и лемм (леммы на данном этапе не доказывались). Однако такие спецификации не во всех случаях удобны. Например, их в общем случае невозможно отобразить в проверки времени исполнения E-ACSL [18]. Поэтому для тех функций, которым в спецификациях ставилась в соответствие логическая функция, обязательно разрабатывались и «обычные» спецификации.

Функции на языке Си можно сопоставить логическую функцию (один-в-один), только в случае, если Си-функция «чистая» (pure). Логическую функцию рационально писать в том случае, если она пригодится для

разработки спецификаций для других функций. Например, в функциональных требованиях к `memcpy` можно выразить «одинаковость» `src` и `dest` посредством вызова логической функции `memcmp`.

```
/*@ requires valid_strn(cs, cnt);
requires valid_strn(ct, cnt);
assigns \nothing;
ensures \result ≡ -1 ∨ \result ≡ 0 ∨ \result ≡ 1;
behavior equal:
 assumes cnt ≡ 0 ∨
  cnt > 0 ∧
  (∀ ℤ i; 0 ≤ i < strnlen(cs, cnt) ⇒ (cs[i] ≡ ct[i])) ∧
  strnlen(cs, cnt) ≡ strnlen(ct, cnt);
 ensures \result ≡ 0;
behavior len_diff:
 assumes cnt > 0;
 assumes ∀ ℤ i; 0 ≤ i < strnlen(cs, cnt) ⇒ (cs[i] ≡ ct[i]);
 assumes strnlen(cs, cnt) ≢ strnlen(ct, cnt);
 ensures strnlen(cs, cnt) < strnlen(ct, cnt) ⇒ \result ≡ -1;
 ensures strnlen(cs, cnt) > strnlen(ct, cnt) ⇒ \result ≡ 1;
behavior not_equal:
 assumes cnt > 0;
 assumes ∃ ℤ i; 0 ≤ i < strnlen(cs, cnt) ∧ cs[i] != ct[i];
 ensures ∃ ℤ i; 0 ≤ i < strnlen(cs, cnt) ∧
  (∀ ℤ j; 0 ≤ j < i ⇒ cs[j] ≡ ct[j]) ∧
  (cs[i] ≢ ct[i]) ∧
  ((u8 *)cs[i] < (u8 *)ct[i] ? \result ≡ -1 : \result ≡ 1);
complete behaviors;
disjoint behaviors;*/
int strncmp(const char *cs, const char *ct, size_t cnt);
```

Листинг 6. Контракт для функции `strncmp`*. Проект по верификации библиотечных функций ядра Linux*

Listing 6. Conntract for function `strncmp`*. The Linux library functions verification project*

7. Нерешённые проблемы

На уровне спецификаций авторы столкнулись с рядом проблем, связанных с значительными неточностями моделирования операций с указателями, а также недостаточным уровнем поддержки языка ACSL инструментами.

Так, для функции `memmove` инструментами верификации генерируется условие верификации, которое проверяет, что указатели `dest` и `src` лежат в одном выделенном блоке памяти. Это необходимо для того, чтобы результат их сравнения был определён по стандарту [14]. Напомним, как работает функция `memmove`: она осуществляет копирование участка памяти размером *n* байт с адреса `src` по адресу `dst`, при условии, что два обозначенных региона памяти могут как пересекаться, так и не пересекаться. Чтобы реализовать последнее условие, в функции осуществляется порядковое сравнение

указателей `dest` и `src`. В том случае, если `dest` находится до `src`, осуществляется побайтовое копирование с начала `src` (таким образом, если регионы накладываются друг на друга, то в `src` будет затираться часть, ранее уже скопированная в `dest`); если же `dest` находится после `src`, то осуществляется копирование, начиная с конца региона памяти `src`.

Модель памяти, лежащая в основе инструментов верификации, позволяет осуществлять арифметические операции на указателях (в `memmove` это сравнение, реализованное через разницу между указателями) в том случае, если указатели принадлежат одному выделенному блоку памяти. Для `memmove` это не обязательно так. Если условие возможной разницы регионов записано в контракте, то соответствующее условие верификации, требующее совпадения блоков памяти, невозможно доказать. Это отражено в итоговых результатах в таблице 2.

Функция `strcat` осуществляет конкатенацию двух строк, добавляя строку `src` к строке `dest`. Для этого сначала находится конец строки `dest`, а после осуществляется копирование строк аналогично тому, как это происходит в функции `strcpy`. Для того, чтобы доказать условия верификации, проверяющие корректность работы с памятью в данной функции, было достаточно потребовать валидности строк `src` и `dest`, а также достаточного количества памяти за концом строки `dest` для того, чтобы вместить содержимое `src`. Однако при доказательстве корректности относительно функциональных требований к данной функции, потребовалось сформулировать дополнительное требование, утверждающее, что сумма длин строк `src` и `dest` вмещается в тип `size_t`. Функция реализована через итерацию по указателям. Следовательно, возможность доказательства корректности работы с памятью без привлечения дополнительного требования в данной функции означает, что в модели памяти инструмента верификации не учитывается возможность переполнения указателей.

```
void *memset(void *s, int c,
             size_t count) {
  char *xs = s;
  while (count--)
-   *xs++ = c;
+   *xs++ = (char) c;
  return s;
}
```

Листинг 7. Функция `memset`. Ядро Linux 4.12, файл lib/string.c
Listing 7. Function `memset`. Linux kernel 4.12, file lib/string.c

Несколько функций потребовали изменения кода для того, чтобы стало возможным доказать их корректность. Несмотря на то, что авторы ставили своей целью свести к минимуму внесение правок в код, в двух случаях этого избежать не удалось. В функциях `memset` и `strcmp` используется неявное приведение типов с переполнением. В `memset` тип `int` неявным образом

приводится к типу `char` (листинг 3), а в `strcmp` — `char` к `unsigned char`. Для того, чтобы добавить спецификацию о намеренном переполнении, требуется сделать приведение типа явным. В инструментах на уровне спецификаций не хватает конструкции неявного приведения типа с переполнением (например, `/*@ (unsigned int %) */`) для того, чтобы

```
axiomatic NotSupported {
 predicate less(ℤ a, ℤ b) = a < b;
 logic ℤ min(ℤ a, ℤ b) = less(a, b) ? a : b;
 lemma not_supported:
  ∀ ℤ a, b; less(a, b) ? min(a, b) ≡ a : min(a, b) ≡ b;
}

axiomatic Supported {
 predicate less(ℤ a, ℤ b) = a < b;
 logic ℤ min(ℤ a, ℤ b);
 lemma defn1:
  ∀ ℤ a, b; less(a, b) ⇔ min(a, b) ≡ a;
 lemma defn2:
  ∀ ℤ a, b; !less(a, b) ⇔ min(a, b) ≡ b;
}
```

стало возможным обойтись без изменения кода в данных случаях.

Листинг 8. Аксиоматические теории
Listing 8. Axiomatic theories

На уровне спецификаций инструментами не поддерживается использование предикатов в определениях логических функций, а также использование предикатов в леммах и аксиомах в тернарном операторе. Из-за этого иногда сложно дать явное определение логической функции и приходится использовать аксиоматическое определение. Эта особенность мешает дать явное определение логических функций `skip_spaces`, `strcspn`, `strpbrk` и `strspn`.

Функции из файла `ctype.h` (`isspace`, `isdigit`, `isalnum`, `isgraph`, `islower`, ...) определены как макросы, которые оперируют на массиве из 256 ячеек `_ctype`, в котором задаётся принадлежность каждого символа определённому классу. Чтобы упростить верификационную задачу данные макросы были заменены inline-функциями: инструменты верификации не позволяют писать спецификации на макроопределения, только на функции. Из-за того, что глобальная инициализация массивов не транслируется в модель для верификации (WhyML), массив `_ctype` был переопределён как строка (инициализация строк транслируется в модельные аксиомы). Однако доказать соответствие функций из файла `ctype.h` их спецификаций не удалось и после описанных трансформаций: солверы не справляются с доказательством, когда в модели есть аксиоматическое задание массива `_ctype` длиной 256 символов.

8. Результаты

В процессе доказательства полной корректности функций мы пользовались преимущественно солверами Alt-Ergo (1.30) и CVC4 (1.4). Их возможностей хватило, чтобы доказать все условия верификации. Для того, чтобы протестировать возможности других солверов в разных конфигурациях, нами был подготовлен тестовый стенд и специальная стратегия преобразования для каждого условия верификации.

8.1 Конфигурация тестового стенда

Солверы запускались на машине со следующей конфигурацией: AMD FX-8120 (Eight-Core Processor), 16GB RAM. С лимитами по времени в 60 секунд и по памяти в 6000 Mb. Параллельно работало не более трёх солверов. Операционная система — GNU/Linux (kernel: 4.12.12 (smp preempt) x86_64). Использовались инструменты следующих версий с сайта проекта AstraVer: WHY3 (0.87.3+git), FRAMA-C (Silicon-20161101), JESSIE2 (alpha3)

8.2 Стратегия трансформации условий верификации

Для того, чтобы все солверы были поставлены в максимально близкие условия и были доказаны все условия верификации (кроме одного для `memmove`), использовалась следующая стратегия трансформации условий верификации:

1. подразбить условие верификации по конъюнктам (split_goal_wp); перейти к шагу 2.
2. попробовать применить трансформацию из шага 1, иначе перейти к шагу 3;
3. встроить определения всех логических символов (inline_all);
4. попробовать применить трансформацию из шага 1, иначе перейти к шагу 5;
5. сколемизация условия верификации (introduce_premises).

Необходимо отметить, что в некоторых случаях трансформация inline_all наоборот затрудняет работу солверов. Это происходит в том случае, когда используется достаточно большое количество предикатов в спецификации функции, формируя длинную цепочку зависимостей. Однако разработанные нами спецификации не подходят под этот критерий, и эксперименты с запусками солверов показали обоснованность применения данной трансформации.

Эксперименты также показали, что решатели более эффективно разрешают выполнимость формул вида $f(\bar{x}) \wedge \neg g(\bar{x})$ (1), чем формул вида $\neg\forall\bar{x}.f(\bar{x}) \Rightarrow g(\bar{x})$ (2), несмотря на то, что переход от формул вида (2) к формулам вида (1) возможен с помощью простых преобразований, сохраняющих выполнимость (приведения к предваренной нормальной форме и сколемизации). Поэтому в

экспериментах использовалась трансформация WHY3 introduce_premises, соответствующая выполнению этих преобразований.

В остальном можно видеть, что приведённая стратегия максимально нацелена на подразбитие условия верификации на отдельные конъюнкты. Это облегчает работу солверов. При реальной работе с доказательствами подобная стратегия не используется. Некоторые из приведённых трансформаций используются лишь в том случае, когда солверы не могут сразу доказать условие верификации.

8.3 Результаты солверов

Результаты запуска солверов приведены в таблице 2. В запусках участвовали следующие солверы: ALT-ERGO (1.30), CVC3 (2.4.1), CVC4 (1.4), CVC4 (1.5), EPROVER (1.9.1-001), SPASS (3.9), Z3 (4.5.0).

Табл, 2. Запуски солверов
Table 2. Solver runs

Function	VC	Alt-Ergo 1.3		CVC3 2.4.1		CVC4 1.4		CVC4 1.5		Eprover 1.9.1-001		Spass 3.9		Z3 4.5.0	
	total	vc	atime	vc	atime	vc	atime	vc	atime	vc	atime	vc	atime	vc	atime
_parse_integ.	282	276	0.1	280	0.83	✓	0.18	✓	0.1	212	0.24	197	1.69	279	0.06
check_bytes8	50	49	0.55	49	0.09	49	0.09	✓	0.11	38	1.76	31	8.38	36	1.52
kstrtobool	1096	✓	0.05	✓	0.08	✓	0.1	✓	0.09	1006	0.13	937	0.38	1065	0.15
memchr	39	✓	6.05	11	0.22	✓	0.37	✓	0.15	31	2.58	11	5.73	29	0.12
memcmp	60	58	0.13	✓	0.15	58	0.1	✓	0.1	49	0.51	36	4.45	55	0.15
memcpy	43	✓	4.18	✓	0.35	✓	0.16	✓	0.14	30	1.05	16	6.85	30	0.06
memmove	**93***(92)	90	3.94	✓	0.88	87	0.16	✓	0.18	63	0.95	43	11.87	68	0.3
memscan	47	46	0.07	✓	0.1	✓	0.09	✓	0.09	41	0.59	34	4.55	42	0.06
memset	27	26	5.02	14	0.19	✓	0.19	✓	0.16	19	3.82	12	11.12	18	0.08
skip_spaces	34	30	0.76	32	1.96	✓	0.51	33	0.14	27	0.7	24	0.34	30	0.09
strcasecmp	58	50	0.43	52	1.65	57	0.79	✓	0.53	43	0.28	35	2.85	49	0.49
strcat	73	68	0.58	66	2.16	✓	1.13	71	0.17	54	2.56	39	0.67	60	0.94
strchr	43	35	4.57	23	0.17	✓	0.23	✓	0.22	31	1.03	24	3.65	32	0.11
strchrnul	46	42	2.07	37	0.26	✓	0.19	✓	0.16	40	1.91	31	2.27	39	0.31
strcmp	60	51	1.76	16	0.6	✓	1.75	59	1.08	47	1.05	36	1.65	47	0.1
strcpy	46	43	1.33	45	0.66	✓	0.48	✓	0.17	33	1.13	26	0.65	39	1.43
strcspn	78	68	0.38	69	0.37	74	2.95	75	1.82	58	1.85	46	1.68	61	0.11
strlcpy	84	82	0.15	82	0.14	✓	1.08	✓	0.24	67	1.2	52	1.74	78	0.42
strlen	26	✓	1.12	24	0.12	✓	0.16	✓	0.23	19	3.36	14	2.96	21	0.08
strnchr	49	38	4.44	19	0.23	46	3.34	✓	0.72	35	2.57	24	1.56	27	0.09
strncmp	102	81	2.57	25	0.25	94	2.39	99	2.32	76	1.06	55	2.56	76	0.57
strnlen	44	39	1.91	42	1.04	39	1.23	✓	1.31	31	2.4	26	5.52	32	0.08
strpbrk	70	57	0.64	58	1.54	62	3.18	67	1.57	48	1.89	39	0.75	53	0.09
strrchr	62	53	4.57	12	0.17	✓	1.09	60	0.85	46	2.33	31	4.67	46	0.11
strsep	62	60	0.25	60	0.09	✓	0.19	✓	0.15	55	0.12	51	1.48	58	0.06
strspn	107	99	0.84	100	0.69	104	1.32	103	0.61	89	1.37	75	1.59	91	0.13
TOTAL	2781	2645	0.9	2454	0.42	2740	0.61	2761	0.37	2288	0.76	1945	1.72	2461	0.22

В первой колонке таблица содержит имя функции ядра Linux, корректность которой доказывается. Во второй колонке содержится информация о

количестве условий верификации для указанной функции после применения описанной стратегии для тестирования солверов. В последующих колонках приводится информация о запусках солверов: количеству доказанных ими условий верификации и среднему времени работы на успешных запусках.

Там, где солвер сумел доказать все условия верификации для конкретной функции в таблице проставлена ✓. Солверы с максимальным количеством доказанных условий верификации помечаются зеленым цветом. Минимум среди остальных солверов выделяется красным цветом. Минимальное среднее время среди всех запусков отмечается синим цветом. Максимальное — жёлтым цветом.

Также солверы CVC4 (1.4, 1.5) и Z3 (4.5.0) дополнительно тестировались с драйвером noBV. Это специальный драйвер WHY3, который убирает теории для битовых векторов из задач солверов. Так как в нашем наборе функций отсутствуют побитовые операции и манипуляции с битовыми полями, предполагалось что использование данного драйвера себя оправдывает. Результаты данных конфигураций не вошли в итоговую таблицу, так как они оказались хуже по количеству доказанных условий верификации, чем другие конфигурации этих же солверов. При лимите по времени в 40 секунд и по памяти в 4000 Mb солверы CVC4 с драйвером noBV показывали результат лучше, однако при текущих лимитах картина радикально поменялась.

Необходимо отметить, что в нормальном цикле работ нами используются солверы ALT-ERGO (1.30), CVC4 (1.4), Z3 (4.5.0) в порядке частоты использования.

Солвер CVC4 версии 1.5 запускался с драйвером для версии солвера CVC4 1.4, так как в текущей версии WHY3 отсутствует драйвер для релизной версии солвера 1.5. Версия драйвера для пререлизной версии выдаёт результат хуже [19].

Табл. 3. Максимальное время работы солверов и случаи уникальных доказательств
Table 3. Maximum time of solvers and cases of unique proof

Alt-Ergo 1.3		CVC3 2.4.1		CVC4 1.4		CVC4 1.5		Eprover 1.9.1-001		Spass 3.9		Z3 4.5.0	
mtime	uniq	mtime	uniq	mtime	uniq	mtime	uniq	mtime	uniq	mtime	uniq	mtime	uniq
58.75	1	56.68	0	57.97	7	52.27	20	47.8	0	59.74	0	26.74	0

8.4 Интерпретация результатов

Все условия верификации, за исключением одного для `memmove`, успешно доказываются солверами. Лучше всего себя показали ALT-ERGO и CVC4, что легко объяснимо тем, что инструменты верификации тестируются, в основном, на этих двух солверах.

Лучший результат по количеству доказанных условий верификации показывает CVC4 версии 1.5.

Лучшие результаты по времени показывает Z3. Это можно частично объяснить тем, что учитываются только успешные запуски солверов. Z3 доказывает меньше условий верификации, чем ALT-ERGO и CVC4.

Также стоит отметить, что механизм запуска солверов WHY3 иногда давал сбои и некорректно учитывал лимит по времени. Так, для солвера Spass в 4х случаях было существенно (в 1.5-2 раза) превышено максимальное время доказательства, для солвера CVC4 (1.5) было одно превышение на 18 секунд. Несмотря на то, что солверы успешно доказали условия верификации, эти времени и для других солверов.

В таблице 3 приведены данные по максимальному времени работы солвера для успешного доказательства, а также по количеству уникальных доказательств. результаты не были им засчитаны и не учитывались в итоговой таблице. В процессе подготовки результатов мы сталкивались с некорректным учётом

Под уникальными доказательствами понимаются условия верификации, которые смог доказать только один солвер.

В целом, результаты показали, что версия солвера CVC4 (1.5) заслуживает пристального внимания к себе, и, возможно, перехода к использованию его как основного, при работе с данным стеком инструментов. Пока этому мешает отсутствие полноценной поддержки со стороны WHY3 и общая нестабильность его работы. Солвер часто падает с ошибкой сегментирования на условиях верификации, на которых не применялись вышеописанные трансформации.

9. Дальнейшая работа

Для части логических функций, реализованных к текущему моменту возможно доказать корректность лемм. Из-за ограничений инструментов, не все логические функции могли быть определены явно. Для тех из них, которые не заданы явно, доказать корректность лемм на текущий момент невозможно по очевидной причине. Для остальных корректность лемм должна быть доказана с помощью инструмента Coq.

Следующим логичным шагом является расширение набора функций для формальной верификации и включения в этот набор не только функций для работы со строками, но и, например, с битовой арифметикой, а также иных типов функций из папки `lib` ядра Linux. На этом этапе наибольший интерес для нас представляют функции, так или иначе использующие конструкции языка Си, которые сложны для формальной верификации.

После того, как контракты функций корректно сформулированы, имеет смысл проверить соблюдение предусловий в точках вызовов библиотечных функций. Ввиду того, что кода ядра Linux имеет существенный объем, осуществить подобную проверку методами дедуктивной верификации практически невозможно. Однако, с помощью плагина E-ACSL возможно часть

предусловий отобразить в проверки времени исполнения и осуществлять проверку соблюдения предусловий библиотечных функций в динамике. Это потребует существенной доработки плагина E-ACSL для работы с кодовой базой ядра Linux, а также частичного изменения самого кода ядра. Плагин использует несколько библиотек (для отслеживания состояния памяти, для работы с неограниченными числами), которые должны быть интегрированы в код ядра.

10. Заключение

В рамках работы были полностью доказаны 26 библиотечных функций ядра Linux. Большинство данных функций оперируют с данными строкового типа. На наборе этих функций удалось выявить значительное количество недостатков в использованных инструментах верификации, выработать подходы к доказательству и разработке спецификаций для функций аналогичного вида, предложить и реализовать расширение набора спецификационных конструкций языка ACSL. Авторы старались разрабатывать спецификации таким образом, чтобы не менять исходный код функций. Введение в язык ACSL двух дополнительных конструкций позволило доказать 11 функций, не изменяя их кода.

Итоговое соотношение спецификаций и размера кода составляет примерно ~2.6, то есть примерно две с половиной строчки спецификации на одну строчку кода.

Код функций, спецификации, протоколы доказательств выложены в открытом доступе вместе с инструкциями по воспроизведению результата [7]. Спецификации с исходным кодом могут в дальнейшем служить тестовым набором для инструментов дедуктивной верификации и солверов.

Список литературы

[1]. MISRA C: 2012. Guidelines for the Use of the C Language in Critical Systems, ISBN 978-1-906400-10-1 (paperback), ISBN 978-1-906400-11-8 (PDF), March 2013.

[2]. AstraVer Toolset: инструменты дедуктивной верификации моделей и механизмов защиты ОС. ИСП РАН. Доступно по ссылке: http://linuxtesting.org/astraver, 15.10.2017.

[3]. Мандрыкин М.У., Хорошилов А.В. Высокоуровневая модель памяти промежуточного языка Jessie с поддержкой произвольного приведения типов указателей. Программирование, 2015, т. 41, №4, стр. 23–29.

[4]. Мандрыкин М.У., Хорошилов А.В. Анализ регионов для дедуктивной верификации Си-программ. Программирование. 2016, т. 42, № 5, стр. 3–29.

[5]. Carvalho N., Silva Sousa C., Pinto J.S., Tomb A. Formal verification of kLIBC with the WP frama-C plug-in. In: Badger, J.M., Rozier, K.Y. (eds.) NFM 2014. LNCS, vol. 8430, pp. 343–358. Springer, Heidelberg (2014)

[6]. Torlakcik M. Contracts in OpenBSD. MSc thesis. University College Dublin. April, 2010.

[7]. Efremov D., Mandrykin M. VerKer: Verification of Linux Kernel Library Functions. Доступно по ссылке: http://forge.ispras.ru/projects/verker, 15.10.2017.
[8]. Cuoq P., Kirchner F., Kosmatov N., Prevosto V., Signoles J., Yakobowski B. Frama-C: A Software Analysis Perspective. Proceedings of the 10th International Conference on Software Engineering and Formal Methods. SEFM'12. Thessaloniki, Greece. 2012. Lecture Notes in Computer Science, vol. 7504, pp. 233–247. Springer-Verlag. Berlin, Heidelberg. 2012.
[9]. Baudin P., Cuoq P., Filliâtre J., Marché C., Monate B., Moy Y., Prevosto V. ACSL: ANSI/ISO C Specification Language. Version 1.12. CEA LIST and INRIA. May, 2017. Доступно по ссылке: https://frama-c.com/download/acsl_1.12.pdf, 15.10.2017.
[10]. Moy Y. Automatic Modular Static Safety Checking for C Programs. PhD thesis. Université Paris-Sud. January, 2009.
[11]. Filliâtre J., Paskevich A. Why3: Where Programs Meet Provers. Proceedings of the 22Nd European Conference on Programming Languages and Systems. ESOP'13. Rome, Italy. LNCS, vol. 7792, pp. 125–128. Springer-Verlag.
[12]. Burghardt J., Clausecker R., Gerlach J., Pohl H. ACSL By Example: Towards a Verified C Standard Library. Version 15.1.1. Доступно по ссылке: https://github.com/fraunhoferfokus/acsl-by-example/raw/master/ACSL-by-Example.pdf, 15.10.2017.
[13]. Cok D., Blissard I., Robbins J. C Library annotations in ACSL for Frama-C: experience report. GrammaTech, Inc. March, 2017. Доступно по ссылке: http://annotationsforall.org/resources/links/GT-libc-experience-report.pdf, 15.10.2017.
[14]. Hatcliff J., Leavens G. T., Leino K. R. M., Müller P., Parkinson M. Behavioral interface specification languages. ACM Comput. Surv. vol. 44, issue 3, article 16, 58 p. June 2012.
[15]. ISO/IEC 9899: 2011: C11 standard for C programming language. JTC and ISO. April 7, 2016.
[16]. Hubert T., Marché C. Separation Analysis for Deductive Verification. Heap Analysis and Verification (HAV'07). Braga, Portugal, March 2007, pp. 81–93.
[17]. Moy Y. Union and Cast in Deductive Verification. Proceedings of the C/C++ Verification Workshop. Technical Report ICIS-R07015, pp. 1–16. Radboud University Nijmegen. July, 2007.
[18]. Signoles J. E-ACSL Executable ANSI/ISO C Specification Language. Version 1.12. CEA LIST. May, 2017. Доступно по ссылке: https://frama-c.com/download/e-acsl/e-acsl.pdf, 15.10.2017.
[19]. Marché C. [Frama-c-discuss] Frama-C/WP and CVC4 (version 1.5). Доступно по ссылке: https://lists.gforge.inria.fr/pipermail/frama-c-discuss/2017-August/005338.html, 15.10.2017.

Formal Verification of Linux Kernel Library Functions

[1]D.V. Efremov <defremov@hse.ru >
[2]M.U. Mandrykin <mandrykin@ispras.ru>
[1]NRU Higher School of Economics,
20 Myasnitskaya Ulitsa, Moscow, 101000, Russia
[2]Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

Abstract. The paper presents result of a study on deductive verification of 26 Linux kernel library functions with AstraVer toolset. The code includes primarily string-manipulating functions and is verified against contract specifications formalizing its functional correctness properties. The paper presents a brief review of the related earlier studies, discusses their results and indicates both the previous issues that were successfully solved in this study and the ones that remain and still prevent successful verification. The paper also presents several specification practices that were applied in the study, including some common specification patterns. The authors have successfully and fully proved functional correctness of 25 functions. The paper includes results of benchmarking 5 state-of-the-art SMT solvers on the resulting verification conditions.

Keywords: static analysis; formal verification; deductive verification; standard library.

DOI: 10.15514/ISPRAS-2017-29(6)-3

For citation: Efremov D.V., Mandrykin M.U. Formal Verification of Linux Kernel Library Functions. *Trudy ISP RAN/Proc. ISP RAS*, vol. 29, issue 6, 2017. pp. 49-76 (in Russian). DOI: 10.15514/ISPRAS-2017-29(6)-3

References

[1]. MISRA C: 2012. Guidelines for the Use of the C Language in Critical Systems, ISBN 978-1-906400-10-1 (paperback), ISBN 978-1-906400-11-8 (PDF), March 2013.

[2]. AstraVer Toolset: deductive verification of Linux kernel modules and security policy models. ISP RAS. Available at: http://linuxtesting.org/astraver, accessed 15.10.2017.S

[3]. Mandrykin M. U., Khoroshilov A. V. High-level memory model with low-level pointer cast support for Jessie intermediate language. Programming and Computer Software, 2015, vol. 41, no. 4, pp. 197–207. DOI: 10.1134/S0361768815040040

[4]. Mandrykin M. U., Khoroshilov A. V. Region analysis for deductive verification of C programs. Programming and Computer Software, 2016, vol. 42, no. 5, pp. 257–278. DOI: 10.1134/S0361768816050042

[5]. Carvalho N., Silva Sousa C., Pinto J.S., Tomb A. Formal verification of kLIBC with the WP frama-C plug-in. In: Badger, J.M., Rozier, K.Y. (eds.) NFM 2014. LNCS, vol. 8430, pp. 343–358. Springer, Heidelberg (2014).

[6]. Torlakcik M. Contracts in OpenBSD. MSc thesis. University College Dublin. April, 2010.

[7]. Efremov D., Mandrykin M. VerKer: Verification of Linux Kernel Library Functions. Available at: http://forge.ispras.ru/projects/verker, accessed 15.10.2017.

[8]. Cuoq P., Kirchner F., Kosmatov N., Prevosto V., Signoles J., Yakobowski B. Frama-C: A Software Analysis Perspective. Proceedings of the 10th International Conference on Software Engineering and Formal Methods. SEFM'12. Thessaloniki, Greece. 2012. Lecture Notes in Computer Science, vol. 7504, pp. 233–247. Springer-Verlag. Berlin, Heidelberg. 2012.

[9]. Baudin P., Cuoq P., Filliâtre J., Marché C., Monate B., Moy Y., Prevosto V. ACSL: ANSI/ISO C Specification Language. Version 1.12. CEA LIST and INRIA. May, 2017. Available at: https://frama-c.com/download/acsl_1.12.pdf, accessed 15.10.2017.

[10]. Moy Y. Automatic Modular Static Safety Checking for C Programs. PhD thesis. Université Paris-Sud. January, 2009.

[11]. Filliâtre J., Paskevich A. Why3: Where Programs Meet Provers. Proceedings of the 22Nd European Conference on Programming Languages and Systems. ESOP'13. Rome, Italy. LNCS, vol. 7792, pp. 125–128. Springer-Verlag.

[12]. Burghardt J., Clausecker R., Gerlach J., Pohl H. ACSL By Example: Towards a Verified C Standard Library. Version 15.1.1. Available at: https://github.com/fraunhoferfokus/acsl-by-example/raw/master/ACSL-by-Example.pdf, accessed 15.10.2017.

[13]. Cok D., Blissard I., Robbins J. C Library annotations in ACSL for Frama-C: experience report. GrammaTech, Inc. March, 2017. Available at: http://annotationsforall.org/resources/links/GT-libc-experience-report.pdf, accessed 15.10.2017.

[14]. Hatcliff J., Leavens G. T., Leino K. R. M., Müller P., Parkinson M. Behavioral interface specification languages. ACM Comput. Surv. vol. 44, issue 3, article 16, 58 pages. June 2012.

[15]. ISO/IEC 9899: 2011: C11 standard for C programming language. JTC and ISO. April 7, 2016.

[16]. Hubert T., Marché C. Separation Analysis for Deductive Verification. Heap Analysis and Verification (HAV'07). Braga, Portugal, March 2007, pp. 81–93.

[17]. Moy Y. Union and Cast in Deductive Verification. Proceedings of the C/C++ Verification Workshop. Technical Report ICIS-R07015, pp. 1–16. Radboud University Nijmegen. July, 2007.

[18]. Signoles J. E-ACSL Executable ANSI/ISO C Specification Language. Version 1.12. CEA LIST. May, 2017. Available at: https://frama-c.com/download/e-acsl/e-acsl.pdf, accessed 15.10.2017.

[19]. Marché C. [Frama-c-discuss] Frama-C/WP and CVC4 (version 1.5). Available at: https://lists.gforge.inria.fr/pipermail/frama-c-discuss/2017-August/005338.html, accessed 15.10.2017.