Верификация драйверов операционной системы Linux[1]

Бейер Д. (Университет Пассау, Германия), Петренко А. К. (ИСП РАН)
dirk.beyer@sosy-lab.org, petrenko@ispras.ru

Аннотация. Верификация драйверов ОС Linux – это широкая область для применения различных методов верификации, в частности, методов проверки свойств безопасности и надежности программ, а также функциональной верификации. Драйверы Linux – это индустриальное программное обеспечение, на стабильность которого полагаются ИТ-инфраструктуры. В силу этого к надежности и корректности их работы предъявляют жесткие требования, в свою очередь, это означает, что если инженер-верификатор обнаружил ошибку в драйвере, он может рассчитывать на быструю реакцию сообщества разработчиков в плане подтверждения и исправления этой ошибки. Драйверы Linux – сложное низкоуровневое системное программное обеспечение, и его характеристики требуют применения различных техник анализа программ: использования SMT-решателей, методов верификации моделей (model checking) и других методов верификации. Точность и эффективность этих методов за последнее время значительно повысилась, и поэтому сложная задача верификации драйверов ОС Linux становится реальной по мере использования этих достижений в инструментах верификации.

Ключевые слова. Ядро операционной системы, Linux, инструменты верификции, анализ программ, верификация моделей, анализ указателей, анализ структур данных, свойства безопасности, ограничиваемая проверка моделей, символьная верификация, анализ с явными значениями, условная верификация моделей, анализ завершимости, верификация параллельных программ, сравнение методов верификации.

1. Введение

Ядро ОС Linux в настоящее время является одной из наиболее важных программных систем в нашем обществе. Linux используется в качестве ядра в ряде распространенных настольных операционных систем (таких как Ubuntu, Fedora, Debian, Gentoo), и поэтому в надежности и стабильности ядра заинтересовано большое число пользователей этих систем. Возможно, еще

[1] Данная работа была .поддержана Государственным контрактом с Министерством образования и науки РФ № 11.519.11.4006 от 18 августа 2011 г., программное мероприятие 1.9 «Проведение научно-исследовательских работ совместно с иностранными научными организациями».

важнее то, что доминирующие на рынке серверные операционные системы основаны на Linux. Практически все суперкомпьютеры (90% в 2010 году) работают под управлением операционной системы на основе этого ядра. Все большее число встраиваемых устройств, таких как смартфоны, используют Linux в качестве ядра (например, Android, Maemo, WebOS). Все это объясняет возрастающую потребность в автоматической верификации компонентов операционной системы Linux.

Компания Microsoft установила, что ошибки в драйверах устройств являются одной из наиболее важных причин сбоев в работе выпускаемых ею операционных систем. Как следствие, компания существенно увеличила надежность Windows, интегрировав инструмент Static Driver Verifier (SDV) в свой производственный цикл. Основные принципы, лежащие в основе работы этого инструмента, были разработаны в рамках исследовательского проекта SLAM [1]. Сейчас SDV по умолчанию входит в состав набора инструментов для разработки драйверов ОС Windows – Windows Driver Kit (WDK).

Для ОС Linux в настоящее время не существует индустриального проекта по верификации, сопоставимого по размерам с SDV. Но сообщество разработчиков постоянно ищет средства автоматической верификации наиболее критичных аспектов работы этой ОС, а сообщество исследователей в области верификации использует драйверы Linux как область для применения новых подходов анализа. За последние годы было создано три среды верификации на основе драйверов ОС Linux: Linux Driver Verification (LDV)[2] [23, 31], Avinux [27] и DDVerify[3] [34].

Код ядра Linux является распространенным источником задач верификации [17, 22, 24, 25, 30]. Драйверы Linux предоставляют уникальное сочетание характеристик, которые привлекают исследователей и практиков возможностью испытать на них свои инструменты. Наиболее важные преимущества использования кода ядра Linux для постановки задач верификации следующие:

- Данное программное обеспечение является востребованным – многие люди заинтересованы в результатах его верификации.
- Любая ошибка в драйвере является потенциально критической, потому что драйверы работают с тем же уровнем привилегий, что и остальное ядро ОС Linux, в адресном пространстве ядра.
- Суммарный объем исходного кода драйверов весьма значителен (около 10 млн. строк) и при этом постоянно возрастает.
- Задачи верификации драйверов достаточно трудны и интересны, но не настолько сложны, чтобы быть безнадежными.

[2] http://linuxtesting.org/project/ldv
[3] http://www.cprover.org/ddverify

- Большинство драйверов Linux имеют открытый исходный код и поэтому их легко использовать в проектах по верификации и в исследовательских проектах.

Хотя в области верификации программного обеспечения было сделано много новых достижений, требуются дополнительные усилия для того, чтобы использовать их на практике и применять их к сложному индустриальному коду, такому как драйверы устройств ОС Linux. Недавно проведенное соревнование по верификации программного обеспечения (SV-COMP'12)[4] [3] показало, что категория драйверов устройств представляет сложность даже для самых современных инструментов анализа программ.

2. Направления исследований

2.1. Анализ указателей и динамические структуры данных

Многие свойства надежности и безопасности драйверов устройств зависят от точного анализа указателей и различных структур данных в куче. Анализ указателей изучен достаточно хорошо, однако использование низкоуровневого кода, характерное для системного программирования, существенно осложняет реализацию существующих подходов. Проект LDV достиг существенных успехов в этой области благодаря реализации более точного анализа указателей в инструменте верификации BLAST [29]. Улучшенная версия оригинального инструмента BLAST стала победителем соревнований SV-COMP'12 на задачах верификации, полученных из драйверов ОС Linux [28].

Анализ структур данных является актуальной исследовательской темой, по которой в последние годы были получены заметные результаты. Следует отметить, что достаточно большого набора общедоступных тестовых задач для практического сравнения существующих реализаций не существует. Одним из примеров является инструмент Predator – современный статический анализатор, предоставляющий средства проверки структур данных и безопасности работы с памятью [16].

2.2. Символьная верификация

Благодаря развитию SMT-решателей, символьное представление абстрактных состояний программы на основе формул в настоящее время стало эффективным и результативным. Инструменты SDV и SLAM компании Microsoft [1], а также ряд современных академических инструментов основаны на предикатной абстракции [5, 8, 12, 18]. Несколько инструментов сочетают такие концепции, как уточнение абстракции по контрпримерам (counterexample-guided abstract refinement, CEGAR) [11], различные виды

[4] http://sv-comp.sosy-lab.org

анализа динамической памяти (так называемого шейп-анализа, shape analysis), абстрактные деревья достижимости [5], ленивую абстракцию [21], интерполяцию [20] и крупноблочное кодирование (large-block encoding) [4, 9]. Ограничиваемая проверка моделей (bounded model checking) [10] также имеет большое практическое значение и показывает впечатляющие результаты на соревнованиях по верификации [14, 32].

Недостаточно исследованными остаются проблемы определения интерполянтов (существует большой диапазон между слабыми и сильными интерполянтами), проблемы размеров блоков для кодирования (какой критерий нужно использовать для определения конца блока, который целиком кодируется в формулу в ходе одной операции вычисления абстрактного постусловия) и проблемы выбора порядка обхода графа потока управления программы (нацеленность на максимизацию покрытия (coverage-directed verification), поиск в ширину, в глубину и др.). Другим важным и многообещающим подходом, который до недавнего времени в значительной степени игнорировался в верификации программного обеспечения, является возможность представления абстрактных состояний и отношения переходов полностью в виде двоичных решающих диаграмм (Binary Decision Diagrams, BDD). Некоторый успех в этой области наблюдается, например, в опыте расширения инструментов CPAchecker и Java PathFinder механизмами использования BDD для представления пространства состояний, которое образуют булевы переменные, задающие параметры построения конфигураций, например, операционных систем и других систем, образующих так называемые product lines [33].

2.3. Анализ с явными значениями (Explicit-State Verification)

Некоторые инструменты проверки моделей программ с явными значениями являются достаточно успешными в своих областях применения (например, Spin и Java PathFinder). Чтобы применить эту технологию для верификации драйверов операционных систем масштабируемым образом, представляется интересным включить в нее современные успешные подходы из области символьной верификации. Например, CEGAR должен быть использован для автоматического построения абстракции, а интерполяция Крейга для явных значений может указать, какие части пространства состояний необходимо анализировать.

2.4. Комбинация приемов верификации

В прошлом было предложено несколько методов параллельной комбинации различных существующих видов анализа программ [7, 15]. Эти методы чрезвычайно эффективны и должны получить большее внимание и воплощение на практике. Их практическому применению препятствуют технические трудности: объединяемые виды анализа должны использовать одинаковый алгоритм обхода графа потока управления, они должны быть

реализованы на одном и том же языке программирования в совместимых средах и должны работать на одной и той же физической машине.

Последовательная комбинация различных видов анализа с использованием условной верификации моделей (conditional model checking) представляет эффективное решение этой проблемы [6]. Разные инструменты и методы могут выполняться последовательно один за другим и пытаться решить задачу верификации, используя свои наиболее сильные стороны. Инструменту условной верификации моделей сообщается, когда нужно отказаться от продолжения верификации (при помощи так называемого входного условия, input condition). Входные условия дают гибкий способ ограничения или сдерживания хода верификации одним из выбранных методов. Выходные условия предоставляют успешно верифицированную часть пространства состояний. Следующий верификатор может использовать условия, полученные от предыдущих запусков других инструментов, чтобы не выполнять повторно ту же самую работу, а сосредоточить свои усилия на оставшейся части задания.

2.5. Анализ завершимости

Область верификации, требующая большего внимания исследователей, – это анализ завершимости. Существует несколько инструментов проверки завершимости программ (наиболее заметным среди которых является ARMC [26]), но метод пока еще не настолько широко распространен, как это следовало бы ожидать. Подход был заимствован и в дальнейшем улучшен компанией Microsoft в рамках проекта Terminator [13].

2.6. Параллелизм (Concurrency)

В силу возрастающего распространения многоядерных компьютеров, верификация многопоточного программного обеспечения со временем становится все более важной областью исследований. Поиск гонок и взаимных блокировок является неотъемлемой частью средств контроля качества. Поэтому данные виды проверок необходимо применять в том числе и к драйверам ОС Linux. Сообщество исследователей в области верификации активно открывает новые концепции и реализует новые инструменты для решения этой задачи (например, ESBMC [14], SATabs [2], Threader [19]). Интересно заметить, что наилучшим инструментом проверки задач на параллелизм на последнем соревновании оказался инструмент ограничиваемой проверки моделей [14].

3. Заключение

Мы обрисовали основные аргументы в пользу рассмотрения драйверов операционной системы Linux в качестве области для практического применения различных методов и инструментов верификации программ. Важно разрабатывать инструменты верификации, которые достаточно

эффективны и результативны для того, чтобы успешно справиться с задачей проверки программных компонентов, столь сложных, как драйверы устройств. Выгоды двойные: обществу важно получать такое ответственное программное обеспечение верифицированным, а сообществу исследователей в области верификации важно получать практические задачи по верификации для дальнейшей разработки и улучшения своих методов. Мы представили обзор современного состояния дел в области верификации и указали направления исследований, больше всего нуждающиеся в их дальнейшем развитии.

Литература

[1] T. Ball, S. K. Rajamani. The Slam Project: Debugging System Software via Static Analysis // Proc. POPL, pp. 1–3. ACM (2002)

[2] G. Basler, A. Donaldson, A. Kaiser, D. Kröning, M. Tautschnig, T. Wahl. SATABS: A Bit-Precise Verifier for C Programs // C. Flanagan, B. König (eds.). TACAS 2012. LNCS, vol. 7214, pp. 552–555. Springer, Heidelberg (2012)

[3] D. Beyer. Competition on Software Verification // C. Flanagan, B. König (eds.). TACAS 2012. LNCS, vol. 7214, pp. 504–524. Springer, Heidelberg (2012)

[4] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, R. Sebastiani. Software Model Checking via Large-block Encoding // Proc. FMCAD, pp. 25–32. IEEE (2009)

[5] D. Beyer, T. A. Henzinger, R. Jhala, R. Majumdar. The Software Model Checker Blast // Int. J. Softw. Tools Technol. Transfer 9(5-6), 505–525 (2007)

[6] D. Beyer, T. A. Henzinger, M. E. Keremoglu, P. Wendler. Conditional Model Checking: A Technique to Pass Information Between Verifiers // Proc. FSE. ACM (2012)

[7] D. Beyer, T. A. Henzinger, G. Theoduloz. Program Analysis with Dynamic Precision Adjustment // Proc. ASE, pp. 29–38. IEEE (2008)

[8] D. Beyer, M. E. Keremoglu. CPAchecker: A Tool for Configurable Software Verification // G. Gopalakrishnan, S. Qadeer (eds.). CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011)

[9] D. Beyer, M. E. Keremoglu, P. Wendler. Predicate Abstraction with Adjustable-Block Encoding // Proc. FMCAD, pp. 189–197. FMCAD (2010)

[10] A. Biere, A. Cimatti, E. Clarke, Y. Zhu. Symbolic Model Checking without BDDs. // W. R. Cleaveland (ed.). TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)

[11] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. Counterexample-Guided Abstraction Refinement for Symbolic Model Checking // J. ACM 50(5), 752–794 (2003)

[12] E. Clarke, D. Kröning, N. Sharygina, K. Yorav. SatAbs: SAT-Based Predicate Abstraction for ANSI-C // N. Halbwachs, L. D. Zuck (eds.). TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)

[13] B. Cook, A. Podelski, A. Rybalchenko. Terminator: Beyond Safety // T. Ball, T., R. B. Jones (eds.). CAV 2006. LNCS, vol. 4144, pp. 415–418. Springer, Heidelberg (2006)

[14] L. Cordeiro, J. Morse, D. Nicole, B. Fischer. Context-Bounded Model Checking with ESBMC 1.17 // C. Flanagan, B. König (eds.). TACAS 2012. LNCS, vol. 7214, pp. 534–537. Springer, Heidelberg (2012)

[15] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, X. Rival. Combination of Abstractions in the ASTREE Static Analyzer // M. Okada, I. Satoh (eds.). ASIAN 2006. LNCS, vol. 4435, pp. 272–300. Springer, Heidelberg (2008)

[16] K. Dudka, P. Müller, P. Peringer, T. Vojnar. Predator: A Verification Tool for Programs with Dynamic Linked Data Structures // C. Flanagan, B. König (eds.). TACAS 2012. LNCS, vol. 7214, pp. 545–548. Springer, Heidelberg (2012)
[17] A. Galloway, G. Lüttgen, J. T. Mühlberg, R. I. Siminiceanu. Model-Checking the Linux Virtual File System // N. D. Jones, M. Müller-Olm (eds.). VMCAI 2009. LNCS, vol. 5403, pp. 74–88. Springer, Heidelberg (2009)
[18] S. Grebenshchikov, A. Gupta, N. P. Lopes, C. Popeea, A. Rybalchenko. HSF(C): A Software Verifier Based on Horn Clauses // C. Flanagan, B. Koenig (eds.). TACAS 2012. LNCS, vol. 7214, pp. 549–551. Springer, Heidelberg (2012)
[19] A. Gupta, C. Popeea, A. Rybalchenko. Threader: A Constraint-Based Verifier for Multi-threaded Programs // G. Gopalakrishnan, S. Qadeer (eds.). CAV 2011. LNCS, vol. 6806, pp. 412–417. Springer, Heidelberg (2011)
[20] T. A. Henzinger, R. Jhala, R. Majumdar, K. L. McMillan. Abstractions from Proofs // Proc. POPL, pp. 232–244. ACM (2004)
[21] T. A. Henzinger, R. Jhala, R. Majumdar, G. Sutre. Lazy Abstraction // Proc. POPL, pp. 58–70. ACM (2002)
[22] A. Khoroshilov, V. Mutilin, E. Novikov, P. Shved, A. Strakh. Towards an Open Framework for C Verification Tools Benchmarking // E. Clarke, I. Virbitskaite, A. Voronkov (eds.). PSI 2011. LNCS, vol. 7162, pp. 179–192. Springer, Heidelberg (2012)
[23] A. Khoroshilov, V. Mutilin, A. Petrenko, V. Zakharov. Establishing Linux Driver Verification Process // A. Pnueli, I. Virbitskaite, A. Voronkov (eds.). PSI 2009. LNCS, vol. 5947, pp. 165–176. Springer, Heidelberg (2010)
[24] J. T. Mühlberg, G. Lüttgen. Blasting Linux Code // L. Brim, B. R. Haverkort, M. Leucker, J. van de Pol (eds.). FMICS 2006 and PDMC 2006. LNCS, vol. 4346, pp. 211–226. Springer, Heidelberg (2007)
[25] W. Penninckx, J. T. Mühlberg, J. Smans, B. Jacobs, F. Piessens. Sound Formal Verification of Linux's USB BP Keyboard Driver // A. E. Goodloe, S. Person (eds.). NFM 2012. LNCS, vol. 7226, pp. 210–215. Springer, Heidelberg (2012)
[26] A. Podelski, A. Rybalchenko. Transition Predicate Abstraction and Fair Termination // Proc. POPL, pp. 132–144. ACM (2005)
[27] H. Post, C. Sinz, W. Küchlin. Towards Automatic Software Model Checking of Thousands of Linux Modules — A Case Study with Avinux // Softw. Test., Verif. Reliab. 19(2), 155–172 (2009)
[28] P. Shved, M. Mandrykin, V. Mutilin. Predicate Analysis with BLAST 2.7 // C. Flanagan, B. König (eds.). TACAS 2012. LNCS, vol. 7214, pp. 525–527. Springer, Heidelberg (2012)
[29] П. Е. Швед, В. С. Мутилин, М. У. Мандрыкин. Опыт развития инструмента статической верификации BLAST // Программирование. 2012. Т. 3. с. 24–35.
[30] М. У. Мандрыкин, В. С. Мутилин, Е. М. Новиков, А. В. Хорошилов, П. Е. Швед. Использование драйверов устройств операционной системы Linux для сравнения инструментов статической верификации // Программирование. 2012. Т. 5. С. 54–71.
[31] В. С. Мутилин, Е. М. Новиков, А. В. Страх и др. Архитектура Linux Driver Verification // Труды Института системного программирования РАН. 2011. Т. 20. С. 163–187.
[32] C. Sinz, F. Merz, S. Falke. LLBMC: A Bounded Model Checker for LLVM's Intermediate Representation // C. Flanagan, B. König (eds.). TACAS 2012. LNCS, vol. 7214, pp. 542–544. Springer, Heidelberg (2012)

[33] A. von Rhein, S. Apel, F. Raimondi. Introducing Binary Decision Diagrams in the Explicit-State Verification of Java Code // Proc. Java Pathfinder Workshop (2011)
[34] T. Witkowski, N. Blanc, D. Kröning, G. Weissenbacher. Model Checking Concurrent Linux Device Drivers // Proc. ASE, pp. 501–504. ACM (2007)