ИСП Институт Системного Программирования
Российской Академии наук

ISSN 2079-8156 (Print)
ISSN 2220-6426 (Online)

Труды Института Системного Программирования РАН

Proceedings of the Institute for System Programming of the RAS

Том 28, выпуск 5

Volume 28, issue 5

Москва 2016

ИСП РАН

ISSN 2079-8156 (Print), ISSN 2220-6426 (Online)

Труды Института системного программирования РАН

Proceedings of the Institute for System Programming of the RAS

Труды ИСП РАН – это издание с двойной анонимной системой рецензирования, публикующее научные статьи, относящиеся ко всем областях системного программирования, технологий программирования и вычислительной техники. Целью издания является формирование научно-информационной среды в этих областях путем публикации высококачественных статей в открытом доступе.

Издание предназначено для исследователей, студентов и аспирантов, а также практиков. Оно охватывает широкий спектр тем, включая, в частности, следующие:

- операционные системы;
- компиляторные технологии;
- базы данных и информационные системы;
- параллельные и распределенные системы;
- автоматизированная разработка программ;
- верификация, валидация и тестирование;
- статический и динамический анализ;
- защита и обеспечение безопасности ПО;
- компьютерные алгоритмы;
- искусственный интеллект.

Журнал издается по одному тому в год, шесть выпусков в каждом томе.

Поддерживается открытый доступ к содержанию издания, обеспечивая доступность результатов исследований для общественности и поддерживая глобальный обмен знаниями.

Труды ИСП РАН реферируются и/или индексируются в:

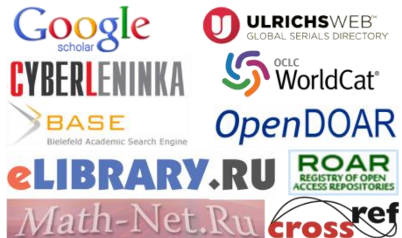
Proceedings of ISP RAS are a double-blind peer-reviewed journal publishing scientific articles in the areas of system programming, software engineering, and computer science. The journal's goal is to develop a respected network of knowledge in the mentioned above areas by publishing high quality articles on open access.

The journal is intended for researchers, students, and practitioners. It covers a wide variety of topics including (but not limited to):

- Operating Systems.
- Compiler Technology.
- Databases and Information Systems.
- Parallel and Distributed Systems.
- Software Engineering.
- Software Modeling and Design Tools.
- Verification, Validation, and Testing.
- Static and Dynamic Analysis.
- Software Safety and Security.
- Computer Algorithms.
- Artificial Intelligence.

The journal is published one volume per year, six issues in each volume.

Open access to the journal content allows to provide public access to the research results and to support global exchange of knowledge. **Proceedings of ISP RAS** is abstracted and/or indexed in:



УДК004.45

Редколлегия

Главный редактор - Аветисян Арутюн Ишханович, член-корр. РАН, д.ф.-м.н., ИСП РАН (Москва, Российская Федерация)

Заместитель главного редактора - Кузнецов Сергей Дмитриевич, д.т.н., профессор, ИСП РАН (Москва, Российская Федерация)

Бурдонов Игорь Борисович, д.ф.-м.н., ИСП РАН (Москва, Российская Федерация)
Воронков Андрей Анатольевич, д.ф.-м.н., профессор, Университет Манчестера (Манчестер, Великобритания)
Вирбицкайте Ирина Бонавентуровна, профессор, д.ф.-м.н., Институт систем информатики им. академика А.П. Ершова СО РАН (Новосибирск, Россия)
Гайсарян Сергей Суренович, к.ф.-м.н., ИСП РАН (Москва, Российская Федерация)
Евтушенко Нина Владимировна, профессор, д.т.н., ТГУ (Томск, Российская Федерация)
Карпов Леонид Евгеньевич, д.т.н., ИСП РАН (Москва, Российская Федерация)
Коннов Игорь Владимирович, к.ф.-м.н., Технический университет Вены (Вена, Австрия)
Косачев Александр Сергеевич, к.ф.-м.н., ИСП РАН (Москва, Российская Федерация)
Кузюрин Николай Николаевич, д.ф.-м.н., ИСП РАН (Москва, Российская Федерация)
Ластовецкий Алексей Леонидович, д.ф.-м.н., профессор, Университет Дублина (Дублин, Ирландия)
Ломазова Ирина Александровна, д.ф.-м.н., профессор, Национальный исследовательский университет «Высшая школа экономики» (Москва, Российская Федерация)
Новиков Борис Асенович, д.ф.-м.н., профессор, Санкт-Петербургский государственный университет (Санкт-Петербург, Россия)
Петренко Александр Константинович, д.ф.-м.н., ИСП РАН (Москва, Российская Федерация)
Петренко Александр Федорович, д.ф.-м.н., Исследовательский институт Монреаля (Монреаль, Канада)
Семенов Виталий Адольфович, д.ф.-м.н., профессор, ИСП РАН (Москва, Российская Федерация)
Томилин Александр Николаевич, д.ф.-м.н., профессор, ИСП РАН (Москва, Российская Федерация)
Черных Андрей, д.ф.-м.н., профессор, Научно-исследовательский центр CICESE (Энсенада, Нижняя Калифорния, Мексика)
Шнитман Виктор Зиновьевич, д.т.н., ИСП РАН (Москва, Российская Федерация)
Шустер Ассаф, д.ф.-м.н., профессор, Технион — Израильский технологический институт Technion (Хайфа, Израиль)

Адрес: 109004, г. Москва, ул. А. Солженицына, дом 25.
Телефон: +7(495) 912-44-25
E-mail: info-isp@ispras.ru
Сайт: http://www.ispras.ru/proceedings/

Editorial Board

Editor-in-Chief - Arutyun I. Avetisyan, Corresponding Member of RAS, Dr. Sci. (Phys.–Math.), Institute for System Programming of the RAS (Moscow, Russian Federation)

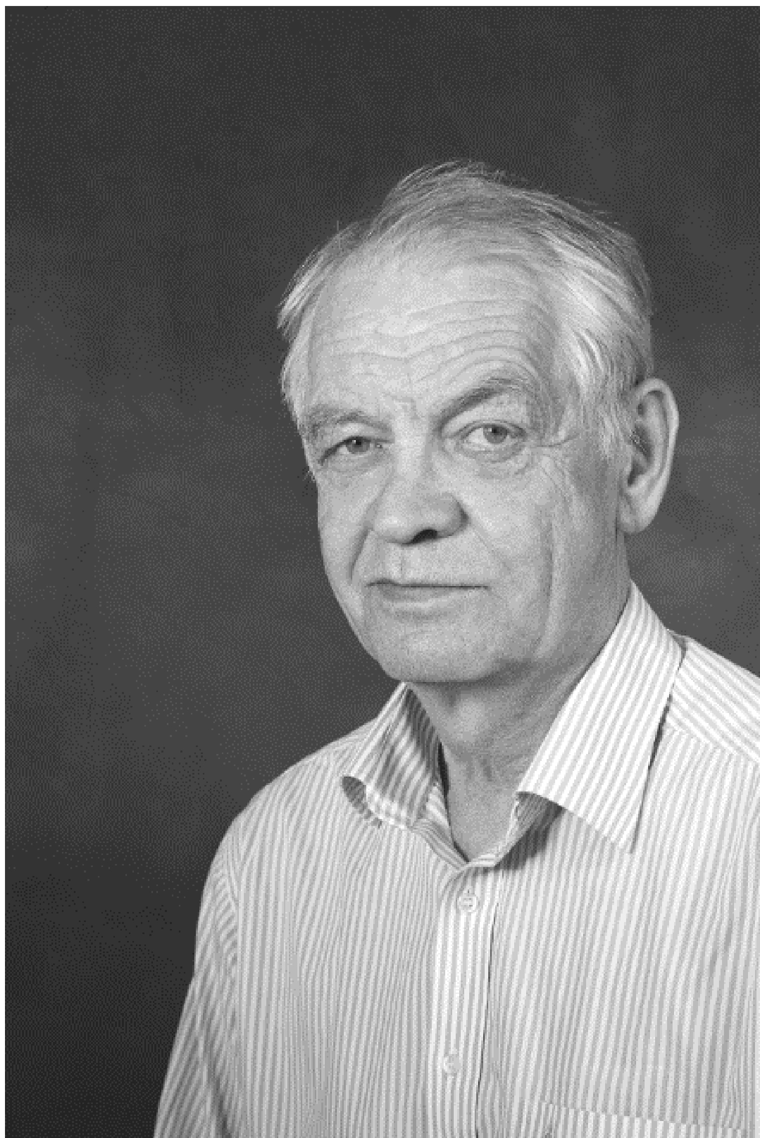
Deputy Editor-in-Chief - Sergey D. Kuznetsov, Dr. Sci. (Eng.), Professor, Institute for System Programming of the RAS (Moscow, Russian Federation)

Igor B. Burdonov, Dr. Sci. (Phys.–Math.), Institute for System Programming of the RAS (Moscow, Russian Federation)
Andrei Chernykh, Dr. Sci., Professor, CICESE Research Centre (Ensenada, Lower California, Mexico)
Sergey S. Gaissaryan, PhD (Phys.–Math.), Institute for System Programming of the RAS (Moscow, Russian Federation)
Leonid E. Karpov, Dr. Sci. (Eng.), Institute for System Programming of the RAS (Moscow, Russian Federation)
Igor Konnov, PhD (Phys.–Math.), Vienna University of Technology (Vienna, Austria)
Alexander S. Kossatchev, PhD (Phys.–Math.), Institute for System Programming of the RAS (Moscow, Russian Federation)
Nikolay N. Kuzyurin, Dr. Sci. (Phys.–Math.), Institute for System Programming of the RAS (Moscow, Russian Federation)
Alexey Lastovetsky, Dr. Sci. (Phys.–Math.), Professor, UCD School of Computer Science and Informatics (Dublin, Ireland)
Irina A. Lomazova, Dr. Sci. (Phys.–Math.), Professor, National Research University Higher School of Economics (Moscow, Russian Federation)
Boris A. Novikov, Dr. Sci. (Phys.–Math.), Professor, St. Petersburg University (St. Petersburg, Russia)
Alexander K. Petrenko, Dr. Sci. (Phys.–Math.), Institute for System Programming of the RAS (Moscow, Russian Federation)
Alexandre F. Petrenko, PhD, Computer Research Institute of Montreal (Montreal, Canada)
Assaf Schuster, Ph.D., Professor, Technion - Israel Institute of Technology (Haifa, Israel)
Vitaly A. Semenov, Dr. Sci. (Phys.–Math.), Professor, Institute for System Programming of the RAS (Moscow, Russian Federation)
Victor Z. Shnitman, Dr. Sci. (Eng.), Institute for System Programming of the RAS (Moscow, Russian Federation)
Alexander N. Tomilin, Dr. Sci. (Phys.–Math.), Professor, Institute for System Programming of the RAS (Moscow, Russian Federation)
Irina B. Virbitskaite, Dr. Sci. (Phys.–Math.), The A.P. Ershov Institute of Informatics Systems, Siberian Branch of the RAS (Novosibirsk, Russian Federation)
Andrey Voronkov, Dr. Sci. (Phys.–Math.), Professor, University of Manchester (Manchester, UK)
Nina V. Yevtushenko, Dr. Sci. (Eng.), Tomsk State University (Tomsk, Russian Federation)

Address: 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Tel: +7(495) 912-44-25
E-mail: info-isp@ispras.ru
Web: http://www.ispras.ru/en/proceedings/

© Институт Системного Программирования РАН, 2016



Посвящается памяти академика

ВИКТОРА ПЕТРОВИЧА ИВАННИКОВА

(27.02.1940 - 27.11.2016)

Содержание

Предисловие
А.И. Аветисян 9

Автоматическое обнаружение использования неинициализированных значений в рамках полносистемной эмуляции
Н.А. Белов *11*

Автоматическое доказательство безопасности локальных пустых указателей
А.В. Когтенков *27*

Когда защита стека в компиляторах не срабатывает?
П. Довгалюк, В. Макаров *55*

Оценка критичности программных дефектов в условиях работы современных защитных механизмов
А.Н. Федотов, В.А. Падарян, В.В. Каушан, Ш.Ф. Курмангалеев, А.В. Вишняков, А.Р. Нурмухаметов *73*

Применение диверсифицирующих и обфусцирующих преобразований для изменения сигнатуры программного кода
А.Р. Нурмухаметов *93*

Формализация определения ошибок при статическом символьном выполнении
В.К. Кошелев *105*

Обнаружение ошибок доступа к буферу в программах на языке C/C++ с помощью статического анализа
И.А. Дудина *119*

Поиск ошибок выхода за границы буфера в бинарном коде программ
В.В. Каушан *135*

Использование анализа недостижимого кода в статическом анализаторе для поиска ошибок в исходном коде программ
Р.Р. Мулюков, А.Е. Бородин *145*

Вычисление входных данных для достижения определенной функции в программе методом итеративного динамического анализа
А.Ю. Герасимов, Л.В. Круглов 159

Ускорение оптимизации программ во время связывания
К.Ю. Долгорукова, С.В. Аришин 175

Задача глобального распределения регистров во время динамической двоичной трансляции
К.А. Батузов 199

Платформенно-независимый и масштабируемый инструмент поиска клонов кода в бинарных файлах
А.К. Асланян, Ш.Ф. Курмангалеев, В.Г. Варданян, М.С. Арутюнян, С.С. Саргсян 215

Оптимизация читаемости тестов порождаемых при символьных вычислениях
И.А. Якимов, А.С. Кузнецов 227

Декларативный язык FlexT – инструмент анализа и документирования бинарных форматов данных
А.Е. Хмельнов, И.В. Бычков, А.А. Михайлов 239

Table of Contents

Foreword
A.I. Avetisyan 9

Automatic uninitialized value usage detection during full-system emulation
N.A. Belov *11*

Mechanically Proved Practical Local Null Safety
A.V. Kogtenkov *27*

When stack protection does not protect the stack?
P. Dovgalyuk, V. Makarov *55*

Severity software defects estimation in presence of modern defense mechanisms
A.N. Fedotov, V.A. Padaryan, V.V. Kaushan, Sh.F. Kurmangaleev, A.V. Vishnyakov, A.R. Nurmukhametov *73*

The Application of Compiler-based Obfuscation and Diversification for Program Signature Modification
A.R. Nurmukhametov *93*

Formalization of Error Criteria for static symbolic execution
V.K. Koshelev *105*

Inter-procedural buffer overflows detection in C/C++ source code via static analysis
I. Dudina *119*

Buffer overrun detection method in binary code
V.V. Kaushan *135*

Using unreachable code analysis in static analysis tool for finding defects in source code
R.R. Mulyukov, A.E. Borodin *145*

Input data generation for reaching specific function in program by iterative dynamic analysis method
A.Y. Gerasimov, L.V. Kruglov *159*

Link-time optimization speedup
K. Dolgorukova, S. Arishin 175

Global register allocation during dynamic binary translation
K.A. Batuzov 199

Platform-independent and scalable tool for binary code clone detection
H.K. Aslanyan, S.F. Kurmangaleev, V.G. Vardanyan, M.S. Arutunian, S.S. Sargsyan 215

Test Readability Optimization in Context of Symbolic Execution
I.A. Yakimov, A.S. Kuznetsov 227

A declarative language FlexT for analysis and documenting of binary data formats
A.Y. Hmelnov, I.V. Bychkov, A.A. Mikhailov. 239

Предисловие

Первая Открытая конференция ИСП РАН состоялась 1-2 декабря 2016 г. в здании Президиума РАН в Москве. Основным инициатором этой конференции являлся выдающийся российский ученый и организатор науки, основатель и многолетний директор ИСП РАН академик Виктор Петрович Иванников, безвременно и скоропостижно скончавшийся за три дня до начала работы конференции. Организаторы Открытой конференции ИСП РАН провели конференцию без Виктора Петровича, посвятив ее светлой памяти этого замечательного человека.

По замыслу В.П. Иванникова и других организаторов конференции, Открытая конференция ИСП РАН – это ежегодная конференция, в рамках которой ИСП РАН проводит серию тематических мероприятий по направлениям IT-индустрии, в которых Институтом накоплен многолетний опыт при выполнении фундаментальных исследований, разработок инновационных технологий, а также реализации совместно с партнерами конкретных проектов по внедрению разработанных технологий в промышленность.

Целью Открытой конференции является поддержка и развитие созданной в ИСП РАН экосистемы генерации инноваций и воспроизводства кадров высшей квалификации в области системного программирования, а также создания предпосылок для повышения уровня использования новейших IT-технологий в деятельности образовательных, научно-исследовательских организаций и дальнейшего их внедрения в промышленность России.

На Открытой конференции ИСП РАН представляются доклады как по фундаментальным и прикладным исследованиям, так и по аспектам, связанным с внедрением новых технологий. С докладами приглашаются признанные эксперты отечественных и зарубежных научных и образовательных организаций, представители ведущих IT-компаний, а также исследователи, чьи работы прошли независимое профессиональное рецензирование.

В настоящем выпуске Трудов ИСП РАН представлены статьи по одному из наиболее развитых направлений исследований и разработок в ИСП РАН – технологии анализа, моделирования и трансформации программ. Все эти статьи подготовлены по материалам докладов, представленных на Открытой конференции ИСП РАН в 2016 г.

Член-корреспондент РАН А.И. Аветисян

Preface

The first Open Conference of ISPRAS has been held December 1-2, 2016 in the building of the Presidium of the Russian Academy of Sciences in Moscow. The main initiator of this conference was an outstanding Russian scientist and science organizer, founder and long-standing director of ISP RAS Academician Viktor Ivannikov, who prematurely and suddenly died three days before the start of the conference. The organizers of the Open Conference ISPRAS held a conference without Victor Petrovich, dedicating it to the bright memory of this remarkable man.

According to the V.P. Ivannikov and other organizers of the conference, The Open Conference of the ISPRAS is the annual event, under which ISP RAS arranges a number of conferences devoted to sectors of IT-industry, in which it possesses a many years' experience of fundamental researches conduction, innovative technologies development and implementation, together with partners, of certain projects on introduction the developed technologies in the industry.

The goal of this event is to provide support and development for the innovations generation and high-skilled system programming workers reproduction ecosystem, created in the ISP RAS as well as to create prerequisites for improving the level of using newest information technologies in activity of educational, scientific and research organizations and their further introduction in Russian industry. During the Open Conference of the ISPRAS events, reports on fundamental and applied researches as well as on issues related to new technologies introduction are made.

Acknowledged experts from national and foreign scientific and educational organizations, representatives of leading IT-companies and researchers whose studies have been professionally reviewed are invited to make reports.

This issue of the Proceedings of ISP RAS presents papers on one of the most developed areas of research and development in the ISP RAS – technologies for analysis, modeling and transformation of programs. All these articles are prepared on the basis of reports submitted at the Open Conference of ISPRAS in 2016.

Corresponding Member of RAS A.I. Avetisyan

Автоматическое обнаружение использования неинициализированных значений в рамках полносистемной эмуляции

Н. А. Белов <zodiac@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25
Московский государственный университет имени М.В. Ломоносова,
119991, Россия, Москва, Ленинские горы, д. 1

Аннотация. Описанный в данной статье метод позволяет автоматически обнаруживать использование неинициализированных значений в рамках полносистемной эмуляции. Это актуально для такого низкоуровневого программного обеспечения, как, например, BIOS или начальный загрузчик, выполняющие функции инициализации оборудования и загрузки операционной системы. Ошибки в данных программных системах наиболее опасны и приводят к неработоспособности всей системы целиком. Программное обеспечение подобного рода затруднительно тестировать на реальной аппаратуре, поэтому для этих целей используются эмуляторы различных архитектур. В рамках работы был разработан метод использования теневой памяти (памяти, содержащей информацию об исходной памяти) для хранения и отслеживания состояния регистров и ячеек гостевой памяти. Также были сформулированы критерии обнаружения использования неинициализированных значений и уведомления об ошибках. Разработанный метод был реализован и протестирован на гостевой системе архитектуры x86 в полносистемном эмуляторе QEMU.

Ключевые слова: обнаружение неинициализированных значений; полносистемная эмуляция; инструментирование.

DOI: 10.15514/ISPRAS-2016-28(5)-1

Для цитирования: Белов Н.А. Автоматическое обнаружение использования неинициализированных значений в рамках полносистемной эмуляции. Труды ИСП РАН, том 28, вып. 5, 2016 г., стр. 11-26. DOI: 10.15514/ISPRAS-2016-28(5)-1

1. Введение

В современном мире происходит постоянное усложнение компьютерных систем, будь то персональные компьютеры, мобильные телефоны, некоторые модели электронных наручных часов, различное сетевое оборудование или

другие программно-аппаратные комплексы. Поскольку каждая такая система использует множество своего различного низкоуровневого программного обеспечения, такого как BIOS, UEFI, начальные загрузчики и прошивки, выполняющие, к примеру, функции инициализации и тестирования оборудования и дальнейшей загрузки операционной системы, то ошибки в данных программных системах наиболее опасны и приводят к неработоспособности всей системы целиком.

Однако, программное обеспечение подобного рода затруднительно тестировать на реальной аппаратуре, так как нет доступа ко всему контексту выполнения (регистрам и памяти), а использовать аппаратный отладчик дорого и неудобно, поэтому для этих целей используются специальные программы – эмуляторы [1].

2. Обзор и метод решения задачи

В данной статье рассматривается разработка и реализация метода автоматического обнаружения использования неинициализированных значений в рамках полносистемной эмуляции.

Для решения данной задачи был разработан метод хранения и отслеживания состояния регистров и ячеек памяти гостевой системы (инициализированы они или нет), а также сформулированы критерии обнаружения использования неинициализированных значений и уведомления об ошибках.

После этого разработанный метод был реализован и протестирован в полносистемном эмуляторе QEMU [2].

2.1 Обзор существующих методов

Рассмотрим работы, в которых решались похожие задачи инструментирования внутреннего представления и поиска ошибок при работе с памятью.

1. Android Memory Checker Component входил в состав операционной системы Android и использовался для тестирования только пользовательских приложений внутри системы. Метод был реализован на основе эмулятора QEMU и специальной версии библиотеки libc.so. Он позволял находить такие ошибки при работе с памятью, как запись и чтение за пределами выделенного блока, попытки использования некорректных указателей для освобождения или перераспределения памяти, утечки памяти.

   В 2014 году данное средство было исключено из состава Android в связи со сложностью поддержки [3].
2. Valgrind – свободное программное обеспечение, предназначенное для профилирования, отладки использования памяти и обнаружения ее утечек. Memcheck является его основным модулем, обеспечивающим проверку корректности работы тестируемого приложения с памятью. Он может обнаруживать различные виды ошибок, преимущественно

возникающих в программах на Си и Си++. Memcheck обнаруживает множество ошибок, но минусом данного инструмента является то, что он может проверять только пользовательские приложения [4].

Таким образом, оба инструмента не позволяют тестировать низкоуровневое программное обеспечение и используются только для тестирования пользовательских приложений внутри загруженной операционной системы.

2.2 Инструментирование внутреннего представления

Ключевой метод, лежащий в основе работы программной системы Valgrind, – это динамическая двоичная трансляция. Данный метод используется для перевода двоичного кода из одного представления в другое «на лету» и имеет широкое распространение. Например, он применяется для полносистемной эмуляции [5]. Также с его помощью можно инструментировать различное программное обеспечение.

Существует два различных подхода к реализации инструментирования исходного двоичного кода при полносистемной эмуляции, различия которых изображены на рис. 1.

1. **Инструментирование внутри гостевой среды.** В данном случае инструментирование выполняется программным обеспечением внутри гостевой системы, используя ее адресное пространство. Однако, подход имеет очень сильное замедление и трансляция выполняется дважды: во время инструментирования необходимого процесса и во время эмуляции.
2. **Инструментирование вне гостевой среды.** В отличие от предыдущего случая, при таком подходе инструментирование производится эмулятором. Инструментирующий код использует адресное пространство основной машины.

В данной статье мы будем использовать инструментирование вне гостевой среды, так как поставленная задача предполагает возможность тестирования низкоуровневого программного обеспечения, когда невозможно использовать адресное пространство гостевой системы для инструментирующего кода и сопутствующих данных.

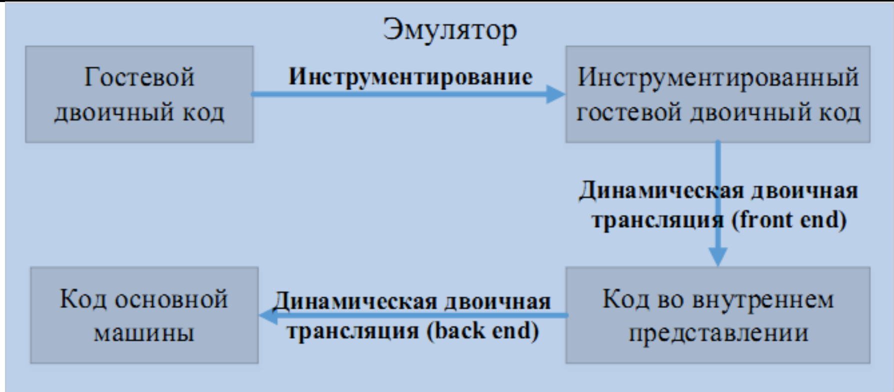


Рис. 1. Различие между методами инструментирования кода

Fig. 1. Differences between code instrumentation methods

2.3 Теневая память

Теневая память – это память, значения которой содержат какую-либо информацию об исходной памяти. При решении поставленной задачи будем использовать теневую память, содержащую информацию об инициализированности значений по соответствующим адресам физической памяти виртуальной машины. Теневая память будет покрывать каждый бит гостевой памяти, причем будем использовать значение «единица» для обозначения того, что бит инициализирован, и «ноль» в противном случае.

Определим операции над значениями теневой памяти аналогично тому, как это сделано в Memcheck [6]. Будем использовать обозначение ***d*** для фактического значения гостевой памяти и ***s*** для соответствующего ему

теневого значения, а ***X*** и ***Y*** для размера операнда и результата в байтах (размер «ноль» используется для однобитового операнда).

1. ***DifD(s1, s2)*** («инициализировано если хотя бы одно инициализировано»). Операция возвращает результат, каждый бит которого установлен в значение «инициализирован», если хотя бы один соответствующий бит операнда имеет значение «инициализирован». Операция реализуется через побитовое логическое «ИЛИ».
2. ***UifU(s1, s2)*** («неинициализировано если хотя бы одно неинициализировано»). Операция возвращает результат, каждый бит которого установлен в значение «неинициализирован», если хотя бы один соответствующий бит операнда имеет значение «неинициализирован». Операция реализуется через побитовое логическое «И».
3. ***ImproveAND(d, s)*** и ***ImproveOR(d, s)***. Операции используются для вычисления теневого значения результата побитовых логических операций «И» и «ИЛИ» соответственно. Значение очередного бита результата определяется в зависимости от соответствующих битов ***d*** и ***s*** по следующим правилам:
   а) для операции ***ImproveAND***: бит имеет значение «инициализирован», если ***d = 0*** и ***s*** имеет значение «инициализировано», так как если один из аргументов логической операции «И» инициализированный нулевой бит, то второй аргумент не повлияет на результат,
   б) для операции ImproveOR: бит имеет значение «инициализирован», если d = 1 и s имеет значение «инициализировано» (из тех же соображений, что и для операции ImproveAND),
   в) для обеих операций в остальных случаях бит получает значение «неинициализирован».
4. ***Left(s)***. Операция симулирует наиболее неблагоприятный вариант распространения статуса неинициализированности значения через флаг переноса во время сложения или вычитания целых чисел. Результатом операции является значение v, в котором установлены все биты от самого старшего до установленного самого младшего. Например, ***Left(0001 0100) = 1111 1100***. Операция реализуется через операции побитового логического «ИЛИ», побитового отрицания и смены знака (***NEG***).
5. ***PCastXY(s)***. Данная группа операций представляет собой операции изменения размера операнда по следующему правилу: если все биты операнда имеют значение «инициализирован», то все биты результата будут иметь значение «инициализирован», иначе все биты результата имеют значение «неинициализирован». Например, ***PCast12(0001***

0100) = 0000 0000 0000 0000 и ***PCast12(1111 1111) = 1111 1111 1111 1111***.

Данное преобразование используется в различных приближениях, в которых проверка статуса инициализированности требует рассмотрения полного значения.

6. ***ZWidenXY(s)***. Группа операций беззнакового расширения аргумента. Для заполнения новых битов используется значение «инициализирован».
7. ***SWidenXY(s)***. Группа операций знакового расширения аргумента. Для заполнения новых битов используется значение старшего бита аргумента.

Начальная инициализация теневой памяти выполняется следующим образом:

1) вся память и регистры помечаются как неинициализированные;
2) участки ***ROM***-памяти помечаются как инициализированные;
3) регистр указателя стека помечается как инициализированный;
4) регистр счетчика инструкций помечается как инициализированный.

2.4 Обнаружение неинициализированных значений

При каждом обнаружении использования неинициализированного значения существуют две стратегии уведомления об ошибке: сообщить об этом сразу или распространить статус инициализированности значений на результат, а затем при наступлении определенных условий проверить его.

Первый вариант будет давать множество ложноположительных срабатываний, так как даже программы на Си, полностью соответствующие стандарту, регулярно копируют неинициализированные значения из памяти и обратно. Рассмотрим пример такого кода, приведенный на рис. 2.

```
1 struct S { int i; char c; };
2 struct S s1, s2;
3 s1.i = 42;
4 s1.c = 'z';
5 s2 = s1;
```

Рис. 2. Пример кода на языке Си

Fig. 2. Code example in C

Все распространенные компиляторы (например, GCC) выровняют размер ***struct S*** до целого числа слов, поэтому структура ***s1*** будет занимать восемь байт, из которых только пять будут инициализированными. Для присваивания ***s1 = s2*** компилятор GCC сгенерирует ассемблерный код, представленный на рис. 3.

```
1 mov eax,DWORD PTR [esp+0x8]
2 mov edx,DWORD PTR [esp+0xc]
3 mov DWORD PTR [esp],eax
4 mov DWORD PTR [esp+0x4],edx
```

*Рис. 3. Ассемблерный код для присваивания **s1 = s2***

*Fig. 3. Assembly code for **s1 = s2** assignment*

Таким образом, из структуры ***s1*** в структуру ***s2*** будут скопированы все восемь байт, несмотря на их смысл, что должно привести к сообщению об ошибке использования неинициализированных значений. В нашей работе будем сообщать о таких ошибках только при наступлении определенных событий, а в остальных случаях передавать неинициализированные значения в результат.

Выделим случаи, при которых необходимо сообщать об использовании неинициализированных значений:

1) неинициализированное значение является адресом для операций загрузки или выгрузки значений из памяти;
2) выполняется условный переход на основании неинициализированного значения;
3) выполняется переход на неинициализированный участок памяти.

2.5 Обновление значений теневой памяти

Как уже было сказано, немедленное информирование об ошибке требуется в малом числе случаев, в остальных же необходимо обновить значение теневой памяти после выполнения исходной операции. Для этого введем следующие правила.

1. **Константы.** Все константы всегда считаются инициализированными значениями.
2. **Операции копирования данных.** Выполняем копирование теневого значения источника в теневое значение принимающего аргумента.
3. **Сложение и вычитание.** Пусть дано ***d3 = Add(d1, d2)*** или ***d3 = Sub(d1, d2)***, тогда очередной бит результата будет иметь значение «инициализирован», когда биты обоих аргументов «инициализированы». Однако, бит результата может быть также «неинициализирован» в том случае, если заем из старших или перенос из младших битов оказался неинициализированным значением. Таким образом, теневой результат определяется как ***s3 = Left(UifU(s1, s2))***.

   Такие же правила используются и для умножения, хотя это не совсем точно. Произведение двух множителей с **N** и **M** последовательными инициализированными младшими битами имеет **N + M** инициализированных младших бит, а не **min(N, M)**, как получается при использовании данного метода. Однако, как показывает опыт разработчиков Memcheck, смысла усложнять используемое решение

нет, так как ложных срабатываний при умножении возникает ничтожно мало.

4. **Операция смены знака (NEG).** Результат теневого значения аналогичен результату выполнения операции ***Sub(0, d)***.
5. **Сложение с переносом и вычитание с займом.** В данной операции будем использовать дополнительный однобитовый аргумент ***vfl***, являющийся теневым значением регистра флагов (например, в архитектуре x86 регистра ***EFLAGS***), который отвечает за перенос или займ. Если этот флаг имеет значение «неинициализирован», то значение всей операции также будет неинициализировано. Таким образом, теневой результат операции определяется как ***s3 = UifU(Left(UifU(s1, s2)), PCast0X(vfl))***, где ***X*** – длина ***s1***, ***s2*** и ***s3***.
6. **Операция побитового исключающего «ИЛИ».** Для ***d3 = Xor(d1, d2)*** результат теневой памяти будет равен ***s3 = UifU(s1, s2)***.
7. **Операция побитового отрицания.** Результат теневого значения аналогичен результату выполнения операции ***Xor(0xFF…FF, d)***.
8. **Побитовые логические «И» и «ИЛИ».** Данные операции требуют проверки фактических значений аргументов, а также их теневых значений. Пусть даны операции ***d3 = And(d1, d2)*** и ***d3 = Or(d1, d2)***, тогда теневой результат операций определяется соответственно как ***s3 = DifD(UifU(d1, d2), DifD(ImproveAND(d1, s1), ImproveAND(d2, s2)))*** и ***s3 = DifD(UifU(d1, d2), DifD(ImproveOR(d1, s1), ImproveOR(d2, s2)))***.
9. **Битовые сдвиги.** Существуют следующие операции битовых сдвигов: сдвиг влево (***shl***), беззнаковый и знаковый сдвиги вправо (***shr*** и ***sar*** соответственно), циклические сдвиги влево и вправо (***rotl*** и ***rotr*** соответственно). Во всех случаях если значение, на которое производится сдвиг, неинициализировано, то весь результат также будет неинциализирован. Иначе для получения теневого результата выполним над теневым значением аргумента такую же операцию, как и для оригинальных значений. Таким образом, для ***d3 = OP(d1, d2)***, где ***d2*** – число, на которое происходит сдвиг, а ***X*** и ***Y*** – длина ***d1***/***d3*** и ***d2*** соответственно, получаем теневой результат ***s3 = UifU(PCastYX(s2), OP(s1, d2))***.

   Для всех операций теневое значение аргумента обрабатывается как и сам аргумент. Для операций ***rotl*** и ***rotr*** теневое значение должно быть сдвинуто абсолютно так же, как и оригинальное. Операции ***shl*** и ***shr*** используют для заполнения нулевые значения, поэтому новые биты теневого результата должны получить значение «инициализировано». Операция ***sar*** копирует старший бит аргумента, поэтому в теневой результат также необходимо копировать старший бит теневого аргумента.

10. **Операции изменения размера.** Операции изменения размера выполняются для теневого результата с помощью операций ***SWiden*** и ***ZWiden*** (для знаковых и беззнаковых расширений соответственно).
11. ***Операции, влияющие на флаги.*** В архитектуре x86 большинство арифметических инструкций устанавливают флаги в регистре ***EFLAGS***. Будем использовать один бит ***vfl*** для хранения состояния инициализированности данного регистра. Для каждой арифметической операции, устанавливающей флаги, будем сначала вычислять теневой результат по описанным выше правилам, а затем получать значение флага как значение функции ***PCastX0*** от теневого результата.

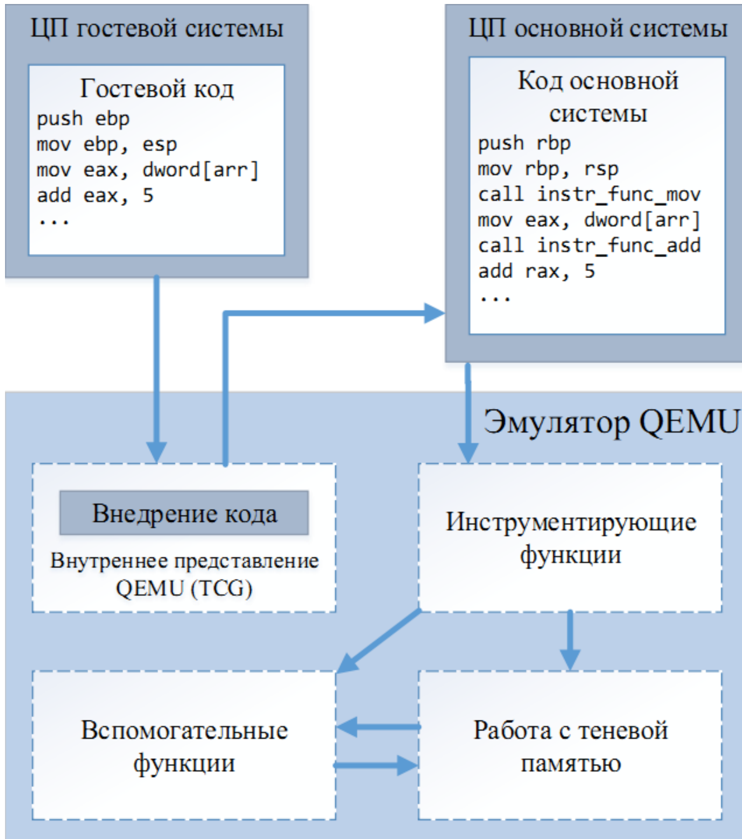


Рис. 4. Взаимодействие частей разработанного метода

Fig. 4. Interaction between parts of developed method

3. Описание реализации метода

В данном разделе будет изложен способ реализации каждого этапа, описанного в предыдущем разделе, в эмуляторе QEMU. Для этого рассмотрим алгоритм взаимодействия частей разрабатываемого метода, представленный на рис. 4.

На данном рисунке мы видим, что внедрение инструментирующего кода будет происходить на стадии трансляции гостевого кода в код внутреннего представления эмулятора. Затем при выполнении кода основной машиной будут вызываться специальные инструментирующие функции по одной перед выполнением каждой инструментируемой операции. Вычисление адреса теневой памяти и операции над ней выделим во множество «вспомогательных функций».

3.1 Внедрение кода

Как уже было сказано ранее, эмулятор QEMU для своей работы использует динамический транслятор TCG, хранящий внутреннее представление в виде двух массивов: коды операций и аргументы операций. Для выполнения инструментирования будем использовать специальные функции – ***helpers*** (здесь и далее для обозначения таких функций будет использоваться термин «функции-помощники»), которые выполняются в адресном пространстве основной машины и специальным образом работают с теневой памятью в зависимости от инструментируемой операции.

Таким образом, внедрение кода будет происходить в три шага.

1. **Вычисление размера внедряемого инструментирующего кода: количества операций и количества аргументов.** Для каждой инструментируемой операции будем хранить два параметра внедряемого кода: количество операций и их суммарное число аргументов. Затем создадим цикл по массиву операций внутреннего представления, в котором будем проверять необходимо ли инструментировать данную операцию. При положительном ответе будем увеличивать соответствующие счетчики: ***total_count*** и ***total_size***.
2. **Сдвиг массивов внутреннего представления для освобождения места под инструментирующий код.** В каждом из двух массивов внутреннего представления сдвинем последовательно все элементы таким образом, чтобы в начале оказалось свободное место под ***total_count*** и ***total_size*** элементов соответственно.
3. **Внедрение кода.** Сформируем массив операций и массив аргументов следующим образом. Для каждой исходной операции будем смотреть, необходимо ли ее инструментировать. В массивы будем записывать исходную операцию и ее аргументы. В том случае, если требуется

инструментирование данной операции, то перед этим еще будем записывать инструментирующий код и его аргументы.

3.2 Теневая память в эмуляторе QEMU

В данном подразделе рассмотрим существующий способ организации гостевой памяти в эмуляторе QEMU, реализацию для него теневой памяти, а также теневую память для переменных внутреннего представления.

Организация памяти в QEMU и теневая память.

Все страницы гостевой памяти, используемые в процессе работы эмулятора, выделяются непрерывными блоками и хранятся в связном списке. Для организации теневой памяти поступим аналогичным образом.

Каждый такой блок гостевой памяти выделяется вызовом функции ***qemu_ram_alloc()***, которая принимает размер в байтах и выделяет память через системный вызов ***mmap()*** на основной машине в процессе QEMU. Существует также несколько других функций, которые позволяют выделять блоки гостевой памяти, но все они, включая функцию ***qemu_ram_alloc()***, для добавления блока в список используют функцию ***ram_block_add()***. Добавим в ее исходный код вызов функции, которая будет создавать блок теневой памяти, соответствующий новому блоку гостевой памяти.

Для ускорения работы по поиску виртуального адреса основной машины, соответствующего виртуальному адресу гостевой машины, эмулятор использует буфер ассоциативной трансляции (*Translation lookaside buffer*, *TLB*), который хранит смещение гостевого виртуального адреса к виртуальному адресу гостевой машины. Поэтому нам необходимо иметь функцию, вычисляющую указатель на теневую память, соответствующую данному виртуальному адресу.

Для работы с теневой памятью реализуем функции проверки гостевой памяти на инициализированность и изменения значения теневой памяти для соответствующего участка гостевой памяти.

Теневая память для переменных внутреннего представления.

Внутренним представлением эмулятора поддерживаются переменные, которые подразделяются на три типа: глобальные (регистры), временные («*temporary*») и локальные временные («*local*»). Глобальные переменные существуют всегда и определяются разработчиком эмулируемой архитектуры. Временная переменная существует только в пределах базового блока (последовательности инструкций, имеющей одну точку входа и одну точку выхода) и уничтожается после его окончания. Локальная временная переменная существует только в пределах блока трансляции (последовательности инструкций, которую эмулятор транслирует за раз). Временные и локальные временные переменные выделяются для каждой функции по отдельности.

QEMU для хранения всех описанных выше типов переменных использует единый массив, поэтому теневую память для них реализуем аналогично. Теневую память каждой переменной будем описывать двумя параметрами: адрес, с которого было загружено значение, и теневое значение. Инициализируем полученный массив в соответствии с правилами, описанными в главе «Обзор и метод решения задачи».

Для работы с теневой памятью переменных внутреннего представления реализуем функции проверки переменной на инициализированность и изменения значения теневой памяти для соответствующей переменной.

При выполнении трансляции гостевого кода во внутреннее представление эмулятор не использует в качестве аргументов для операций переменные, значения которых не объявлены. Более того, в эмуляторе имеются такие проверки, поэтому нет необходимости каждый раз сбрасывать значение теневой памяти для переменных в исходное состояние.

3.3 Обнаружение перехода на неинициализированную область памяти

Переход на неинициализированную область памяти осуществляется следующим образом. В QEMU используется функция ***get_page_addr_code()***, вызывающаяся каждый раз при трансляции гостевого кода (при включенном режиме «singlestep», когда за один раз транслируется одна инструкция). Эта функция имеет одним из своих аргументов гостевой адрес, поэтому добавим в ее исходный код вызов нашей функции, проверяющей инициализированность значения по этому адресу и в случае необходимости выводящей ошибку.

3.4 Инструментирование операций загрузки и сохранения

В эмуляторе QEMU существует две инструкции внутреннего представления, предназначенные для копирования между эмулируемой памятью гостевой машины и переменными:

1. ***qemu_ld_i32*** загружает из эмулируемой памяти гостевой машины значение по адресу ***addr*** в переменную ***ret***.
2. ***qemu_st_i32*** сохраняет значение переменной ***val*** в эмулируемую память гостевой машины по адресу ***addr***.

Инструментирование инструкции загрузки.

Инструментирующая функция-помощник для инструкции загрузки выполняет следующие действия:

1) проверяет значение аргумента ***addr*** на инициализированность, при необходимости уведомляя об ошибке;
2) копирует значение теневой памяти по адресу ***addr*** в теневую память для аргумента ***ret***.

Инструментирование инструкции сохранения.

Инструментирующая функция-помощник для инструкции сохранения работает аналогично инструкции загрузки и выполняет следующие действия:

1) проверяет значение аргумента ***addr*** на инициализированность, при необходимости уведомляя об ошибке;
2) копирует значение теневой памяти для аргумента ***ret*** в теневую память по адресу ***addr***.

3.5 Инструментирование инструкции условного перехода

Инструкция условного перехода во внутреннем представлении имеет следующий прототип: ***brcond_i32 arg1, arg2, cond, int***. Данная инструкция выполняет переход на метку с индексом ***int***, если условие перехода, задаваемое аргументом ***cond*** – истина.

Инструментирующая функция-помощник для инструкции условного перехода проверяет значение аргументов ***arg1*** и ***arg2*** на инициализированность и, если хотя бы один из них неинициализирован, уведомляет об ошибке.

3.6 Инструментирование остальных операций

Инструментирование остальных операций (операции копирования значения, арифметические и логические операции, операции сдвига и так далее) осуществляется согласно правилам, описанным в главе «Обзор и метод решения задачи».

Каждой функции-помощнику передаются индексы переменных внутреннего представления, являющихся аргументами исходной операции, после чего в теневую память результата записывается вычисленное теневое значение.

4. Тестирование

Тестирование производилось на гостевой архитектуре x86. Была использована операционная система Baremetal OS – небольшой код, инициализирующий процессор и передающий управление пользовательской программе.

Код операционной системы был изменен для создания условий, позволяющих протестировать все три случая обнаружения использования неинициализированных значений, описанных в главе 4.3.

Тестирование производилось следующим образом.

1. **Неинициализированное значение является адресом для операций загрузки или сохранения значений из памяти.** Был написан код, загружающий значение по адресу, указывающему на заведомо неинициализированное значение, а затем выполнена загрузка по адресу, на который указывает значение.
2. **Выполняется условный переход на основании неинициализированного значения.** Был написан код, загружающий значение по адресу, указывающему на заведомо

неинициализированное значение, а затем выполняющий условный переход на основе этого значения.

3. **Выполняется переход на неинициализированный участок памяти.** Был написан код, выполняющий переход на заведомо неинициализированное значение адреса.

5. Заключение

В данной статье был описан метод автоматического обнаружения использования неинициализированных значений в рамках полносистемной эмуляции. В разработанном методе используется технология теневой памяти для хранения и отслеживания состояния регистров и ячеек гостевой памяти.

Также в статье были сформулированы критерии обнаружения использования неинициализированных значений и уведомления об ошибках.

Разработанный метод был реализован и протестирован на гостевой системе архитектуры x86 в полносистемном эмуляторе QEMU. Для тестирования было написано три простых примера на каждый случай обнаружения использования неинициализированных значений. Метод показал свою корректную работу на всех примерах.

Список литературы

[1]. Smith J., Nair R. Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005, 656 p.

[2]. QEMU Emulator User Documentation (online). Доступно по ссылке: http://qemu.weilnetz.de/qemu-doc.html, 13.11.2014.

[3]. Android Memory Checker Component (online). Доступно по ссылке: https://github.com/android/platform_external_qemu/blob/791e96ffc61d52eae80f94129a93ff67474f3ff9/docs/ANDROID-MEMCHECK.TXT, 3.12.2014.

[4]. Memcheck: a memory error detector (online). Доступно по ссылке: http://valgrind.org/docs/manual/mc-manual.html, 16.11.2014.

[5]. Bellard F. QEMU, a Fast and Portable Dynamic Translator. Proceedings of the Annual Conference on USENIX Annual Technical Conference, 2005, p. 41.

[6]. Seward J., Nethercote N. Using Valgrind to Detect Undefined Value Errors with Bit-precision. Proceedings of the Annual Conference on USENIX Annual Technical Conference, 2005, p. 2.

Automatic uninitialized value usage detection during full-system emulation

N.A. Belov <zodiac @ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia
Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia

Abstract. Developed method, which is described in this paper, is capable of automated detection of uninitialized values within the scope of full-system emulation. This method is of immediate interest for low-level software, such as BIOS or initial loader, which initializes hardware and loads the operating system. Errors in this kind of software are the most dangerous and lead to system shutdown. This sort of software is difficult to test on real hardware, consequently emulators of different architectures are used for these tasks. In the context of this work a new method of using shadow memory for storing and tracking register states and guest system memory cells. Criteria for detection of uninitialized variables usage and error reporting were defined. For example, these situations fall under the criteria: uninitialized value is the address for loading and unloading values from and to the memory, conditional jump is performed based on uninitialized value or to an uninitialized memory chunk. Developed method was implemented and tested in the guest system of x86 architecture in full-system emulator QEMU. System consists of few instructions, which initialize a processor and transfers control to a user application. Testing was performed on three simple examples for each of the criteria for unitialized values detection. Developed method demonstrated correct results on all examples.

Keywords: automatic uninitialized value usage detection; full-system emulation; instrumentation.

DOI: 10.15514/ISPRAS-2016-28(5)-1

For citation: Belov N.A. Automatic uninitialized value usage detection during full-system emulation. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 5, 2016. pp. 11-26 (in Russian). DOI: 10.15514/ISPRAS-2016-28(5)-1

References

[1]. Smith J., Nair R. Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005, 656 p.

[2]. QEMU Emulator User Documentation (online publication). Available at: http://qemu.weilnetz.de/qemu-doc.html, accessed 13.11.2014.

[3]. Android Memory Checker Component (online publication). Available at: https://github.com/android/platform_external_qemu/blob/791e96ffc61d52eae80f94129a93ff67474f3ff9/docs/ANDROID-MEMCHECK.TXT, accessed 3.12.2014.
[4]. Memcheck: a memory error detector (online publication). Available at: http://valgrind.org/docs/manual/mc-manual.html, accessed 16.11.2014.
[5]. Bellard F. QEMU, a Fast and Portable Dynamic Translator. Proceedings of the Annual Conference on USENIX Annual Technical Conference, 2005, p. 41.
[6]. Seward J., Nethercote N. Using Valgrind to Detect Undefined Value Errors with Bit-precision. Proceedings of the Annual Conference on USENIX Annual Technical Conference, 2005, p. 2.

Mechanically Proved Practical Local Null Safety

A.V. Kogtenkov <kwaxer@mail.ru>
ETH Zürich
Universitätstrasse 19, 8092 Zürich, Switzerland

Abstract. Null pointer dereferencing is a well-known bug in object-oriented programs. It can be avoided by adding special validity rules to a language in which programs are written. Are the rules sufficient to ensure absence of such exceptions? This work focuses on null safety for intra-procedural context where no additional type annotations are needed and formalizes the rules in Isabelle/HOL proof assistant. It then proves null-safety preservation theorem for big-step semantics in a computer-checkable way. Finally, it demonstrates that with such rules null-safe and null-unsafe semantics are equivalent.

Keywords: null safety; void safety; static analysis; Eiffel; formal methods; big-step operational semantics; preservation theorem; operational semantics equivalence.

DOI: 10.15514/ISPRAS-2016-28(5)-2

For citation: Kogtenkov A.V. Mechanically Proved Practical Local Null Safety. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 5, 2016. pp. 27-54. DOI: 10.15514/ISPRAS-2016-28(5)-2

1. Introduction

In his talk at a conference in 2009 Tony Hoare called his invention of the null reference in 1965 a “billion-dollar mistake” ([8]). The reason is simple: most object-oriented languages suffer from a problem of null pointer dereferencing. What does it mean in practice? It is possible that at run-time some variables (or expressions in general) do not reference any existing object, or are *null*. On the other hand the core of object-oriented languages is in the ability to make a call on an object. Given that there is no object when the reference is null, the run-time should signal to the program about the issue.

Provided that most popular languages do not prevent null-pointer dereferencing at compile time, it remains one of the day-to-day issue discovered in open source and private software. As of May 2016 a public database of cybersecurity vulnerabilities known as Common Vulnerabilities and Exposures (CVE®) [3] operated by *MITRE* and funded by Computer Emergency Readiness Team (*CERT*) has 727 entries

mentioning null pointer dereference bugs explicitly. The distribution by years is shown in figure 1.

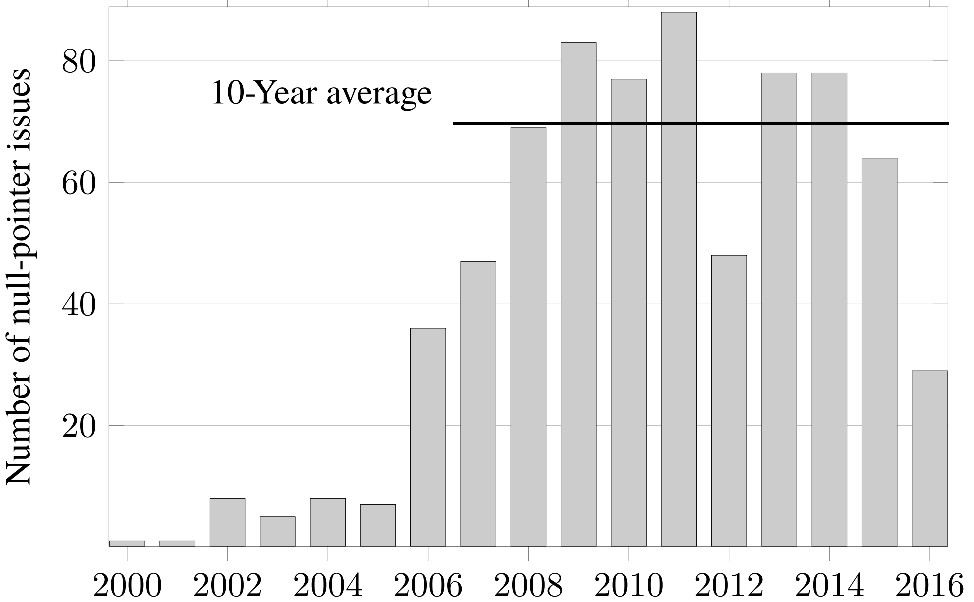


Fig. 1. Null pointer issues (such as null pointer dereferencing) in Common Vulnerabilities and Exposures database

A solution to this problem was proposed in [15] as an extension of the type system of Eiffel with a set of so called certified attachment patterns (CAP). Similar approach was proposed for Spec#, but was not adopted for inclusion into C# and the Common Runtime Language because of difficulties caused by required changes in the underlying platform and unsoundness of the prototype implementation ([2]). Indeed, it was discovered that a simple extension of the type system with "non-null" types does not allow for safe initialization of objects ([22]) and the void-safety property can be compromised. (In Eiffel null references are known as void references, hence the name "void safety". In this paper *void* and *null* are used interchangeably.) The same paper proposed a fix by introducing special annotations `[Free]` and `[Unclassified]` to source code and some new rules that should be checked by a compiler. Because the cases when the annotation is required are rare, an attempt to use a light-weight solution that does not require any additional annotation is implemented in [5]. Even though both approaches were shown to be usable, neither is accompanied with a formal machine-checkable soundness proof of the proposed type systems combined with additional restrictions placed on source code.

Current work formalizes the part related to certified attachment patterns, relaxes void safety rules for local intra-procedural context and proves that the rules ensure void safety with a generic proof assistant Isabelle/HOL.

2. Overview

Existing proposals ([2, 15, 22]) that address void safety issue in languages supporting null references use a type system extended with a notion of detachable and attached types for expressions that may and may not produce a null value. This extended type system is then uniformly applied to the language (e.g., [4]) meaning that types of variables are specified explicitly. This type information is then used to check reattachment validity rules. For example, an assignment $x := y$ would be valid only when a type of y conforms to a type of x. If the type of x is attached, the type of y should be attached as well.

This rule makes perfect sense for class attributes that can be accessed in different features. Type information is essential in that case because objects can be aliased at run-time and it would be impossible to do type checks at compile time modularly. However, for local variables there is no aliasing, or, more precisely, the locals can be changed only in a current feature. As a result it should be possible to get rid of type annotations altogether. It turns out that CAPs are absolutely sufficient for local variables and attachment annotations can be safely discarded.

The CAPs for local variables can also be applied to function's **Result**. For example, one can replace the code on the left with the code on the right:

```
foo: X
    do
        if attached something as r then
            Result := r
        else
            Result := something_else_attached
        end
    end
```

```
foo: X
    do
        Result := something
        if not attached Result then
            Result := something_else_attached
        end
    end
```

This allows not only for less code in new classes, but also for keeping original code unchanged if it follows this pattern.

Certified attachment patterns in [4] treat every boolean connective as a single use case: their combinations or nesting are not supported. Even though it might be a good practice to avoid complex expressions and to replace them with short and simple ones, when the first version of the compiler supporting void-safety was released, some users complained about missing cases. Moreover, complex expressions might be useful when code is not written but is generated automatically – then it can be arbitrary complex. This work addresses the demand by replacing arbitrary boolean connectives with conditional expressions and specifying CAPs in terms of these expressions. As a result, expressions of any complexity or nesting can be supported.

Even though simple branches and loop conditions were taken care by the original CAPs, the rules did not cover loop bodies. Simple analyses like definite assignment required by Java [7] can be done in one pass because on every iteration a variable can only become assigned, not the reverse. This does not work for void safety. A local variable can become attached or void on different iterations, or even to flip-flop on every iteration from attached state to detached and back as shown in the example on figure 2.

```
from
    x := something_attached
    y := Void
until
    whatever
loop
    tmp := y
    y := x
    x := tmp
end
x.foo
```

Fig. 2. Example of an issue with loop CAPs

If safety checks rely only on type declarations and *x* is of a detachable type, there are no guarantees that it will be attached after the loop (the original rules are quite pessimistic). As demonstrated by this work, the rules for loops could be based on fixed-point computation to meet program developer's expectations.

A set of CAPs specified in [4] do ensure void safety. However, they cannot be used in practice for any large scale application without provision for rules to escape void safety checks. It is just physically impossible to write 285 (or any other number of) classes in one go without intermediate compilation and testing. If at some point a feature is returning a value of a deferred class and there are no effective descendants of this class yet, the program will not compile. The solution adopted in [6] is to rely on exceptions, including forced checks of assertions. This triggers so called "design mode" when compiler ignores attachment status in type checks. Modeling the mode in the formalization essentially affects proofs but makes it possible to show soundness of real-life analysis.

To my knowledge this is the first attempt to formalize void safety rules and program semantics with attachment properties in a proof assistant environment. Moreover, this is the first time a formal void safety model is mechanically checked.

Presented formalization is done with big-step semantics style that is known to be suitable for proving preservation property, but to have issues with proving progress property. To address this, two different semantics are considered: void-unsafe and void-safe. It is demonstrated that both are equivalent as soon as void safety rules are

satisfied. A similar proof scheme can be applied to small-step semantics to prove progress property in that formalism if required.

In order to remain sound the formalization relies on attachment properties of expressions. Therefore, extending the approach to an inter-procedural context requires non-local void safety guarantees that can be achieved with a void-safe type system. This is done in Eiffel by augmenting its type system with **attached** and **detachable** marks added to type declarations and by specifying conformance and initialization rules that ensure an expression of an attached type always yields an object. Abstracting away language dissimilarities, the proposed rules can be used in other languages including mainstream ones as soon as they strengthen their type systems to be not only type-safe, but also null-safe.

All presented void safety rules are also implemented by me in the compiler [6] and are in production.

3. Formalization

Isabelle/HOL was successfully used in different projects starting from algebraic topology to verification of an operating system micro-kernel ([9]). It is build on top of a logic-neutral core called *Pure* with a specialized formalism of Higher-Order Logic (*HOL*). Talking about safety properties it was used to verify type soundness of JinjaThreads using operational semantics for concurrent execution of Java-like programs ([11, 12, 13]). Some decisions used in that formalization are adopted in the current work, some are new.

Even though selection of Isabelle/HOL is both voluntary (I knew it better) and traditional (it was used to formalize and prove type safety of Jinja), there are some other features that make it more attractive compared to other proof assistants:

- ability to write forward proofs in *Isar* language that makes reasoning closer to conventional textbooks;
- proof automation allowing for finding direct (i.e. not involving case analysis or induction) proofs automatically without diving into low-level details;
- built-in document preparation system enabling to type set all formulas (e.g. in this paper) directly from verified lemmas and preventing from using them for unfinished or failed proof scripts.

3.1 Translation of source language

Source language syntax is modeled in Isabelle/HOL with appropriate constructors of a datatype *expression* (figure 3). In most cases there is one-to-one relation between source language and Isabelle/HOL terms with two important points of divergence: one for voidness tests and the other one for operator expressions.

$expression =$

Value $value$	|	– Value (constant)
Local $name$	|	– Local variable
$expression$;; $expression$	|	– Sequence
$name ::= expression$	|	– Assignment
create $name$	|	– Creation instruction
$expression.name$ ($expression\ list$)	|	– Feature call
if $expression$ then $expression$ else $expression$ end	|	– Conditional expression
until $expression$ loop $expression$ end	|	– Loop
attached $type$ $expression$ as $name$	|	– Object test
Exception		– Exception

Fig. 3. Datatype expression

Voidness tests are source language expressions that check if a particular expression evaluates to *Void* at run-time or not:

expression /= Void

However, there is a more powerful construct that can be used instead of voidness tests: object tests. The most general form of an object test has 3 parts: a type, an expression and an object test variable:

attached {*SOME_TYPE*} *expression* **as** *my_variable*

The type is used to determine whether an expression is attached to an object of a type that conforms to the given one. If this is the case then the expression value is attached to the variable *my_variable* and the object test evaluates to **True**. Otherwise, it evaluates to **False**. The key observation here is that if the object test succeeds, both *expression* and *my_variable* are attached. Therefore, the type part of object test is irrelevant in most of the following discussion. When the type part is absent, the object test behaves like a regular voidness test. So the test *expression /= Void* is translated into

attached $None$ $expression$ as $unique_variable$

where *unique_variable* is a unique name not used anywhere else in the code.

The optional type part is still reflected in the formal semantics of the object test expression (see section 5.1).

3.2 Practical considerations

3.2.1 Design mode and unreachable code

As mentioned in section 2, there should be means to develop void-safe applications gradually. The most important issue is with features that take or return values of attached types. If there are no suitable effective classes yet, one cannot call such features or properly initialize their results. The idea to address such a need is to treat some code as unreachable. If the code is unreachable, there is no harm to skip void safety checks. In [6] the following constructs are used as indicators of unreachable code:

- enforced check: **check False then end**
- infinite loop: **from** ... **until False loop** ... **end**
- false postcondition: **ensure False**

Note that in general assertion checks are optional at run-time. However, to preserve soundness of void safety rules the assertion **ensure False** is always checked at run-time and triggers an exception. As a result, clients calling a feature with such a postcondition can rely on the fact that it never returns normally.

[12] proposes to model definite assignment property in presence of exceptions in Jinja with a type *set option*. A value *None* corresponds to an exceptional state and a value *Some x* – to a normal state. x is then a set of names of local variables that are definitely assigned. This approach perfectly works to model exceptional cases and unreachable code during attachment analysis too. But instead of using somewhat ad hoc rules to handle *set option*, a new type *topset* is introduced. It is obtained from a regular *set* type by adding a new top element. The operations are defined as shown in figure 4.

$$\begin{aligned} X \sqsubseteq \top &= True \\ \top \sqsubseteq \lceil A \rceil &= False \\ \lceil A \rceil \sqsubseteq \lceil B \rceil &= A \subseteq B \end{aligned} \qquad \begin{aligned} X \sqcup \top &= \top \\ \top \sqcup X &= \top \\ \lceil A \rceil \sqcup \lceil B \rceil &= \lceil A \cup B \rceil \end{aligned}$$

$$\begin{aligned} x \in^{\top} \top &= True \\ x \in^{\top} \lceil A \rceil &= x \in A \end{aligned} \qquad \begin{aligned} X \sqcap \top &= X \\ \top \sqcap X &= X \\ \lceil A \rceil \sqcap \lceil B \rceil &= \lceil A \cap B \rceil \end{aligned}$$

Fig. 4. Operations on topset.

The type *topset* is proved to be a complete lattice and a distributive lattice. These properties are essential in proofs involving fixed point of a transfer function (section 4.1).

Transitions of a local variable status from detachable to attached and back is modeled by two operations similar to insertion to a set and removal from a set. But neither insertion nor removal changes a top element $\top$:

$$A \oplus x = A \sqcup \lceil\{x\}\rceil \qquad \top \oplus x = \top$$

$$A \ominus x = A \sqcap \lceil\overline{\{x\}}\rceil \qquad \top \ominus x = \top$$

3.2.2 Operator transformation

A source code snippet, where a variable is considered attached because of a previous test to *Void* or when it is an object test local (see [4]), is called an attachment scope of this variable. The following kinds of scopes exist:

1. **Control flow scope –** an attachment scope based on language constructs that change execution flow.
2. **Operator scope** – an attachment scope based on semistrict boolean operators.

In practice both kinds of attachment scopes are applied together. An exhaustive list of scope combinations involving at most one unary and at most one binary boolean operator in a conditional instruction is given in figure 5.

1	**if**	**attached** *x*	**and**	...	**then**	...	**else**	...	**end**
	if	...	**and** [**then**]	**attached** *x*	**then**	...	**else**	...	**end**
2	**if**	**attached** *x*	**and then**	...	**then**	...	**else**	...	**end**
3	**if**	**not attached** *x*	**or**	...	**then**	...	**else**	...	**end**
	if	...	**or** [**else**]	**not attached** *x*	**then**	...	**else**	...	**end**
	if	...	**implies**	**not attached** *x*	**then**	...	**else**	...	**end**
4	**if**	**not attached** *x*	**or else**	...	**then**	...	**else**	...	**end**
	if	**attached** *x*	**implies**	...	**then**	...	**else**	...	**end**

Fig. 5. Scope combinations (code fragments where variable x is considered attached are marked with ...).

The language standard [4] specifies scopes of object test locals in terms of instructions and boolean operators, there are 8 clauses in total: 3 for expressions, 2 for conditional instructions and expressions, 1 for loops and 2 for assertion clauses. It might be tempting to mimic the rules in the logical framework and then prove that they are sound. But this approach has several drawbacks:

- The formalization would be limited to the selected set of boolean operators. Applying results to another language with different set of boolean operators would not be straightforward if some operators of that other language are not covered.
- There are 3 semistrict boolean operators, 2 regular operators and one unary operator. Adding them to the formalization would mean either addition of 6

new constructors to the datatype *expression* (figure 3) or addition of 2 new constructors to this datatype and introduction of new datatypes for operators with specific operator kinds. In both cases all induction-based proofs would have to be performed for new constructors.

- There is already some redundancy in the current operators because some of them can be expressed in terms of others using, for example, properly adapted De Morgan's laws.
- The rules as specified in the standard are not general enough and do not allow for deeper analysis of expressions. For example, they do not cover code like **if not not attached** *x* **then** ... **end** but cover its equivalent **if attached** *x* **then** ... **end**.

Generalization can be done with just 3 variants of expression: truth constants, conditional expressions and sequences. Every boolean expression can be translated into a conditional expression with nested boolean constants and optional sequences. The conversions of the boolean operators mentioned earlier and some others added for completeness are listed in figure 6. Following the terminology used in [4] they are called unfolded forms of boolean operators.

Operator	Original expression	Translation	
Negation	**not** *e*	**if** *e* **then False else True end**	(1)
Conjunction	*e1* **and then** *e2*	**if** *e1* **then** *e2* **else False end**	(2)
	e1 **and** *e2*	**if** *e1* **then** *e2* **else** *e2*; **False end**	(3)
Disjunction	*e1* **or else** *e2*	**if** *e1* **then True else** *e2* **end**	(4)
	e1 **or** *e2*	**if** *e1* **then** *e2*; **True else** *e2* **end**	(5)
Implication	*e1* **implies** *e2*	**if** *e1* **then** *e2* **else True end**	(6)
	not *e1* **or** *e2*	**if** *e1* **then** *e2* **else** *e2*; **True end**	(7)
Converse	**not** *e1* **and then** *e2*	**if** *e1* **then False else** *e2* **end**	(8)
nonimplication	**not** *e1* **and** *e2*	**if** *e1* **then** *e2*; **False else** *e2* **end**	(9)

Fig. 6. Unfolded forms of boolean operators

It turns out that all unfolded forms of boolean operators are variants of the following patterns:

if *x* **then** *y*; *Const* **else** *z* **end**
if *x* **then** *y* **else** *z*; *Const* **end**

where *Const* is either **True** or **False**. So instead of reasoning in terms of various forms of boolean operators and their combinations it is sufficient to reason in terms of special forms of conditional expressions. This approach does not only go beyond single-level scope definitions, but also allows for ternary operations in addition to unary and binary ones.

The special form of the branches ending with a boolean constant is captured by two functions defined in Isabelle/HOL as:

$$is_false\ (c\ ;;\ False_c) = True \qquad is_true\ (c\ ;;\ True_c) = True$$
$$is_false\ _ = False \qquad is_true\ _ = False$$

The cases from figure 6, when instead of an expression followed by a constant there is just a single constant **False** or **True**, can be represented by the sequences *unit ;;* $False_c$ or *unit ;;* $True_c$ respectively. It would be possible to handle constants **False** and **True** directly, however it would just add one more case in the function definitions without any additional benefit.

The functions *is_false* and *is_true* can be also generalized by adding other variants of expressions that knowingly produce fixed boolean constants, for example

$$is_true\ (if\ b\ then\ e_1\ else\ e_2\ end) = (if\ (is_true\ b)\ then\ is_true\ e_1)$$
$$\vee\ (if\ (is_false\ b)\ then\ is_true\ e_2) \vee (is_true\ e_1 \wedge is_true\ e_2)$$

This and other more complicated cases, however, are covered by optimization and code transformation techniques familiar from compiler technology, such as common sub-expression elimination, constant propagation, invariant code motion and others ([17, 18]).

Read-only scopes. General rules that define scopes are intermingled with the rules of an attachment status transfer function if (unlike [4]) the scopes are seen as means to determine potential attachment status of an arbitrary variable, not just a read-only one. For the sake of simplicity, consider the scopes of read-only variables first because they may be defined without bringing general attachment rules into play.

Scopes are defined for two cases: when an associated expression evaluates to **True** and when it evaluates to **False**:

Definition 3.1 (Scope function)**.** *A function that computes a set of read-only variables that are considered attached for an expression that evaluates to a particular boolean value is called a **scope function**.*

*A scope function for an expression e that evaluates to True (False) is called **positive (negative)** and is denoted* $+[e]$ *(*$-[e]$*).*

The rules to compute scope function are shown in figure 7. Only expressions that can produce non-empty sets of attached variables are listed. In all other cases the associated sets are empty. The notation $\lceil\{\ldots\}\rceil$ is used for sets adjusted to handle design mode (section 3.2.1).

$$+[\textit{attached t Local } n' \textit{ as } n] = \lceil \{n', n\} \rceil \qquad (10)$$

$$+[\textit{attached t e as } n] = \lceil \{n\} \rceil \qquad (11)$$

if e is not a variable

$$+[_] = \lceil \varnothing \rceil \qquad (12)$$

$$-[_] = \lceil \varnothing \rceil \qquad (13)$$

$$+[\textit{if } b \textit{ then } e_1 \textit{ else } e_2 \textit{ end}] = \begin{cases} -[b] \sqcup +[e_2] & \text{if } \textit{is_false } e_1 \\ +[b] \sqcup +[e_1] & \text{if } \textit{is_false } e_2 \\ \lceil \varnothing \rceil & \text{otherwise} \end{cases} \qquad (14)$$

$$-[\textit{if } b \textit{ then } e_1 \textit{ else } e_2 \textit{ end}] = \begin{cases} -[b] \sqcup -[e_2] & \text{if } \textit{is_true } e_1 \\ +[b] \sqcup -[e_1] & \text{if } \textit{is_true } e_2 \\ \lceil \varnothing \rceil & \text{otherwise} \end{cases} \qquad (15)$$

Fig. 7. Scope rules for read-only variables

The rules for object tests follow the explanations ealier: if an object test evaluates to $True_c$ (positive scope function), the corresponding object test variable is attached. Moreover, if the object test expression is a variable it is also known to be attached.

Two other cases cover general conditional expressions. Positive and negative scope functions recursively depend on each other. If a conditional expression does not evaluate to a boolean constant in at least one of its branches, nothing can be said about attachment status of object test locals involved in its sub-expressions. The reason is that information whether an object test succeeded, be it a conditional expression b or one of the branch expressions e_1 or e_2, is lost in that case.

Consider one of the cases when a branch expression meets a condition to produce a known constant value, for example, *is_false* e_1. Because this is the rule for positive scope function +[*if b then* e_1 *else* e_2 *end*], the computed sets correspond to the case when the conditional expression evaluates to $True_c$. From the condition *is_false* e_1 we know that b could not have been evaluated to $True_c$. Also, we know that the only case to get $True_c$ for the whole expression is to get $True_c$ for e_2. Therefore, a set of attached variables in that case is a union of the negative scope function for b and a positive scope function for e_2.

Other cases can be explained the same way. As an example let's see how the rules work for double negation:

	$+[$**not not attached** $x]$	
=	$+[$**if not attached** x **then False else True end**$]$	by (6)
=	$-[$**not attached** $x] \sqcup +[$**True**$]$	by (14)
=	$-[$**not attached** $x]$	by (12)
=	$-[$**if attached** x **then False else True end**$]$	by (6)
=	$+[$**attached** $x] \sqcup -[$**False**$]$	by (15)
=	$+[$**attached** $x]$	by (13)

What if for a given conditional expression both *is_false* e_1 and *is_false* e_2 would be true? Would the positive scope function yield a consistent result? For sub-expressions the function gives $+[e_1] = \lceil\varnothing\rceil$ and $+[e_2] = \lceil\varnothing\rceil$. So the result for the whole conditional is $+[b]$ and $-[b]$ at the same time that looks weird. The puzzle is solved by noticing that in this case the whole conditional expression evaluates to $False_c$ and does not fit the assumption that it produces $True_c$ (see definition 3.1).

4. Attachment properties

4.1 Transfer function

Given a set of attached variables A, a transfer function $A \triangleright e$ computes a set of attached variables for a given expression e. It is defined inductively as 5 mutually recursive functions:

$\cdot \rhd \cdot$ the transfer function itself (figure 8)

$\cdot \rhd\!+ \cdot$ $\cdot \rhd\!- \cdot$ computes a set of attached variables with an assumption that the expression evaluates to true/false (positive/negative scope) (figure 10)

$\cdot \rhd\rhd \cdot$ computes a set of attached variables for a given list of expressions (used to model arguments in feature calls) (figure 11)

$\cdot \hookrightarrow \cdot$ tells if a given expression is attached (figure 9)

$$
\begin{aligned}
& A \rhd \mathit{Value}\ v && = A \\
& A \rhd \mathit{Local}\ n && = A \\
& A \rhd e_1 \;;;\; e_2 && = A \rhd e_1 \rhd e_2 \\
& A \rhd \mathit{create}\ n && = A \oplus n \\
& A \rhd \mathit{attached}\ t\ e\ \mathit{as}\ n && = A \rhd e \\
& A \rhd \mathit{Exception} && = \top \\
& A \rhd n ::= e && = \begin{cases} (A \rhd e) \oplus n & \text{if } A \hookrightarrow e \\ (A \rhd e) \ominus n & \text{otherwise} \end{cases} \\
& A \rhd e\ .f\ (a) && = A \rhd e \rhd\rhd a \\
& A \rhd \mathit{if}\ c\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \mathit{end} && = A \rhd\!+ c \rhd e_1 \sqcap A \rhd\!- c \rhd e_2 \\
& A \rhd \mathit{until}\ e\ \mathit{loop}\ b\ \mathit{end} && = A \rhd\!* (- e \rhd b) \rhd\!+ e
\end{aligned}
$$

Fig. 8. Transfer function

Let's have a look at the most interesting cases. For an assignment a variable is added to a set of attached variables after the assignment if the source expression is attached and is removed from the initial set otherwise.

An attachment status of an expression is $True$ if it is a value other than $Void$, a local in the set of attached variables, or, a conditional expression with both branches attached (figure 9). Note that an attachment status of a conditional branch takes into account whether it is positive or negative.

$$
\begin{aligned}
& A \hookrightarrow \mathit{Value}\ v && = v \neq \mathit{Void}_v \\
& A \hookrightarrow \mathit{Local}\ n && = n \in^{\top} A \\
& A \hookrightarrow \mathit{if}\ c\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \mathit{end} && = A \rhd\!+ c \hookrightarrow e_1 \wedge A \rhd\!- c \hookrightarrow e_2 \\
& A \hookrightarrow _ && = \mathit{True}
\end{aligned}
$$

Fig. 9. Attachment status function

A similar formula is used for the transfer function on a conditional expression: one branch is evaluated with an assumption that a condition is true (the part $A \rhd\!+ b$) and the other one – when it is false (the part $A \rhd\!- b$).

A special value $\top$ is used for an exception to indicate that all variables can be safely considered as attached.

Positive and negative transfer functions (figure 10) differ from a regular transfer function only in two cases: for object tests and for conditional expressions. They mimic the scope function discussed in section 3.2.2.

$$A \rhd\!+\ \textit{attached T Local}\ n'\ \textit{as}\ n = A \sqcup \lceil\{n', n\}\rceil$$

$$A \rhd\!+\ \textit{attached T e as n} = A \rhd e \sqcup \lceil\{n\}\rceil \quad \text{if } e \text{ is not a variable}$$

$$A \rhd\!+\ \textit{if c then}\ e_1\ \textit{else}\ e_2\ \textit{end} = \begin{cases} A \rhd\!-\ b \rhd\!+\ e_2 & \text{if } \textit{is_false}\ e_1 \\ A \rhd\!+\ b \rhd\!+\ e_1 & \text{if } \textit{is_false}\ e_2 \\ A \rhd \textit{if b then}\ e_1\ \textit{else}\ e_2\ \textit{end} & \text{otherwise} \end{cases}$$

$$A \rhd\!-\ \textit{if c then}\ e_1\ \textit{else}\ e_2\ \textit{end} = \begin{cases} A \rhd\!-\ b \rhd\!-\ e_2 & \text{if } \textit{is_true}\ e_1 \\ A \rhd\!+\ b \rhd\!-\ e_1 & \text{if } \textit{is_true}\ e_2 \\ A \rhd \textit{if b then}\ e_1\ \textit{else}\ e_2\ \textit{end} & \text{otherwise} \end{cases}$$

$$A \rhd\!+\ e = A \rhd e$$

$$A \rhd\!-\ e = A \rhd e$$

Fig. 10. Transfer functions for positive and negative scopes

For a loop the transfer function is specified using a loop operator. A loop body is evaluated in a negative branch of an exit condition and the effect of the loop as a whole is evaluated in a positive branch of the same condition. The loop operator is defined as a greatest fixed point for $\lambda X. X \rhd\!-\ e \rhd b$ where e is an exit condition and b is a loop body:

$$A \rhd\!*\ (-\ e \rhd b) \equiv \textit{gfp}\ (\lambda X.\ A \sqcap X \rhd\!-\ e \rhd b)$$

The strange form of a loop function reflects what is implemented in the compiler. It reuses the same set of classes and functions for both conditional expressions and for loops. The transfer function for loops is then implemented by iterating over a loop until it stabilizes. A theorem from *HOL-Library* states that in this case the result is equal to the greatest fixed point. The theorem depends on monotonicity of the function. Firstly, observe that the loop function is monotone on both arguments, but we need only monotonicity on the last one:

Lemma 4.1 (Loop function monotonicity). $\textit{mono}\ f \implies \textit{mono}\ (\lambda X.\ A \sqcap f\ X)$

Then, instead of proving lemmas with a specific loop function, a generalized version can be used: $\textit{loop_operator}\ f\ A \equiv \textit{gfp}\ (\lambda x.\ A \sqcap f\ x)$. The loop operator is monotone and idempotent on both arguments:

Lemma 4.2 (Loop operator monotonicity). $\textit{mono}\ (\textit{loop_operator}\ f)$

Proof. From monotonicity of greatest fixed point. □

Lemma 4.3 (Loop operator unfolding).

$$mono\, f \Longrightarrow loop_operator\, f\, A = loop_function\, f\, A\ (loop_operator\, f\, A)$$

Lemma 4.4 (Loop operator idempotence).

$$mono\, f \Longrightarrow loop_operator\, f\ (loop_operator\, f\, x) = loop_operator\, f\, x$$

As one would expect, an application of a loop operator produces a smaller set of attached variables:

Lemma 4.5. $mono\, f \Longrightarrow$ $loop_operator\, f\, A \leq A$

$loop_operator\, f\, A \leq loop_operator\, f\ (f\, A)$

To conclude this section let's look at the rules for an expression list modeling arguments. Arguments of a call are subject to chained processing even though it might seem unnecessary. It turns out that an attachment status of an object test local could be affected because of the rules for "design mode" (section 3.2.1). The transfer function for every argument is evaluated in the context of a previous one (figure 11) or in the context of a target (for the first argument). The same effect can be achieved by using Isabelle/HOL function fold.

$$A \rhd\!\rhd [] \;=\; A \qquad\qquad A \rhd\!\rhd (e \cdot es) \;=\; A \rhd e \rhd\!\rhd es$$

Fig. 11. Transfer functions for argument lists

Lemma 4.6. $\mathrm{A} \rhd\!\rhd \mathrm{es} = \mathrm{fold}\ (\lambda\ \mathrm{e\ X.\ X} \rhd \mathrm{e})\ \mathrm{es\ A}$

According to [12] for subsequent proofs it is more convenient to use the direct definition of the transfer function rather than the one based on $fold$. This work adopts the same approach.

Here is an essential property of the transfer function that will be used later: it is monotonic. Intuitively this means that the more attached variables are known before an expression, the more there are after the expression:

Lemma 4.7 (Transfer function monotonicity). $mono\ (\lambda X.\ X \rhd c)$

Proof. By structural induction on all 5 mutually recursive function definitions. □

4.2 Expression validity

Validity rules are specified in Isabelle/HOL using an inductive predicate

$$\Gamma, A \vdash e : T$$

where Γ is an environment, A – a set of attached variables, e – an expression being checked, T – either $Attached$ or $Detachable$ – an attachment status of the expression e. $Attached$ means the expression produces a value that is not $Void$, $Detached$ means this value may be $Void$. If the predicate is true, the expression satisfies void safety rules in the given environment and attachment set and its type is T. The rules to compute predicate are shown in figure 12.

$$\frac{v \neq Void_v}{\Gamma, A \vdash Value\ v : Attached}\ \textsc{Value}_{att} \qquad \frac{v = Void_v}{\Gamma, A \vdash Value\ v : Detachable}\ \textsc{Value}_{det}$$

$$\frac{n \in^{\top} A}{\Gamma, A \vdash Local\ n : Attached}\ \textsc{Local}_{att} \qquad \frac{\neg\, n \in^{\top} A}{\Gamma, A \vdash Local\ n : Detachable}\ \textsc{Local}_{det}$$

$$\frac{}{\Gamma, A \vdash Exception : Attached}\ \textsc{Exception}$$

$$\frac{\Gamma, A \vdash e_1 : Attached \wedge \Gamma, A \rhd e_1 \vdash e_2 : Attached}{\Gamma, A \vdash e_1\ ;;\ e_2 : Attached}\ \textsc{Seq}$$

$$\frac{\Gamma, A \vdash e : T}{\Gamma, A \vdash n ::= e : Attached}\ \textsc{Assign} \qquad \frac{}{\Gamma, A \vdash create\ n : Attached}\ \textsc{Create}$$

$$\frac{\Gamma, A \vdash e : Attached \wedge \Gamma, A \rhd e \vdash a\ [:]\ Ts}{\Gamma, A \vdash e\ .\ f\ (a) : Attached}\ \textsc{Call}$$

$$\frac{\Gamma, A \vdash e : T}{\Gamma, A \vdash attached\ t\ e\ as\ n : Attached}\ \textsc{Test}$$

$$\frac{\Gamma, A \vdash b : Attached \wedge \Gamma, A \rhd\!+\, b \vdash e_1 : T_1 \wedge \Gamma, A \rhd\!-\, b \vdash e_2 : T_2}{\Gamma, A \vdash if\ b\ then\ e_1\ else\ e_2\ end : upper_bound\ T_1\ T_2}\ \textsc{If}$$

$$\frac{\Gamma, A \rhd\!*\, (-\ e \rhd b) \vdash e : Attached \wedge \Gamma, A \rhd\!*\, (-\ e \rhd b) \rhd\!-\, e \vdash b : Attached}{\Gamma, A \vdash until\ e\ loop\ b\ end : Attached}\ \textsc{Loop}$$

$$\frac{}{\Gamma, A \vdash []\ [:]\ []}\ \textsc{Arg}_{Nil} \qquad \frac{\Gamma, A \vdash e : T \wedge \Gamma, A \rhd e \vdash es\ [:]\ Ts}{\Gamma, A \vdash e \cdot es\ [:]\ T \cdot Ts}\ \textsc{Arg}_{Cons}$$

Fig. 12. Void safety rules

There are two rules for local variables: if a local variable name is in the set of attached variables, the corresponding expression is of an attached type ($\textsc{Local}_{att}$), otherwise it is of a detachable type ($\textsc{Local}_{det}$).

An attachment type of a conditional expression is computed as an upper bound of attachment types of both positive and negative branches ($\textsc{If}$). The upper bound is $Detachable$ if any of the operands is $Detachable$, and $Attached$ otherwise.

The validity predicate is properly defined, i.e. it cannot be true for attached and detachable types at the same time:

Lemma 4.8 (Attachment type uniqueness)**.** *A valid expression has one attachment type:* $\Gamma, A \vdash e : T \wedge \Gamma, A \vdash e : T' \Longrightarrow T = T'$

If an input set of attached variables becomes larger, computed attachment type for an expression may only become "more attached". Therefore, an attachment type computed for a larger attachment set conforms to the attachment type for a smaller one.

Lemma 4.9 (Attachment type monotonicity)**.**

$$A \leq B \wedge \Gamma, A \vdash e : T_A \implies \exists T_B.\ \Gamma, B \vdash e : T_B \wedge T_B \rightarrow_a T_A$$

Proof. The proof is done by structural induction on the predicate definition. It relies on monotonicity of transfer function (lemma 4.7) for compound expressions such as sequences and calls. For a conditional expression, types of both branches can be obtained thanks to lemma 4.8 and the resulting type will be computed as their upper bound, preserving monotonicity property. Validity of a loop expression follows from monotonicity of a loop operator (lemma 4.2). □

If a loop is valid in a given context, it is valid in a context obtained by a single or multiple application of the loop exit condition and loop body:

Lemma 4.10. *A loop remains valid after applying its transfer function to a set of attached variables one or any number of times:*

$$\Gamma, A \vdash \mathit{until}\ e\ \mathit{loop}\ c\ \mathit{end} : T \Longrightarrow \Gamma, A \rhd\!-\ e \rhd c \vdash \mathit{until}\ e\ \mathit{loop}\ c\ \mathit{end} : T$$

$$\Gamma, A \vdash \mathit{until}\ e\ \mathit{loop}\ c\ \mathit{end} : T \Longrightarrow \Gamma, A \rhd\!*\ (-\ e \rhd c) \vdash \mathit{until}\ e\ \mathit{loop}\ c\ \mathit{end} : T$$

Proof. Follows from monotonicity of transfer function and expression validity predicate (lemmas 4.7 and 4.9), idempotence of loop operator (lemma 4.4) and lemma 4.5. □

A notion of void-safe expressions is defined using the expression validity predicate with or without an associated context:

Definition 4.1 (Void-safe expression)**.** *An expression* e *is void-safe with type* T *in an environemnt* Γ *iff there is type that satisfies expression validity predicate with an empty set of attached variables:*

$$\Gamma \vdash e : T \equiv \Gamma, \lceil \varnothing \rceil \vdash e : T$$

An expression e *is void-safe in an environemnt* Γ *iff there is type* T *with which* e *is void-safe in an environment* Γ*:*

$$\Gamma \vdash e \surd_e \equiv \exists T.\ \Gamma \vdash e : T$$

In the context of null safety, if a local variable is considered attached at compile time, it should have an associated object at run-time. This property is captured by the notion of a valid state.

4.3 State validity

A state of a program is modeled by two functions: a function that maps local variable names to their value (a stack) and a function that maps memory addresses to object values (a heap). This work discusses only local variables, so the heap part can be arbitrary. Information about local variable types is available from an environment denoted in earlier formulas as Γ.

Definition 4.2. *A local state* l *is valid w.r.t. an environment* Γ *iff for every local in* Γ *the state* l *has a value for this local:*

$$\Gamma \vdash l \surd_s{}^E \equiv \forall\, \mathit{name}\ T.\ \Gamma\ \mathit{name} = \lfloor T \rfloor \longrightarrow (\exists\, v.\ l\ \mathit{name} = \lfloor v \rfloor)$$

Definition 4.3. *A local state* l *is valid w.r.t. an attachment set* A *iff for every local in* A *the state* l *has an attached value for this local provided that* A *is not* $\top$ *:*

$$A \vdash l \surd_s{}^A \equiv A \neq \top \longrightarrow (\forall\, n.\ n \in^{\top} A \longrightarrow (\exists\, v.\ l\ n = \lfloor v \rfloor \wedge v \neq \mathit{Void}_v))$$

Definition 4.4 (Void-safe state)**.** *For an environment* Γ *with an attachment set* A*, a state* (l, h) *is void-safe iff for any local variable name in* A *there is a local variable of this name in* l *attached to an object:*

$$\Gamma, A \vdash (l, h) \surd_s \equiv \Gamma \vdash l \surd_s{}^E \wedge A \vdash l \surd_s{}^A$$

For an environment Γ*, a state* s *is attachment-valid iff it is void-safe for* Γ *with an empty attachment set:*

$$\Gamma \vdash s \surd_s \equiv \Gamma, \lceil \varnothing \rceil \vdash s \surd_s$$

Most important properties of the state validity function are anti-monotonicity and what could be said about state validity if the corresponding attachment set changes:

Lemma 4.11 (Attachment state anti-monotonicity)**.**

$$B \leq A \wedge A \neq \top \wedge A \vdash l \surd_s{}^A \Longrightarrow B \vdash l \surd_s{}^A$$

Lemma 4.12. *Detaching, attaching and updating a local:*

$$A \vdash l \surd_s{}^A \implies A \ominus name \vdash l(name \mapsto value) \surd_s{}^A$$

$$value \neq Void_v \wedge A \vdash l \surd_s{}^A \implies A \oplus name \vdash l(name \mapsto value) \surd_s{}^A$$

$$value \neq Void_v \wedge A \vdash l \surd_s{}^A \implies A \vdash l(name \mapsto value) \surd_s{}^A$$

5. Local null safety

5.1 Big-step semantics

The big-step semantics is defined in Isabelle/HOL as an inductive predicate on transitions from an initial expression-state pair to a resulting one (figures 13 and 14). The rules are similar to those used in type system soundness proofs (e.g., [11, 12, 13]). The key differences are in the additional rules for object tests and in a modified rule for feature calls.

$$\frac{}{\Gamma \vdash \langle \mathit{Value}\ v,\ (l,\ m)\rangle \Rightarrow \langle \mathit{Value}\ v,\ (l,\ m)\rangle}\ \textsc{Value} \qquad \frac{l\ n = \lfloor v \rfloor}{\Gamma \vdash \langle \mathit{Local}\ n,\ (l,\ m)\rangle \Rightarrow \langle \mathit{Value}\ v,\ (l,\ m)\rangle}\ \textsc{Local}$$

$$\frac{\Gamma \vdash \langle e_1,\ s\rangle \Rightarrow \langle \mathit{unit},\ s'\rangle \wedge \Gamma \vdash \langle e_2,\ s'\rangle \Rightarrow \langle e_2{}',\ s''\rangle}{\Gamma \vdash \langle e_1\ ;;\ e_2,\ s\rangle \Rightarrow \langle e_2{}',\ s''\rangle}\ \textsc{Seq}$$

$$\frac{\Gamma \vdash \langle e,\ s\rangle \Rightarrow \langle \mathit{Value}\ v,\ (l,\ m)\rangle}{\Gamma \vdash \langle n ::= e,\ s\rangle \Rightarrow \langle \mathit{unit},\ (l(n \mapsto v),\ m)\rangle}\ \textsc{Assign}$$

$$\frac{\Gamma\ n = \lfloor T \rfloor \wedge \mathit{instance}\ m\ T = \lfloor (m',\ v) \rfloor}{\Gamma \vdash \langle \mathit{create}\ n,\ (l,\ m)\rangle \Rightarrow \langle \mathit{unit},\ (l(n \mapsto v),\ m')\rangle}\ \textsc{Create}$$

$$\frac{\Gamma\ n = \lfloor T \rfloor \wedge \mathit{instance}\ m\ T = \mathit{None}}{\Gamma \vdash \langle \mathit{create}\ n,\ (l,\ m)\rangle \Rightarrow \langle \mathit{Exception},\ (l,\ m)\rangle}\ \textsc{Create}_{fail}$$

$$\frac{\Gamma \vdash \langle e,\ s\rangle \Rightarrow \langle \mathit{Value}\ v,\ s_e\rangle \wedge v \neq \mathit{Void}_v \wedge \Gamma \vdash \langle es,\ s_e\rangle\ [\Rightarrow]\ \langle \mathit{map\ Value\ vs},\ s'\rangle}{\Gamma \vdash \langle e\ \boldsymbol{.}\ f\ (es),\ s\rangle \Rightarrow \langle \mathit{unit},\ s'\rangle}\ \textsc{Call}$$

$$\frac{\Gamma \vdash \langle b,\ s\rangle \Rightarrow \langle \mathit{True}_c,\ s'\rangle \wedge \Gamma \vdash \langle e_1,\ s'\rangle \Rightarrow \langle e_1{}',\ s''\rangle}{\Gamma \vdash \langle \mathit{if}\ b\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \mathit{end},\ s\rangle \Rightarrow \langle e_1{}',\ s''\rangle}\ \textsc{If}_{True}$$

$$\frac{\Gamma \vdash \langle b,\ s\rangle \Rightarrow \langle \mathit{False}_c,\ s'\rangle \wedge \Gamma \vdash \langle e_2,\ s'\rangle \Rightarrow \langle e_2{}',\ s''\rangle}{\Gamma \vdash \langle \mathit{if}\ b\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \mathit{end},\ s\rangle \Rightarrow \langle e_2{}',\ s''\rangle}\ \textsc{If}_{False}$$

$$\frac{\Gamma \vdash \langle e,\ s\rangle \Rightarrow \langle \mathit{True}_c,\ s'\rangle}{\Gamma \vdash \langle \mathit{until}\ e\ \mathit{loop}\ b\ \mathit{end},\ s\rangle \Rightarrow \langle \mathit{unit},\ s'\rangle}\ \textsc{Loop}_{True}$$

$$\frac{\begin{array}{c}\Gamma \vdash \langle e,\ s\rangle \Rightarrow \langle \mathit{False}_c,\ s_e\rangle\ \wedge \Gamma \vdash \langle b,\ s_e\rangle \Rightarrow \langle \mathit{unit},\ s_c\rangle \wedge \\ \Gamma \vdash \langle \mathit{until}\ e\ \mathit{loop}\ b\ \mathit{end},\ s_c\rangle \Rightarrow \langle c',\ s'\rangle\end{array}}{\Gamma \vdash \langle \mathit{until}\ e\ \mathit{loop}\ b\ \mathit{end},\ s\rangle \Rightarrow \langle c',\ s'\rangle}\ \textsc{Loop}_{False}$$

$$\frac{\Gamma \vdash \langle e,\ s\rangle \Rightarrow \langle \mathit{Value}\ v,\ (l,\ m)\rangle \wedge v \neq \mathit{Void}_v \wedge v\ \mathit{has_type}\ T}{\Gamma \vdash \langle \mathit{attached}\ T\ e\ \mathit{as}\ n,\ s\rangle \Rightarrow \langle \mathit{True}_c,\ (l(n \mapsto v),\ m)\rangle}\ \textsc{Test}_{True}$$

$$\frac{\Gamma \vdash \langle e,\ s\rangle \Rightarrow \langle \mathit{Value}\ v,\ (l,\ m)\rangle \wedge \neg\ (v \neq \mathit{Void}_v \wedge v\ \mathit{has_type}\ T)}{\Gamma \vdash \langle \mathit{attached}\ T\ e\ \mathit{as}\ n,\ s\rangle \Rightarrow \langle \mathit{False}_c,\ (l,\ m)\rangle}\ \textsc{Test}_{False}$$

$$\frac{}{\Gamma \vdash \langle [],\ s\rangle\ [\Rightarrow]\ \langle [],\ s\rangle}\ \textsc{Arg}_{Nil} \qquad \frac{\Gamma \vdash \langle e,\ s\rangle \Rightarrow \langle \mathit{Value}\ v,\ s_e\rangle \wedge \Gamma \vdash \langle es,\ s_e\rangle\ [\Rightarrow]\ \langle es',\ s'\rangle}{\Gamma \vdash \langle e \cdot es,\ s\rangle\ [\Rightarrow]\ \langle \mathit{Value}\ v \cdot es',\ s'\rangle}\ \textsc{Arg}_{Cons}$$

Fig. 13. Big-step semantics: regular cases

An object test evaluates to $True$ if its object test expression is attached and is of an expected type ($\textsc{Test}_{True}$). In that case the local storage is updated for an object test local to have a computed value. The specification uses an abstract function has_type that is not instantiated. Therefore, all the proofs do not depend on the actual run-time type check.

If any of the conditions is not met, e.g. a value is void or its type is not expected, the state is not changed and the object test evaluates to $False$ ($\textsc{Test}_{False}$).

Note that there is only one (non-exceptional) rule for a feature call. This is the major difference from traditional big-step semantics specifications. What if a target of a call will be $Void$? Would not it mean that excution may be stuck? The answer is given in section 5.3.

Exception propagation rules (figure 14) are not different from the rules used in type soundness proofs.

$$\frac{}{\Gamma \vdash \langle \mathit{Exception}, s\rangle \Rightarrow \langle \mathit{Exception}, s\rangle}\ \textsc{Exception} \qquad \frac{\Gamma \vdash \langle e_1, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle}{\Gamma \vdash \langle e_1 \;;;\; e_2, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle}\ \textsc{Seq}_{ex}$$

$$\frac{\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle}{\Gamma \vdash \langle n ::= e, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle}\ \textsc{Assign}_{ex}$$

$$\frac{\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle}{\Gamma \vdash \langle e \,.\, f\ (es), s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle}\ \textsc{Call}_{ex}$$

$$\frac{\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{Value}\ v, s_e\rangle \wedge \Gamma \vdash \langle es, s_e\rangle\ [\Rightarrow]\ \langle \mathit{map\ Value\ vs}\ @\ (\mathit{Exception} \cdot es'), s'\rangle}{\Gamma \vdash \langle e \,.\, f\ (es), s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle}\ \textsc{Call}_{Arg-ex}$$

$$\frac{\Gamma \vdash \langle b, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle}{\Gamma \vdash \langle \mathit{if\ b\ then\ e_1\ else\ e_2\ end}, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle}\ \textsc{If}_{ex}$$

$$\frac{\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle}{\Gamma \vdash \langle \mathit{attached\ t\ e\ as\ n}, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle}\ \textsc{Test}_{ex} \qquad \frac{\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle}{\Gamma \vdash \langle e \cdot es, s\rangle\ [\Rightarrow]\ \langle \mathit{Exception} \cdot es, s'\rangle}\ \textsc{Arg}_{ex}$$

$$\frac{\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle}{\Gamma \vdash \langle \mathit{until\ e\ loop\ b\ end}, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle}\ \textsc{Loop}_{ex}$$

$$\frac{\Gamma \vdash \langle e, s\rangle \Rightarrow \langle \mathit{False}_c, s_e\rangle \wedge \Gamma \vdash \langle b, s_e\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle}{\Gamma \vdash \langle \mathit{until\ e\ loop\ b\ end}, s\rangle \Rightarrow \langle \mathit{Exception}, s'\rangle}\ \textsc{Loop}_{False\text{-}ex}$$

Fig. 14. Big-step semantics: exception propagation

Conventionally the big-step semantics is shown to end up for a given expression in a final state, meaning an exception or a value.

Definition 5.1 (Final expression)**.** *An expression is called final if it is an exception or a value:*

$$Final\ e \equiv e = Exception \vee \exists v.\ e = Value\ v$$

Lemma 5.1 (Finality of big-step semantics)**.** *If there is a big-step transition for an expression* e *from state* s *to an expression* e' *and state* s' *then* e' *is final:*

$$\Gamma \vdash \langle e, s\rangle \Rightarrow \langle e', s'\rangle \Longrightarrow Final\ e'$$

5.2 Preservation theorem

Anti-monotonicity of attachment state allows to prove that as soon as a state is valid in one of the branches of a conditional expression, it is valid for the expression as a whole. Intuitively there is more information in one branch of a conditional expression and therefore there are more attached variables, so if a state is valid for one branch it is valid for the whole expression with less attached variables.

Lemma 5.2. *If a local state* l *is valid in a context of either branch of a conditional expression, it is valid in the context of the whole expression:*

$$A \rhd\!+\ c \rhd e_1 \vdash l \surd_s{}^A \wedge A \rhd\!+\ c \rhd e_1 \neq \top \implies A \rhd \mathit{if\ c\ then\ e_1\ else\ e_2\ end} \vdash l \surd_s{}^A$$
$$A \rhd\!-\ c \rhd e_2 \vdash l \surd_s{}^A \wedge A \rhd\!-\ c \rhd e_2 \neq \top \implies A \rhd \mathit{if\ c\ then\ e_1\ else\ e_2\ end} \vdash l \surd_s{}^A$$

Proof. Follows from the definition of transfer function and lemma 4.11.□

The big-step semantics preserves valid state for both exceptional (denoted by $\top$, see section 3.2.1) and non-exceptional attachment sets (denoted by $\lceil a \rceil$):

Lemma 5.3. $\Gamma \vdash \langle e, s\rangle \Rightarrow \langle e', s'\rangle \wedge \Gamma, \top \vdash s \surd_s \Longrightarrow \Gamma, \top \vdash s' \surd_s$

Lemma 5.4. $\Gamma \vdash \langle e, s\rangle \Rightarrow \langle e', s'\rangle \wedge \Gamma, \lceil a \rceil \vdash s \surd_s \Longrightarrow \exists b.\ \Gamma, \lceil b \rceil \vdash s' \surd_s$

From the other hand, if a final expression (definition 5.1) is not an exception, an attachment set for the initial expression remains non-exceptional:

Lemma 5.5.

$$\Gamma \vdash \langle e, s\rangle \Rightarrow \langle e', s'\rangle \wedge \Gamma, A \vdash e : T \wedge e' \neq \mathit{Exception} \wedge A \neq \top \Longrightarrow A \rhd e \neq \top$$

Proof. By structural induction on big-step semantics predicate for all mutually recursive transfer functions. □

The main result of this section is an attachment preservation theorem telling that if an expression is void-safe and its evaluation starts in a void-safe state and completes, then it either results in an exception or in a value that is not void if the expression type is attached. The following lemma states this formally.

Lemma 5.6 (Attachment preservation step)**.**

$$\Gamma, A \vdash s \surd_s \wedge A = \lceil a \rceil \wedge \Gamma \vdash \langle e, s\rangle \Rightarrow \langle e', s'\rangle \wedge \Gamma, A \vdash e : T \wedge e' \neq \mathit{Exception} \Longrightarrow$$
$$\exists T'.\ \Gamma, A \rhd e \vdash e' : T' \wedge T' \rightarrow_a T$$

Proof. The proof is done by structural induction on big-step semantics predicate and uses lemmas 4.12, 5.1 and 5.2. For every induction case is shows that a state remains valid using lemmas 5.3 and 5.4 and taking into account preservation of non-exceptional attachment set (lemma 5.5) and applying an inductive hypothesis to finish the proof. □

Replacing variables with initial state values, the lemma gives:

Theorem 5.1 (Attachment preservation)**.**

$$\Gamma \vdash \langle e, \varnothing\rangle \Rightarrow \langle e', s'\rangle \wedge \Gamma \vdash e : \mathit{Attached} \Longrightarrow$$
$$e' = \mathit{Exception} \vee (\exists v.\ e' = \mathit{Value}\ v \wedge v \neq \mathit{Void}_v)$$

5.3 Equivalence of safe and unsafe semantics

Ideally void safety should be a corollary of two theorems: preservation and progress. The third one – determinism – cannot be proved in most concurrent environments, so it is not considered here. Unfortunately a progress theorem cannot be proved with classical big-step semantics because it deals only with final states (lemma 5.1). Therefore, it is impossible to describe intermediate states. At least two options exist:

- Use clocked big-step semantics [20] or similar abstraction that distinguishes between stuck state and divergence (e.g., [1, 19, 21]).
- Use small-step semantics.

The first option is straightforward: the current rules can be adapted to distinguish between stuck state and divergence. The main drawback is missing support for concurrency that cannot be easily expressed with big-step semantics.

The second option is more attractive because it allows for proving a progress theorem directly. Unfortunately, it is not applicable to the current formalization because it does not provide type information. In the big-step rules for conditional expressions and loops the semantics is specified with an assumption that branch or exit conditions are evaluated to a boolean value, i.e. no initial or intermediate type

information is required to state the rule. For small-step semantics this is not the case: for intermediate steps to be sound we need to know that these intermediate steps preserve the property that the expression type is boolean.

Can the requirement to have type information in the semantics rules be avoided, so that only the part of interest is kept for consideration? Here is an idea. Let's assume that the type system is sound. Then both type preservation and progress theorems are true w.r.t. the associated small-step semantics. Assume that the original semantics is specified not taking void safety into account. Then consider a semantics that expects void safety. If both, void-safety aware and void-safety unaware, semantics can be shown to be equivalent for programs that satisfy void safety rules, preservation and progress theorems can be derived for the void-safe semantics from their void-unsafe counterparts.

The approach can be demonstrated with big-step semantics as well. To this end two semantics definitions are considered. The void-safe version is the one described in section 5.1. The void-unsafe version differs from the safe one just by a single rule. The rule makes sure that if in a void-unsafe program a target of a call is *Void* , an exception is raised (figure 15). The exception here is the famous `NullPointerException`.

Safe:
$$\frac{\Gamma \vdash \langle e, s\rangle \Rightarrow \langle Value\ v, s_e\rangle \wedge v \neq Void_v \wedge \Gamma \vdash \langle es, s_e\rangle\ [\Rightarrow]\ \langle map\ Value\ vs, s'\rangle}{\Gamma \vdash \langle e\ .\ f\ (es), s\rangle \Rightarrow \langle unit, s'\rangle}\ \textsc{Call}$$

Unsafe:
$$\frac{\Gamma \vdash \langle e, s\rangle \Rightarrow' \langle Value\ v, s_e\rangle \wedge v \neq Void_v \wedge \Gamma \vdash \langle es, s_e\rangle\ [\Rightarrow]'\ \langle map\ Value\ vs, s'\rangle}{\Gamma \vdash \langle e\ .\ f\ (es), s\rangle \Rightarrow' \langle unit, s'\rangle}\ \textsc{Call}^{unsafe}$$

$$\frac{\Gamma \vdash \langle e, s\rangle \Rightarrow' \langle Value\ v, s'\rangle \wedge v = Void_v}{\Gamma \vdash \langle e\ .\ f\ (es), s\rangle \Rightarrow' \langle Exception, s'\rangle}\ \textsc{Call}^{unsafe}_{fail}$$

Fig. 15. Feature call rule in safe and unsafe big-step semantics

It turns out that if an expression is null-safe, it gives exactly the same result regardless of the semantics behind. This effectively demonstrates absence of `NullPointerException` in null-safe programs.

Theorem 5.2 (Semantics equivalence)**.** *Void-safe ($\Rightarrow$) and void-unsafe ($\Rightarrow'$) semantics of a void-safe program with an initial void-safe state are equivalent:*

$$\Gamma \vdash e \surd_e \wedge \Gamma \vdash s_0 \surd_s \implies \Gamma \vdash \langle e, s_0\rangle \Rightarrow' \langle v, s\rangle = \Gamma \vdash \langle e, s_0\rangle \Rightarrow \langle v, s\rangle$$

5.4 Practical results

The core part of the local code analysis described in the paper is implemented in 19 Eiffel classes of about 2.5KLOC in total. Instead of immutable attachment sets it uses mutable ones, optimized with bitwise operations. Branching instructions, such as loops, and conditional instructions and expressions, share the same code base that explains a form of the loop transfer function (section 4.1). All code is open source and is available at https://dev.eiffel.com/Source_Code.

First void safety checks were added to Eiffel in *EiffelStudio 6.3* at the time when public libraries almost reached a million lines of code. Migration of public libraries took several releases and is still an ongoing work for few remaining libraries that are not completely void-safe.

The current work suggests relaxed rules for local variable declarations and **Result** where no special type annotations are required. All attachment information is derived by static code analysis. This analysis was implemented in [6]. In contrast to previous releases, no changes to public libraries were required. This confirms theoretical soundness of the approach in practice. The rules also remove unneeded annotation burden from programmers and allow for simpler code.

All theories code is proved with *Isabelle 2016* and is available at https://bitbucket.org/kwaxer/void_safety/ (tag 1.2.2).

6. Related work

Three key ingredients of void safety – a type system that allows for specifying attachment properties, certified attachment patterns that allow for expressions of otherwise detachable type to be used when attached values are expected, and validity rules for safe reattachment and initialization – were already used by [4] in 2006. Research efforts were then mostly focused on making sure the typing rules do ensure soundness: [15] explains why the rules work, [2] reports issues with object initialization in a naïve implementation, [22] proposes a solution that requires introduction of new concepts into the type system to represent partially initialized objects with explicit additional annotations, [14] discusses how additional annotations can be avoided if strict modularity is not required. None of the papers went into the details of usability and user experience, inspecting if the rules are reasonable for real-life cases. Nor did the papers discuss how to perform the development of large systems where a complete set of classes is not readily available. This work addresses the first issue by proposing a set of rules for local variables and demonstrates that all the required information for them can be derived from the code. The second issue is solved by changing validity rules to respect exceptional behavior at run-time.

Type system soundness of conventional object-oriented languages became a hot research area with release of Java that claimed to be absolutely type-safe (cf. [10] that explicitly states undefined behavior in certain cases). [12] presented a formal proof for a subset of Java in Isabelle/HOL using big-step semantics. Unfortunately big-step semantics is not good for reasoning about concurrent programs. [13] updated the proof to use small-step semantics instead and formalized Java memory model. The current work focuses on void safety rules introduced in Eiffel. Its concurrency model is quite different from Java. Even though [16] formalized the semantics in Maude, its correctness is not formally proved. So, the work uses big-step semantics to describe and to reason about void safety guarantees.

An algorithm to compute a set of attached variables might seem to be quite similar to definite assignment rules of [7] and formalized by [12]. However, it differs in

several important aspects. Contemporary definite assignment and presented here transfer functions do take into account context of branches with different outcome of preceding conditions, while formalization in [12, 13] does not. Moreover, a set of definitely assigned variables does not depend on initial set of variables. Such a set is useless because an uninitialized variable cannot be used as a source of a reattachment. This is different for void safety. Both an attached or detachable variable can be used as a source or as a target of a reattachment. Finally, described here void safety rules rely on computation of greatest fixed points for loops. This is not needed for definite assignment computation. More similarity with type soundness proofs can be seen in monotonicity of a transfer function, though for void safety this property is also essential for showing that validity rules can be programmatically checked by a compiler.

Leaving concurrency aside, big-step semantics does not distinguish between stuck and diverging states. [20, 21] demonstrate how to deal with that, so the big-step semantics could be changed accordingly. Instead, this work shows that safe and unsafe versions of big step semantics become equivalent when void safety rules are met.

7. Conclusion

Certified attachment patterns are an essential part of void safety guarantees in modern OO languages. This work formalizes them in Isabelle/HOL and proves some safety properties w.r.t. big-step semantics. The novelty of the work is in:

- Generalization of attachment rules for boolean operators. (In particular similar scheme can be used to adapt definite assignment rules described in [12] and later used in [13] to prove Java-like type system soundness to match current Java rules [7].)
- Introduction of "design mode" to void safety rules required to analyze real-world programs.
- Specification of void safety rules for loops.
- Demonstration of theoretical and practical uselessness of attachment annotations for local variables.
- Mechanically verified preservation theorem for void-safe programs and conditional equivalence of void-safe and void-unsafe semantics that can be applied to show complete void safety (both for local contexts only).

The work covers only one part of three key elements of a void-safe language. It needs to be extended to deal with the other two, namely:

- Type system – required to show safety with regular feature calls.
- Initialization – required to show safety for object creation when some objects are in an intermediate half-initialized state.

References

[1]. Nada Amin and Tiark Romp. "Type Soundness Proofs with Definitional Interpreters". In: *OOPSLA*. 2016. url: https://www.cs.purdue.edu/homes/ rompf/papers/amin-draft2016a.pdf. Submitted.

[2]. Mike Barnett et al. "Specification and Verification: The Spec# Experience". In: *Commun. ACM* 54.6 (June 2011). Ed. by Moshe Y. Vardi, pp. 81–91. issn: 0001-0782. doi: 10.1145/1953122.1953145.

[3]. *Common Vulnerabilities and Exposures*. http://cve.mitre.org/. 2016. (Visited on 2016-05-25).

[4]. Ecma International. *ECMA-367: Eiffel analysis, design and programming language*. 2nd. Geneva, Switzerland: Ecma International, June 2006. url: http://www.ecma-international.org/publications/standards/Ecma367.htm.

[5]. Eiffel Software. *EiffelStudio 7.3 Releases*. https://dev.eiffel.com/ EiffelStudio_7.3_Releases. July 2013. (Visited on 2016-05-14).

[6]. Eiffel Software. *EiffelStudio 16.05 Releases*. https://dev.eiffel.com/ EiffelStudio_16.05_Releases. July 2016. (Visited on 2016-09-15).

[7]. James Gosling et al. *The Java Language Specification, Java SE 8 Edition*. 1st. Addison-Wesley Professional, 2014. isbn: 9780133900699.

[8]. Tony Hoare. "Null references: The billion dollar mistake". In: *Presentation at QCon London* (2009).

[9]. *Projects – Isabelle Community wiki*. https://isabelle.in.tum.de/ community/Projects. Apr. 2016. (Visited on 2016-09-15).

[10]. ISO. *ISO/IEC 14882:2014(E): Information technology — Programming languages — C++*. 4th. Geneva, Switzerland: International Organization for Standardization, Dec. 15, 2014.

[11]. Gerwin Klein. "Verified Java Bytecode Verification". PhD thesis. Institut für Informatik, Technische Universität München, 2003. url: http://www4.in. tum.de/~kleing/diss/.

[12]. Gerwin Klein and Tobias Nipkow. "A Machine-checked Model for a Java-like Language, Virtual Machine, and Compiler". In: *ACM Trans. Program. Lang. Syst.* 28.4 (July 2006), pp. 619–695. issn: 0164-0925. doi: 10.1145/1146809.1146811.

[13]. Andreas Lochbihler. "A Machine-Checked, Type-Safe Model of Java Concurrency : Language, Virtual Machine, Memory Model, and Verified Compiler". PhD thesis. Karlsruher Institut für Technologie, Fakultät für Informatik, July 2012. doi: 10.5445/KSP/1000028867. url: http://digbib.ubka.unikarlsruhe.de/volltexte/1000028867.

[14]. Bertrand Meyer. *Targeted expressions: safe object creation with void safety*. http://se.ethz.ch/~meyer/publications/online/targeted.pdf. July 2012. (Visited on 2016-09-24).

[15]. Bertrand Meyer, Alexander Kogtenkov, and Emmanuel Stapf. "Avoid a Void: The Eradication of Null Dereferencing". In: *Reflections on the Work of C.A.R. Hoare*. Ed. by A.W. Roscoe, Cliff B. Jones, and Kenneth R. Wood. History of Computing. Springer London, 2010, pp. 189–211. isbn: 978-1-84882-912-1. doi: 10.1007/978-1-84882-912-1_9.

[16]. Benjamin Morandi et al. "Prototyping a Concurrency Model". In: *Proceedings of the 2013 13th International Conference on Application of Concurrency to System Design*. ACSD '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 170–179. isbn:

978-0-7695-5035-0. doi: 10.1109/ACSD.2013.21. url: http://dx.doi.org/10.1109/ACSD.2013.21.

[17]. Robert Morgan. *Building an Optimizing Compiler*. Newton, MA, USA: Digital Press, 1998. isbn: 1-55558-179-X.

[18]. Steven S. Muchnick. *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. isbn: 1-55860320-4.

[19]. Scott Owens et al. “Functional Big-Step Semantics”. In: *Programming Languages and Systems: 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Ed. by Peter Thiemann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 589–615. isbn: 978-3-662-49498-1. doi: 10.1007/978-3-662-494981_23. url: http://dx.doi.org/10.1007/978-3-662-49498-1_23.

[20]. Jeremy Siek. *Big-step, diverging or stuck?* http://siek.blogspot.ch/2012/07/big-step-diverging-or-stuck.html. July 2012. (Visited on 2016-09-15).

[21]. Jeremy Siek. *Type Safety in Three Easy Lemmas*. http://siek.blogspot.ch/2013/05/type-safety-in-three-easy-lemmas.html. May 2013. (Visited on 2016-09-15).

[22]. Alexander J. Summers and Peter Müller. “Freedom Before Commitment: A Lightweight Type System for Object Initialisation”. In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. Ed. by . OOPSLA ’11. Portland, Oregon, USA: ACM,2011,pp.1013–1032.isbn:978-1-4503-0940-0.doi:10.1145/2048066.2048142.

Автоматическое доказательство безопасности локальных пустых указателей

А.В. Когтенков <kwaxer@mail.ru>
ETH Zürich
8092 Цюрих, Швейцария, Universitätstrasse 19

Аннотация. Разыменование пустого указателя – это хорошо известная ошибка, встречающаяся в объектно-ориентированных программах. Ее можно избежать путем добавления к языку, на котором пишется программа, специальных правил приложимости. Достаточно ли этих правил для гарантии отсутствия таких исключительных ситуаций? Данная статья посвящена безопасности пустых указателей во внутрипроцедурном контексте, в котором не требуются какие-либо дополнительные аннотации. Правила формализуются в системе автоматического доказательства теорем Isabelle/HOL. Затем доказывается теорема о сохранении безопасности пустых указателей в крупношаговой семантике. Наконец, демонстрируется, что при наличии таких правил семантики с безопасностью пустых указателей и без нее эквивалентны.

Ключевые слова: безопасность пустых указателей; статический анализ; Eiffel; формальные методы; крупношаговая операционная семантика; теорема о сохранении; эквивалентность операционных семантик

DOI: 10.15514/ISPRAS-2016-28(5)-2

Для цитирования: Когтенков А.В. Автоматическое доказательство безопасности локальных пустых указателей. Труды ИСП РАН, том 28, вып. 5, 2016, стр.. 27-54 (на английском). DOI: 10.15514/ISPRAS-2016-28(5)-2

Список литературы

[1]. Nada Amin and Tiark Romp. “Type Soundness Proofs with Definitional Interpreters”. In: *OOPSLA*. 2016. url: https://www.cs.purdue.edu/homes/ rompf/papers/amin-draft2016a.pdf. Submitted.

[2]. Mike Barnett et al. “Specification and Verification: The Spec# Experience”. In: *Commun. ACM* 54.6 (June 2011). Ed. by Moshe Y. Vardi, pp. 81–91. issn: 0001-0782. doi: 10.1145/1953122.1953145.

[3]. *Common Vulnerabilities and Exposures*. http://cve.mitre.org/. 2016. (Visited on 2016-05-25).

[4]. Ecma International. *ECMA-367: Eiffel analysis, design and programming language*. 2nd. Geneva, Switzerland: Ecma International, June 2006. url: http://www.ecma-international.org/publications/standards/Ecma367.htm.

[5]. Eiffel Software. *EiffelStudio 7.3 Releases*. https://dev.eiffel.com/ EiffelStudio_7.3_Releases. July 2013. (Visited on 2016-05-14).

[6]. Eiffel Software. *EiffelStudio 16.05 Releases*. https://dev.eiffel.com/ EiffelStudio_16.05_Releases. July 2016. (Visited on 2016-09-15).

[7]. James Gosling et al. *The Java Language Specification, Java SE 8 Edition*. 1st. Addison-Wesley Professional, 2014. isbn: 9780133900699.

[8]. Tony Hoare. “Null references: The billion dollar mistake”. In: *Presentation at QCon London* (2009).

[9]. *Projects – Isabelle Community wiki*. https://isabelle.in.tum.de/ community/Projects. Apr. 2016. (Visited on 2016-09-15).

[10]. ISO. *ISO/IEC 14882:2014(E): Information technology — Programming languages — C++*. 4th. Geneva, Switzerland: International Organization for Standardization, Dec. 15, 2014.

[11]. Gerwin Klein. “Verified Java Bytecode Verification”. PhD thesis. Institut für Informatik, Technische Universität München, 2003. url: http://www4.in.tum.de/~kleing/diss/.

[12]. Gerwin Klein and Tobias Nipkow. “A Machine-checked Model for a Java-like Language, Virtual Machine, and Compiler”. In: *ACM Trans. Program. Lang. Syst*. 28.4 (July 2006), pp. 619–695. issn: 0164-0925. doi: 10.1145/1146809.1146811.

[13]. Andreas Lochbihler. “A Machine-Checked, Type-Safe Model of Java Concurrency : Language, Virtual Machine, Memory Model, and Verified Compiler”. PhD thesis. Karlsruher Institut für Technologie, Fakultät für Informatik, July 2012. doi: 10.5445/KSP/1000028867. url: http://digbib.ubka.unikarlsruhe.de/volltexte/1000028867.

[14]. Bertrand Meyer. *Targeted expressions: safe object creation with void safety*. http://se.ethz.ch/~meyer/publications/online/targeted.pdf. July 2012. (Visited on 2016-09-24).

[15]. Bertrand Meyer, Alexander Kogtenkov, and Emmanuel Stapf. “Avoid a Void: The Eradication of Null Dereferencing”. In: *Reflections on the Work of C.A.R. Hoare*. Ed. by A.W. Roscoe, Cliff B. Jones, and Kenneth R. Wood. History of Computing. Springer London, 2010, pp. 189–211. isbn: 978-1-84882-912-1. doi: 10.1007/978-1-84882-912-1_9.

[16]. Benjamin Morandi et al. “Prototyping a Concurrency Model”. In: *Proceedings of the 2013 13th International Conference on Application of Concurrency to System Design*. ACSD ’13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 170–179. isbn: 978-0-7695-5035-0. doi: 10.1109/ACSD.2013.21. url: http://dx.doi.org/10.1109/ACSD.2013.21.

[17]. Robert Morgan. *Building an Optimizing Compiler*. Newton, MA, USA: Digital Press, 1998. isbn: 1-55558-179-X.

[18]. Steven S. Muchnick. *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. isbn: 1-55860320-4.

[19]. Scott Owens et al. “Functional Big-Step Semantics”. In: *Programming Languages and Systems: 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Ed. by Peter Thiemann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 589–615. isbn: 978-3-662-49498-1. doi: 10.1007/978-3-662-494981_23. url: http://dx.doi.org/10.1007/978-3-662-49498-1_23.

[20]. Jeremy Siek. *Big-step, diverging or stuck?* http://siek.blogspot.ch/2012/07/big-step-diverging-or-stuck.html. July 2012. (Visited on 2016-09-15).

[21]. Jeremy Siek. *Type Safety in Three Easy Lemmas*. http://siek.blogspot.ch/2013/05/type-safety-in-three-easy-lemmas.html. May 2013. (Visited on 2016-09-15).

[22]. Alexander J. Summers and Peter Müller. “Freedom Before Commitment: A Lightweight Type System for Object Initialisation”. In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. Ed. by . OOPSLA ’11. Portland, Oregon, USA: ACM,2011,pp.1013–1032.isbn:978-1-4503-0940-0.doi:10.1145/2048066.2048142.

When stack protection does not protect the stack?

Pavel Dovgalyuk <pavel.dovgaluk@ispras.ru>
Vladimir Makarov <vladimir.makarov@novsu.ru>
Novgorod State University,
41, Bolshaya Sankt Peterburgskaya, Velikiy Novgorod, 173003, Russia

Abstract. The majority of software vulnerabilities originate from buffer overflow. Techniques to eliminate buffer overflows and limit their damage include secure programming, source code audit, binary code audit, static and dynamic code generation features. Modern compilers implement compile-time and execution time protection schemes, that include variables reordering, inserting canary value, and separate stack for return addresses. Our research is targeted to finding the breaches in the compiler protection methods. We tested MSVC, gcc, and clang and found that two of these compilers have flaws that allow exploiting buffer overwrite under certain conditions.

Keywords: buffer overflow; canary protection; gcc; msvc; clang

DOI: 10.15514/ISPRAS-2016-28(5)-3

For citation: P.M. Dovgalyuk, V.A. Makarov. When stack protection does not protect the stack? Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 5, 2016, pp. 55-72. DOI: 10.15514/ISPRAS-2016-28(5)-3

1. Introduction

Even though there is a lots of progress in mitigating attacks against buffer overflows and in building static analysis tools that attempt to detect these vulnerabilities, buffer overflows remain one of the top ranking vulnerabilities year over year [1].

Writing beyond the buffer boundaries results in an undesired modification of adjacent memory locations. This data corruption can be exploited by an attacker to change the control flow of the program [2].

Software flaws that allow buffer overflow are produced by the developers. There are two kinds of countermeasures against buffer overflows [2]. First group is targeted to finding a problem before software is deployed: secure programming, source code audit, binary code audit, automatic testing (including testing with static analysis tools).

There are various static analysis techniques that address buffer overflow problem [3], [4], [5]. Static techniques extract constraints from the code and try to find incorrect memory accesses. However, these techniques produce many false positives due to the challenges faced by static analysis, such as the limited precision of alias analysis and limited loop unrolling. They may also produce false negatives due to the limited ability of constraint solvers, that are used to recover the malicious inputs [6].

Another approach is reducing the damage caused by the overflow. It includes the attempts of stopping an application after overflow is detected and reducing bug exploitability chances (address space randomization, non-executable data segments).

Wilander and Kamkar describe in their paper [7] several code generation approaches for preventing buffer overflow exploitation: reordering local variables, inserting canary values, additional stack for return addresses, checking return addresses range, checking function pointers range, and wrapping unsafe library functions.

Variables reordering and canary values protecting the return address have low runtime overhead and greatly reduce the exploitability of the overflow bugs. Nowadays these techniques are adopted by the commodity compilers Microsoft Visual C++ (MSVC) and GNU C Compiler (gcc) [8], [9].

Attacker can try to bypass the protection using other kind of attack (format string, double free, integer overflow) or bruteforcing the canary value [10], [11], [12], [13].

After bypassing protection scheme an attacker can overwrite instruction pointer and therefore hijack the control flow. This kind of threat is well known and many prior researches focused on this problem [14].

Another possible threat is overwriting the stack frame pointer (SFP) [15]. SFP is often copied to the stack pointer (SP) in the epilog of the function. When SP gets incorrect value, then function returns to the address stored in stack pointed by such fake SP. There are some other possibilities for overwriting SP that result in reading incorrect return address from the ”stack“.

Overwriting stack and frame pointers may also alter function execution before exit, because it accesses local variables through these pointers. Attacker may substitute an address to overwrite variables in another memory region or output sensitive data from the program.

In our work we address stack and frame pointer attacks, related to the buffer overflow problem. Modern compilers try to reduce attacks damage. We decided to test the compilers and stated the main research question: can stack canaries protect from overwriting stack and frame pointers?

Finally, this work makes the following contributions:

- Survey of the prolog/epilog code generation techniques in modern compilers.

- Set of the snippets for generating different stack frame manipulation samples.
- Description of MSVC stack protection bug, which allows attacking callee-saved registers. Attacker can exploit this bug to pass by the stack protection.
- Description of MSVC and clang stack protection bugs, that allow overwriting stack pointer by calling vulnerable function.

2. Protection Against Stack Buffer Overflows Through the Code Generation

We investigated code generation techniques for local buffer overflow prevention. Traditional stack frame includes arguments of the function, return address, saved value of frame pointer register, and local variables [7]. On x86 stack grows from higher to lower addresses. Therefore local buffer overflow in a specific function may overwrite its return address (Figure 1).

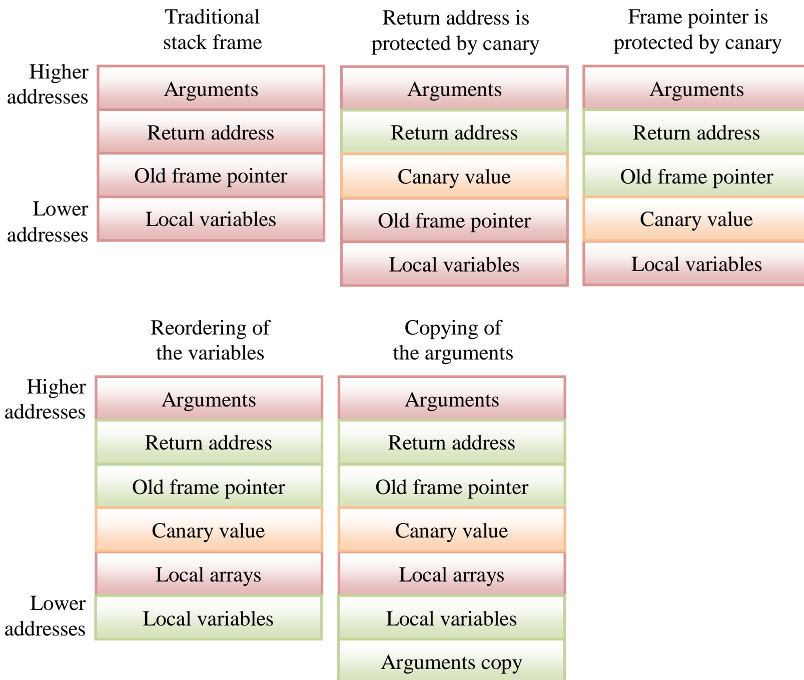


Fig. 1. Stack frame protection methods. Red frames are vulnerable. Green frames are safe.

There are multiple methods to protect from the control flow hijacking through the buffer overflow [7]. Modern compilers adopted embedding canary value into the stack frame, local variables reordering, and copying function arguments.

The structure of stack frame with these protection methods is shown in Figure 1.

2.1 Canary Value

Canary value is a local variable embedded into the stack frame to detect buffer overflows. When canary value is changed it means that some code accessed the stack frame through the wrong pointer. StackGuard concept invented by Crispin Cowan [16] is designed for using canary values to detect and stop stack-based buffer overflows targeting the return address.

Canary value is placed ”below“ the return address to detect when it is changed by user’s code. Overflow of the buffer, located in the lower addresses stack frame, targeted to overwrite the return address, will also overwrite canary value and the attack will be detected. Function checks the canary value before reading the return address from the stack. Program is terminated when canary value is incorrect, preventing buffer overflow exploitation.

Compiler developers enhanced this protection scheme. Canary value now protects not only the return address but also saved frame pointer. Prior protection schemes used predictable canary values. Attacker could ”overwrite“ them with the same values. Random canary values used in modern compilers provide efficient protection from return address and frame pointer overwriting [17], [18].

Canary values is not intended to protect other local variables when one of them overflows [17]. E.g., one local buffer may overwrite other local buffer in case of overflow. We do not address this problem in our research, because it requires more complex protection approach, which is not used in the compilers yet.

On the other hand, variables that are not addressed as buffers, may be protected by moving them to lower addresses in the stack. This protection is called variables reordering.

2.2 Reordering of the Variables

Compiler reorders variables to protect them from buffer overflows. Moving arrays to higher addresses protects other local variables from being overwritten. Buffer overflow can overwrite other variables only in the case when they are also buffers and located at higher addresses. Ordinary variables in single-buffer functions can feel safe.

Attacker can also overwrite arguments that are located at higher addresses, because belong to caller’s stack frame. This attack will be detected only at function’s exit. But values of the arguments are used before exit and attacker may affect function’s behavior before attack will be detected.

2.3 Copying Function Arguments

Compiler may protect function arguments from being overwritten by creating its copies. Arguments are protected by allocating extra space on the stack and copying their values below the local variables. The original argument values located after the return address are not used in the rest of the code [13], [17]. Therefore copying the arguments protects the function from using invalid values in the code between buffer overflow and function return.

3. Method of Analysis

To find flaws in protection mechanisms and answer the research question we decided to create multiple code snippets and compile them with different compilation options on modern compilers.

We focus on software compiled for 32-bit i386 platform. i386 family CPUs use esp register as stack pointer and in most cases use ebp as frame pointer. We tested two versions of MSVC — 2010 and 2015, two versions of gcc — 4.6.2 and 5.2.0, and clang 3.7.1.

All our snippets were on pure C, without C++-related features like exception handling. These snippets are targeted to generate different code structure of prolog and epilog parts of the functions. Prolog stores callee-saved registers and initializes the stack frame. Epilog restores the registers and exits from the function.

Prolog and epilog also include arguments copying, canary value initialization and verification embedded by the compiler.

3.1. Snippets Code Structure

We have found that the following features in program structure affect on prolog and epilog code generation.

- Function main. main function may align stack pointer on entry and restore it at exit, when it is required by the platform application binary interface.
- Variable length arrays (VLA). These arrays cannot be allocated in prolog, because its sizes are unknown. Therefore creating and overflowing them may affect the execution in unusual way.
- Aligned structures on the stack. When structures need an aligned address, compiler must align the stack pointer to allocate these structures. It changes the structure of prolog and epilog, because original unaligned stack pointer value must be restored at exit.
- `alloca` function is an alternative for variable-sized arrays. It allocates specified number of bytes on the stack. We included this function in our snippets, because MSVC doesn’t support VLA.

Every snippet has at least one function which performs operations with stack variables. Function template includes the following operations:

- Stack buffer allocation. Buffers were allocated statically or dynamically depending on function variant.
- Reading data into local buffer with gets function. `gets` was selected to demonstrate whether stack frame may be overwritten with overflow or not.
- Output buffer with `printf` function.

We also tried adding different types of parameters and return values to the functions. These options didn't add any exploitable registers or memory cells in functions prologs and epilogs. Switching between calling conventions (cdecl, fastcall, stdcall) also didn't affect potential prolog/epilog vulnerabilities.

3.1. Compilation Options

We compiled our samples with different combinations of prolog/epilog code generation options. These options are supported by all tested compilers.

- Omit frame pointer. Omitting the frame pointer is the optimization when local variables are accessed directly though the stack pointer. If frame pointer is not used, then it cannot be hijacked by the attacker. But this option does not work in some examples, making frame pointer vulnerable to overwriting.
- Stack protection. Stack protection intended to guard return address and callee-saved registers from being overwritten. However, in some test cases frame pointer or saved registers were not protected and could be overwritten by the attacker.
- Code optimizations. Code in released software binaries is usually optimized by the compiler. We analyzed non-optimized code, but do not present its flaws here, because they probably will not appear in production code.

4. Prolog/epilog in Modern Compilers

Prolog and epilog for compiled templates include different parts. Some of them belong to user's code, others are inserted by the compiler for implicit operations like saving registers. Assembly code for generated functions included the following parts:

- Saving frame pointer register (`ebp`) and copying `esp` to `ebp`. This part is omitted if frame pointer is not used in this function.
- Pushing callee-saved registers into the stack.
- Aligning `esp` and saving its initial value into some register (e.g., `esi`).
- Allocating space for the local variables.
- Initializing the canary — prolog saves canary value as a hidden local variable.

- User's code. This code can modify `esp` to allocate new variables or function parameters.
- Checking the canary.
- Restoring initial value of `esp`. `esp` may be restored by copying saved value from the register.
- Restoring values of the saved registers.
- Recovering `ebp`.
- Exitting the function. `ret` instruction loads return value stored in stack into the program counter.

The only mandatory action in this list is exitting from the function. Others depend on function structure and compiler options. The order of parts may change from compiler to compiler (e.g., the order of saving registers and initializing the canary may change).

We focus on overflows of the buffers located in the stack, but don't consider the following attacks that could be performed with this overflow:

- Attack on local variables. As said before, we don't focus on this attack, because protection mechanisms used by the compilers are not intended to stand against it.
- Direct overwrite of the return address. This attack is well known [14] and doesn't require to be analyzed here. Stack canaries used by the compilers protect return address in the first place.
- Attack on non-optimized code. This code won't appear in deployed software and therefore attackers do not interested in its vulnerabilities.

We detected several templates of prolog and epilog for MSVC, gcc, and clang compilers. Here we present an analysis of compiled samples, focusing on possible attack vectors.

4.1. Scheme of Protection

When compilers embed canary value into the stack, they insert initialization code into the prolog. This code puts some value into the canary variable. We found the following types of canaries in the generated samples.

Value from global variable. Canary value is read from some global variable. Local canary variable may be operated through `ebp` (or or through `esp`, if frame pointer is not used). Reading canary value through `ebp` protects this register from overwriting, because wrong register value will point to wrong canary value. This approach is used by gcc and clang.

Global variable `xor` frame pointer. Canary value is formed by `xor`'ing global variable with the value of frame pointer (`ebp`). When frame pointer is omitted, local variables are accessed through `esp`. In this case canary value includes `esp`, protecting it from corruption. This approach is used by MSVC.

4.2. MSVC 2010/2015

Microsoft visual C includes support of C99 subset. We tested two versions of MSVC — 2010 and 2015. We didn't compile tests with variable-length arrays, because MSVC doesn't support them.

MSVC supports stack frame protection with canary value (which is named "security cookie" in compiler documentation), local variables reordering, and function arguments copying.

Bray describes stack frame of MSVC 2003 and states that security cookie is disabled when `_alloca` function is used [8]. `_alloca` function is intended to allocate buffers in the stack frame dynamically. Therefore it can be used as a replacement of variable-length arrays.

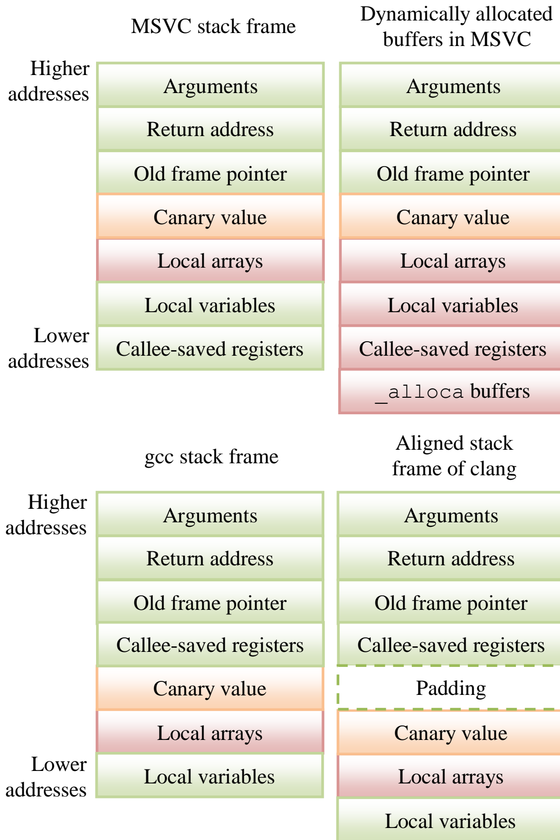


Fig. 2. Stack frame layouts used by the modern C compilers.

```
1 void func(void)
2 {
3  int sz;
4  char *buf;
5  scanf("%d", &sz);
6  buf = (char*)alloca(sz);
7  gets(buf);
8  printf(buf);
9 }
```

Figure 3. MSVC sample using _alloca function to dynamically create buffer in the stack.

```
1  push ebp
2  mov ebp, esp
3  sub esp, 8
4  mov eax, ___security_cookie
5  xor eax, ebp
6  mov [ebp-4], eax
7  push esi
8  ...
9  call __alloca_probe_16
10 ...
11 lea esp, [ebp-12]
12 ; Restore esi
13 pop esi
14 mov ecx, [ebp-4]
15 xor ecx, ebp
15 call @__security_check
17 mov esp, ebp
18 pop ebp
19 ret 0
```

Figure 4. Sample with _alloca compiled by MSVC.

```
1 __declspec(align(32))
2 struct S {
3  long long a, b, c;
4 };
5
6 void func(void)
7 {
8   struct S s;
9   char buf[8];
10  gets(buf);
11  fill(&s);
12  printf("%s %d %d %d\n",
14    buf, (int)s.a,
15    (int)s.b, (int)s.c);
16 }
```

Figure 5. Function with 32-bytes aligned local variable.

```
1  push ebx
2  mov ebx, esp
3  sub esp, 8
4  and esp, -32
5  add esp, 4
6  push ebp
7  mov ebp, [ebx+4]
8  mov [esp+4], ebp
9  mov ebp, esp
10 sub esp, 64
11 mov eax, ___security_cookie
12 xor eax, esp
13 mov [ebp-4], eax
14 ...
15 mov ecx, [ebp-4]
16 xor ecx, ebp
17 call @__security_check
18 mov esp, ebp
19 pop ebp
20 mov esp, ebx
21 pop ebx
22 ret 0
```

Figure 6. MSVC sample with aligned structures in the stack. ebp is protected by the cookie, but restoring esp from ebx is not.

```
1  struct S {
2   long long a, b, c;
3  }
4 __attribute__((aligned(32)));
5  void func(void)
6  {
7   char buf[8];
8   int n, i;
9   gets(buf);
10  sscanf(buf, "%d", &n);
11  struct S s[n];
12  // other stuff
13  ..
14 }
```

Figure 7. Variable-length array of the aligned structures.

```
1  push ebp
2  mov ebp, esp
3  push ebx
4  push edi
5  push esi
6  and esp, 0xffffffe0
7  sub esp, 0x60
8  mov esi, esp
9  mov eax, ___stack_chk_guard
10 mov [esi+0x48], eax
11 ...
12 mov eax, ___stack_chk_guard
13 cmp eax, [esi+0x48]
14 jne L
15 lea esp, [ebp-0xc]
16 pop esi
17 pop edi
18 pop ebx
19 pop ebp
20 ret
21 L: call ___stack_chk_fail
```

Figure 8. Potentially unsafe code generated by clang. esi acts as a frame pointer, esp is loaded from ebp that could be corrupted by unsafe nested functions.

We investigated newer Microsoft C compilers and found that security checks now are enabled when `_alloca` is used, but these compilers place dynamically allocated buffers directly below the callee-saved registers (Figure 2).

It means that callee-saved registers may be overwritten with buffer overflows. Consider the source code in Figure 3. It dynamically allocates memory in stack. This buffer is placed on the top of stack frame, below the callee-saved registers (Figure 2).

When `_alloca` is not used, compiler saves registers from overwriting by placing them at the top of the stack. But stack protection in MSVC lacks complete protection when `_alloca` function is used.

Figure 4 shows the compiled code. `ebp` and `esp` are protected by the security cookie, because `ebp` value is used to check the cookie, and `esp` is restored from `ebp`. `esi` is placed above the security cookie. And buffer allocated by `_alloca` is placed below saved `esi` in the memory. Therefore `esi` is not protected by being overwritten with buffer overflow.

Callee-saved registers are not read until function exit. Therefore this problem may be solved by moving calleesaved registers deeper in stack to protect them with the security cookie.

Another MSVC sample is presented in Figure 5. Function in this sample allocates aligned variable in the stack. The compiler produces the code presented in Figure 6. It aligns esp value to 32 bytes boundary which allows storing aligned structures in the stack. The compiler needs an additional register to store original unaligned esp value. It uses ebx register for that. This is callee-saved register and it is meant to remain unchanged in nested function calls.

But restoring esp from ebx at function exit is not protected by the security cookie. Therefore one can attack esp through corrupting ebx in nested functions. These functions may be located in other modules and therefore may be compiled without stack protection, allowing stack overflow exploitation. Or it may have vulnerability described in the beginning of this section.

Using ”omit frame pointer“ compilation option does not fix this example. Compiler uses ebp to store original esp value, but this ebp value is not verified in security check in the epilog. In this case attacker may hijack the stack pointer by overwriting ebp in unsafe nested function.

4.3. clang 3.7.1

Stack frame used by clang 3.7.1 compiler differs from MSVC. It puts callee-saved registers under the protection of the canary value. Therefore these values will not be popped back, because security check will stop the execution.

We checked all prepared samples and discovered that stack protection in this compiler could be unsafe in programs with aligned variable-length arrays. Consider the sample in Figure 7. Stack frame produced by clang is presented in Figure 2. It included unaligned callee-saved registers and aligned local variables.

clang generates the assembly code presented by Figure 8. It uses two copies of a stack pointer. One is stored in ebp and used to restore esp at exit. Another is located in esi and used as an aligned frame pointer.

Omitting frame pointer doesn’t work here, because variable-length array size is unknown at compile time. Imagine that this function calls some unsafe code which corrupts esi. Then checking of the canary value will not be successful. Therefore frame pointer is also protected by stack guard together with stack frame contents.

But stack guard does not protect ebp. ebp is used to restore esp in the epilog. Therefore if there exists suitable unsafe code invoked by this function (e.g., library compiled without stack protection), attacker may overwrite stack and frame pointers.

How to eliminate vulnerability of the stack pointer? Compiler may place saved esp value in the aligned stack frame above the canary value. In this case the attacker couldn’t overwrite saved esp value without being noticed.

4.4. GCC 5.2.0 and 4.6.2

gcc uses all protection methods described above: stack canaries, variable reordering, and copying function arguments.

Scheme of stack protection in gcc differs from the one used in MSVC (Figure 2). Saved registers are protected by the canary value. Therefore the user cannot attack stack pointer and saved registers when stack protection is enabled.

We investigated all examples and didn't found any cases where saved registers, stack pointer, or frame pointer could be hijacked. gcc does not allocate multiple frame pointers. Therefore it is not vulnerable to attack which corrupts one of them. And all saved registers are protected by the canary value.

5. Attack Vectors

In previous section we described flaws in the security checks of the modern compilers. Code samples described above reveal two possible attack vectors: overwriting stack pointer and overwriting callee-saved registers.

Overwriting `esp` may be used to control program execution through pointing esp at the memory where known address is located. Function will try to return to that address and will jump to the desired code. It could be something like `AuthentificationSuccess` to make "useful" work, exit for DoS'ing without an alert, or shellcode supplied by the attacker.

Another kind of attack is hijacking the stack or frame pointer to control local variables. Function works with local variables through the frame pointer. If its value is supplied by the attacker, function may read or modify unattended data. This may lead to leakage of sensitive data or to taking alternative branches in the function. Finally, the function will jump to some address with ret instruction. Richarte in his paper [12] presents an example of such an attack.

Modern operating systems have address space layout randomization (ASLR) enabled, which hampers these kind of attacks, because an attacker cannot guest the target address. However, this protection may be bypassed in some cases. E.g., by using the addresses from a module without ASLR [11].

Third kind of attack is overwriting caller's local variables. Stack pointer in this attack is not affected — application will not crash due to jumping to some incorrect address. Attacker has to corrupt callee-saved registers, exploiting the flaws of stack protection mechanism. Caller will not detect that these registers changed, because calling convention declares that callee preserves register values.

6. Related Work

Other attacks and compiler enhancements are described by other researchers. Wilander et al. states, that attacks on return address and old base pointer can be successfully prevented by the runtime stack protection methods [7], [19]. These

protection methods are described in section 2. In our research we verified these results and found just few cases when protection can be bypassed.

Paper ”Four different tricks to bypass StackShield and StackGuard protection“ presents description of ways to bypass the stack protection techniques [12]. It focuses on stack frame hijacking, when it is not protected by the canary. State-of-the-art compilers protect frame pointer as well as return address, therefore it cannot be overwritten as easy as in 2002.

Function argument attack described in that paper can also be fought back by the modern compilers. They copy arguments to the stack area above the buffer. Buffer overflow cannot touch arguments when they are located at the lower addresses.

Our research show that `alloca` makes stack protection task harder for the compiler. `alloca` is already known as a source of stack overflow attacks [20]. Therefore every use of this function has to be double-checked.

Canary value protection bypassing methods using exception handlers, virtual tables and couple of other approaches are presented in [11]. It means that stack protection already has some drawbacks in addition to ones that we discovered.

We haven’t found any flaws in gcc stack protection method. However, we didn’t take in account several overflow possibilities, including overwriting of one local buffer with overflow of the another. Paper [17] describes different approaches of enhancing stack protection in gcc. The first improvement is verification of the canary not only when function returns, but also when the function issues a call to another function. This check prevents passing invalid arguments to the nested function. The second improvement is assigning an individual canary for each buffer. With this patch overwriting of one local buffer with another will be detected by the security check. The third improvement makes canary location and failure probabilistic. It makes application harder to attack and reduces amount of information supplied to the attacker in case of the failure. Authors also present patches that implement described protection enhancements. These patches may incur much greater runtime overhead than current protection methods. Therefore they are not used by default yet.

7. Conclusion

In this paper we presented the analysis of the buffer overflow protection methods used in modern compilers. Our tests showed that modern compiler may miss some cases where stack pointer is not protected. Attacker may get control over the application which stack is “protected” with canary value.

We found that methods used in clang and MSVC have several flaws. Both of these compilers does not protect restoring of `esp` from one of the registers, when using aligned data structures in the stack. MSVC also doesn’t protect all saved registers in programs with `_alloca` function.

gcc was the third compiler to examine. We haven’t found any protection issues in samples compiled with gcc.

Tests that were created in this research are released in repository https://github.com/Dovgalyuk/SecurityFlaws. Everyone can generate assembly files for its own compiler and examine the security features of code generation.

Acknowledgments

The work was partially supported by RFBR, research project No. 14-07-00411 a.

References

[1]. Y. Younan, "25 years of vulnerabilities: 1988–2012," Tech. Rep., 2012. [Online]. Available: https://courses.cs.washington.edu/courses/cse484/14au/reading/25-years-vulnerabilities.pdf

[2]. M. Vallentin, "On the evolution of buffer overflows," 2007.

[3]. D. Baca, K. Petersen, B. Carlsson, and L. Lundberg, "Static code analysis to detect software security vulnerabilities - does experience matter?" in Availability, Reliability and Security, 2009. ARES '09. International Conference on, March 2009, pp. 804–810.

[4]. A. Austin and L. Williams, "One technique is not enough: A comparison of vulnerability discovery techniques," in 2011 International Symposium on Empirical Software Engineering and Measurement, Sept 2011, pp. 97–106.

[5]. N. Rutar, C. B. Almazan, and J. S. Foster, "A comparison of bug finding tools for java," in Proceedings of the 15th International Symposium on Software Reliability Engineering, ser. ISSRE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 245–256. [Online]. Available: http://dx.doi.org/10.1109/ISSRE.2004.1

[6]. H. Sun, X. Zhang, C. Su, and Q. Zeng, "Efficient dynamic tracking technique for detecting integer-overflow-to-buffer-overflow vulnerability," in Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ser. ASIA CCS '15. New York, NY, USA: ACM, 2015, pp. 483–494. [Online]. Available: http://doi.acm.org/10.1145/2714576.2714605

[7]. J. Wilander and M. Kamkar, "A comparison of publicly available tools for dynamic buffer overflow prevention," in IN NDSS, 2003.

[8]. B. Bray, "Visual studio .net 2003: Compiler security checks in depth," February 2002. [Online]. Available: https://msdn.microsoft.com/enus/library/Aa290051

[9]. "Stack smashing protector." [Online]. Available: http://wiki.osdev.org/Stack_Smashing_Protector

[10]. Bulba and Kil3r, "Bypassig stackguard and stackshield," Phrack Magazine, vol. 56, May 2000. [Online]. Available: http://phrack.org/issues/56/5.html

[11]. C. Team, "Exploit writing tutorial part 6: Bypassing stack cookies, safeseh, sehop, hw dep and aslr," 2009. [Online]. Available: https://www.corelan.be/index.php/2009/09/21/exploit-writingtutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/

[12]. G. Richarte, "Four different tricks to bypass stackshield and stackguard protection," World Wide Web, vol. 1, 2002.

[13]. A. Sotirov and M. Dowd, "Bypassing browser memory protections," in In Proceedings of BlackHat, 2008. [Online]. Available: http://www.blackhat.com/presentations/bh-usa-08/Sotirov_Dowd/bh08-sotirov-dowd.pdf

[14]. A. One, "Smashing the stack for fun and profit," Phrack Magazine, vol. 49, November 1996. [Online]. Available: http://phrack.org/issues/49/14.html

[15]. klog, “The frame pointer overwrite,” Phrack Magazine, vol. 55, September 1999. [Online]. Available: http://phrack.org/issues/55/8.html

[16]. C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, “Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks,” in In Proceedings of the 7th USENIX Security Symposium, 1998, pp. 63–78.

[17]. A. Seredinschi, Drago¸s-Adrian; Sterca, “Enhancing the stack smashing protection in the gcc,” Studia Universitatis Babe¸s-Bolyai, Informatica, vol. LV, Number 1, 2010.

[18]. Y. WU, “Enhancing security check in visual studio c/c++ compiler,” in Software Engineering, 2009. WCSE ’09. WRI World Congress on, vol. 4, May 2009, pp. 109–113.

[19]. P. Silberman and R. Johnson, “A comparison of buffer overflow prevention implementations and weaknesses.” [Online]. Available: https://www.blackhat.com/presentations/bh-usa-04/bh-us-04-silberman/bh-us-04-silberman-paper.pdf

[20]. C. Evans, “glibc alloca() memory corruption,” 2011. [Online]. Available: https://packetstormsecurity.com/files/98720/

Когда защита стека в компиляторах не срабатывает?

Павел Довгалюк <pavel.dovgaluk@ispras.ru>
Владимир Макаров <vladimir.makarov@novsu.ru>
Новгородский государственный университет,
173003, Россия, Великий Новгород, ул. Большая Санкт-Петербургская, д. 41

Аннотация. Основная часть уязвимостей в программах вызвана переполнением буфера. Чтобы предотвратить переполнение буфера и уменьшить ущерб от него используется безопасное программирование, аудит исходного кода, аудит бинарного кода, статические и динамические особенности кодогенерации. В современных компиляторах реализованы механизмы защиты, работающие на этапе компиляции и на этапе выполнения скомпилированной программы: переупорядочивание переменных, копирование аргументов и встраивание стековой канарейки. В статье описывается исследование, посвященное поиску недостатков в этих механизмах. Мы протестировали компиляторы MSVC, gcc и clang и обнаружили, что два из них содержат ошибки, позволяющие эксплуатировать переполнение буфера при определенных условиях.

Ключевые слова: переполнение буфера; стековая канарейка; gcc; msvc; clang

DOI: 10.15514/ISPRAS-2016-28(5)-3

Для цитирования: П.М. Довгалюк, В.А. Макаров. Когда защита стека не срабатывает? Труды ИСП РАН, том 28, вып. 5, 2016, стр. 55-72 (на английском). DOI: 10.15514/ISPRAS-2016-28(5)-3

Литература

[1]. Y. Younan, “25 years of vulnerabilities: 1988–2012,” Tech. Rep., 2012. [Online]. Available: https://courses.cs.washington.edu/courses/cse484/14au/reading/25-years-vulnerabilities.pdf
[2]. M. Vallentin, “On the evolution of buffer overflows,” 2007.
[3]. D. Baca, K. Petersen, B. Carlsson, and L. Lundberg, “Static code analysis to detect software security vulnerabilities - does experience matter?” in Availability, Reliability and Security, 2009. ARES ’09. International Conference on, March 2009, pp. 804–810.
[4]. A. Austin and L. Williams, “One technique is not enough: A comparison of vulnerability discovery techniques,” in 2011 International Symposium on Empirical Software Engineering and Measurement, Sept 2011, pp. 97–106.
[5]. N. Rutar, C. B. Almazan, and J. S. Foster, “A comparison of bug finding tools for java,” in Proceedings of the 15th International Symposium on Software Reliability

Engineering, ser. ISSRE ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 245–256. [Online]. Available: http://dx.doi.org/10.1109/ISSRE.2004.1

[6]. H. Sun, X. Zhang, C. Su, and Q. Zeng, “Efficient dynamic tracking technique for detecting integer-overflow-to-buffer-overflow vulnerability,” in Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ser. ASIA CCS ’15. New York, NY, USA: ACM, 2015, pp. 483–494. [Online]. Available: http://doi.acm.org/10.1145/2714576.2714605

[7]. J. Wilander and M. Kamkar, “A comparison of publicly available tools for dynamic buffer overflow prevention,” in IN NDSS, 2003.

[8]. B. Bray, “Visual studio .net 2003: Compiler security checks in depth,” February 2002. [Online]. Available: https://msdn.microsoft.com/enus/library/Aa290051

[9]. “Stack smashing protector.” [Online]. Available: http://wiki.osdev.org/Stack_Smashing_Protector

[10]. Bulba and Kil3r, “Bypassig stackguard and stackshield,” Phrack Magazine, vol. 56, May 2000. [Online]. Available: http://phrack.org/issues/56/5.html

[11]. C. Team, “Exploit writing tutorial part 6: Bypassing stack cookies, safeseh, sehop, hw dep and aslr,” 2009. [Online]. Available: https://www.corelan.be/index.php/2009/09/21/exploit-writingtutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/

[12]. G. Richarte, “Four different tricks to bypass stackshield and stackguard protection,” World Wide Web, vol. 1, 2002.

[13]. A. Sotirov and M. Dowd, “Bypassing browser memory protections,” in In Proceedings of BlackHat, 2008. [Online]. Available: http://www.blackhat.com/presentations/bh-usa-08/Sotirov_Dowd/bh08-sotirov-dowd.pdf

[14]. A. One, “Smashing the stack for fun and profit,” Phrack Magazine, vol. 49, November 1996. [Online]. Available: http://phrack.org/issues/49/14.html

[15]. klog, “The frame pointer overwrite,” Phrack Magazine, vol. 55, September 1999. [Online]. Available: http://phrack.org/issues/55/8.html

[16]. C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, “Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks,” in In Proceedings of the 7th USENIX Security Symposium, 1998, pp. 63–78.

[17]. A. Seredinschi, Drago¸s-Adrian; Sterca, “Enhancing the stack smashing protection in the gcc,” Studia Universitatis Babe¸s-Bolyai, Informatica, vol. LV, Number 1, 2010.

[18]. Y. WU, “Enhancing security check in visual studio c/c++ compiler,” in Software Engineering, 2009. WCSE ’09. WRI World Congress on, vol. 4, May 2009, pp. 109–113.

[19]. P. Silberman and R. Johnson, “A comparison of buffer overflow prevention implementations and weaknesses.” [Online]. Available: https://www.blackhat.com/presentations/bh-usa-04/bh-us-04-silberman/bh-us-04-silberman-paper.pdf

[20]. C. Evans, “glibc alloca() memory corruption,” 2011. [Online]. Available: https://packetstormsecurity.com/files/98720/

Оценка критичности программных дефектов в условиях работы современных защитных механизмов*

[1]А.Н. Федотов <fedotoff@ispras.ru>
[1,2]В.А. Падарян <vartan@ispras.ru>
[1]В.В. Каушан <korpse@ispras.ru>
[1]Ш.Ф. Курмангалеев <kursh@ispras.ru>
[1]А.В. Вишняков <vishnya@ispras.ru>
[1]А.Р. Нурмухаметов <oleshka@ispras.ru>
[1]Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25
[2]Московский государственный университет имени М.В. Ломоносова,
119991 ГСП-1, Москва, Ленинские горы

Аннотация. В данной работе предложен уточненный метод автоматизированной оценки степени опасности найденных программных дефектов. На этапе тестирования промышленного программного обеспечения выявляется значительное число дефектов, приводящих к аварийному завершению. Из-за ограниченности ресурсов исправление дефектов растягивается во времени и требует расстановки приоритетов. Основным критерием для назначения высокого приоритета становится возможность использования ошибки в злонамеренных целях. На практике эта задача решается путем автоматизированного построения входных данных, подтверждающего наличие опасной уязвимости. Но в известных публикациях по данной теме не учитывают современные защитные механизмы, препятствующие действиям атакующего, что снижает качество вырабатываемых оценок. В данной статье рассматриваются современные защитные механизмы и дается оценка их распространенности и эффективности. Метод применим к бинарным файлам и не требует какой-либо отладочной информации. В основе метода лежит символьная интерпретация трасс выполнения, полученных при помощи полносистемного эмулятора. Даже в условиях одновременного функционирования DEP, ASLR и защиты стека («канарейка») метод способен демонстрировать возможность использования ошибок, сочетающих условия «write-what-where» и переполнение буфера на стеке. Возможности реализованного метода были продемонстрированы на модельных примерах и реальных программах.

Ключевые слова: уязвимость; критический дефект; бинарный код; символьное выполнение.

---

* Работа поддержана грантом РФФИ № 16-29-09632

DOI: 10.15514/ISPRAS-2016-28(5)-4

Для цитирования: Федотов А.Н., Падарян В.А., Каушан В.В., Курмангалеев Ш.Ф., Вишняков А.В., Нурмухаметов А.Р. Оценка критичности программных дефектов в рамках работы современных защитных механизмов. Труды ИСП РАН, том 28, вып. 5, стр. 73-92, 2016 г. DOI: 10.15514/ISPRAS-2016-28(5)-4

1. Введение

Обеспечение безопасности программного обеспечения является на сегодняшний день одной из важных задач. Программные продукты применяются в повседневно окружающих нас вещах: настольных компьютерах, смартфонах, автомобилях и технологиях «интернета вещей». Кроме того программное обеспечение используется на объектах критической инфраструктуры, сбои в работе которых могут привести к серьезным последствиям, а использование ошибок в злонамеренных целях может причинить колоссальный ущерб. В безопасности программных продуктов заинтересованы не только конечные пользователи, но и разработчики. Различные научные институты и крупные корпорации уделяют особое внимание разработке методов и технических средств для поиска и оценки возможности эксплуатации уязвимостей.

Одной из первых работ, в которой описывался процесс эксплуатации уязвимости переполнения буфера на стеке была статья [1], опубликованная в 1996 году. На тот момент промышленно выпускаемые программы не обладали защитными механизмами и использование такого рода уязвимостей проходило не вызывая затруднений. С появлением защитных механизмов уровня операционной системы, таких как рандомизация адресного пространства (ASLR) и защита от исполнения данных (DEP), методы использования уязвимостей, направленные на выполнение произвольного кода, заметно усложнились, а в некоторых случаях стали просто неприменимыми. В современных дистрибутивах операционных систем ASLR и DEP, как правило, включены по умолчанию. Вместе с тем, существуют ситуации, когда использование уязвимостей возможно, несмотря на работу этих защитных механизмов [2,3]. Метод защиты, предполагающий размещение в автоматической памяти служебных переменных, «канареек», был призван защищать программу от ошибок переполнения буфера на стеке, но так же, как и в предыдущем случае, существуют возможности обхода данного защитного механизма [4].

В настоящее время активно развивается технология автоматизированной генерации эксплойтов [6-10]. В основе этой технологии лежит динамическое символьное выполнение. Символьное выполнение представляет собой процесс выполнения программы, где вместо конкретных значений переменных используются символьные значения. В качестве символьных значений обычно

выступают входные данные программы, которые могут быть получены из различных источников (сеть, файлы, аргументы командной строки, переменные окружения). Процесс преобразования входных данных в результате работы программы описывается в виде формул над символьными переменными. Условные ветвления, результат выполнения которых зависит от входных данных, порождают уравнения, отвечающие прохождению по определенной ветке в программе. Совокупность всех таких уравнений называется предикатом пути. Решением системы уравнений является конкретный набор входных данных, обеспечивающий прохождение потока управления по пути, в рамках которого была построена данная система уравнений. Для решения систем уравнений используются SMT-решатели [20].

Под эксплуатацией уязвимости понимается процесс перехода программы в состояние, в котором данная уязвимость проявляется. Это состояние описывается при помощи дополнительных уравнений, которые в совокупности с предикатом пути обеспечивают процесс эксплуатации уязвимости. Решением такой системы уравнений будет являться эксплойт – набор входных данных, активирующий уязвимость. Формальное описание срабатывания уязвимостей является сложной задачей, но для некоторых классов уязвимостей, например, переполнение буфера на стеке эта задача уже решена и описана в работах [6-10]. Представленные в этих работах инструменты позволяют эксплуатировать уязвимости при отключенных современных защитных механизмах, что накладывает ограничения на применимость данных инструментов в настоящее время. Данная статья предлагает подход к преодолению этих ограничений.

Статья организована следующим образом. Во втором разделе приводится краткое описание современных защитных механизмов, а также способов их обхода. В третьем разделе представлены методы генерации эксплойтов с учетом работы современных защитных механизмов. Реализация и практическое применение рассматриваются в четвертом разделе. В заключении, обсуждаются полученные результаты и дальнейшие направления исследований.

2. Защитные механизмы и способы их обхода

Среди современных механизмов защиты от эксплуатации уязвимостей можно выделить две категории, повсеместно применяющиеся и во многом определяющие состояние защищенности настольных компьютеров и серверов: защитные механизмы операционной системы и защитные механизмы, предоставляемые компилятором. К первой категории относятся рандомизация адресного пространства (ASLR) и предотвращение выполнения данных (DEP). Во вторую категорию входят упомянутый выше механизм размещения «канареек» и технология Fortify source, совмещающая легковесные проверки времени компиляции и автоматическую замену потенциально уязвимых

функций на защищенные аналоги. На стыке двух категорий оказываются методы защиты, работающие во время загрузки программы и динамической компоновки.

Рандомизация адресного пространства (ASLR). Рандомизация адресного пространства предполагает, что программа будет загружаться на различные адреса. В Windows рандомизация поддерживается, начиная с Windows Vista. В ОС Linux данный механизм защиты доступен достаточно давно, начиная с ядра версии 2.6.25 (17 апреля 2008 г.), но полноценная реализация требует решения ряда вопросов.

Рандомизировать необходимо размещение стека, кучи, динамически загружаемых библиотек, адреса загрузки секций ELF-файлов (в том числе секции .text). Рандомизация загрузки динамических библиотек, в свою очередь, требует обязательной рандомизации в функциях отображения файлов в память (mmap). Размещение на произвольных адресах секции .text требует от разработчиков компилировать все исполняемые файлы приложений в позиционно-независимый код, для чего необходимы дополнительные усилия. Отказ от компиляции в позиционно-независимом коде приводит к тому, что образ программы остается нерандомизированным.

Следует отметить, что большинство реализаций ASLR вырабатывают случайную карту памяти для программы один раз на все время работы системы до следующей загрузки. Это означает, что перезапускаемые сервисы, падение которых не приводит к падению всей системы, будут достаточно долго работать на одной и той же карте памяти. Утечка информации об адресах работающей программы позволит настроить эксплойт таким образом, что он сможет преодолеть ASLR.

Таким образом, имеются два основных подхода к противодействию ASLR: (1) передача управления на нерандомизированные области памяти; (2) выяснение и последующее использование знаний об адресах переменных, кучи, стека, а также базовых адресов исполняемых модулей. В первом случае используется передача управления на инструкции «трамплины» вида *call/jump $reg*, выполнение которых в свою очередь передает управление на код полезной нагрузки. Стоит отметить, что для использования этой технологии необходимо наличие нерандомизированной и исполняемой области памяти, а в ней существование «трамплина», и кроме того значение регистра должно указывать в область памяти, находящуюся под контролем атакующего. Второй способ, утечка чувствительных данных, реализуется посредством эксплуатации все тех же уязвимостей форматной строки или переполнения буфера.

Предотвращение выполнения данных DEP. Предотвращение выполнения данных – механизм защиты, встроенный в различные операционные системы Windows, Linux и т.д., который запрещает программе выполнять код из области памяти, помеченной как «данные». Таким образом, стек и куча

становятся недоступными для выполнения. Для обхода этой защиты используется атака возврата в библиотеку, в результате которой происходит подмена адреса возврата из функции адресом библиотечной функции, например, функции system из библиотеки libc.

ASLR и DEP. Одновременная работа этих защитных механизмов значительно усложняет процесс эксплуатации уязвимостей. В условиях работающей технологии DEP бессмысленно использовать «трамплины», передающие управление на стек, так как эта часть памяти выполняться не может. Также нельзя использовать атаку возврата в библиотеку, если адрес загрузки библиотеки рандомизируется. Для обхода этих защит можно использовать технологию, которая называется возвратно-ориентированное программирование (ROP) [8]. Код полезной нагрузки формируется из исполняемых участков кода, заканчивающихся инструкцией передачи управления. Такие участки кода называются гаджетами. Гаджеты выстраиваются в цепочки: первый гаджет передает управление второму, второй – третьему и т.д. Поиск гаджетов происходит в нерандомизированных областях памяти. В Linux исполняемые файлы, как правило, не собираются в виде позиционно-независимого кода, поэтому адрес загрузки образа программы не изменяется от запуска к запуску. В таком случае, исполняемый модуль подходит для поиска гаджетов.

«Канарейка». «Канарейка» – это специальное значение, которое размещено на стеке и разделяет собой пространство автоматических локальных переменных и служебные данные – адрес возврата и сохраненные регистры. Размещение «канарейки» выполняется в прологе функции, в эпилоге ее значение сравнивается с эталоном. Их различие интерпретируется как срабатывание ошибки переполнения буфера с угрозой порчи адреса возврата и других служебных данных. Программа в таком случае аварийно завершается, не доходя до более серьезных нарушений безопасности.

Значение «канарейки» либо заранее определено (состоит из терминальных символов), либо генерируется случайным образом. Кроме добавления «канарейки» происходит перестановка локальных переменных. Все указатели располагаются перед массивами, что препятствует их перезаписи. Стоит отметить, что перестановок полей в структурах не происходит.

Применение «канарейки» сразу же показало эффективность данного метода защиты. Схожим образом защищаются буферы динамической памяти, соответствующая технология была реализована в glibc 2.3.4 [13]. Помимо того, защитные механизмы контролируют порчу служебных данных кучи, и производят дополнительные проверки, например, на двойное освобождение. Защита кучи развивалась вплоть до версии glibc 2.10, учитывая более изощренные способы эксплуатации [14].

Для обхода «канарейки» может быть использована уязвимость CWE-123 [21], когда имеется возможность записать одно произвольное значение по

произвольному адресу. Например, адрес указывает на ячейку памяти, содержащую адрес возврата из функции. Записываемым значением выступает адрес, по которому расположен код полезной нагрузки. Уязвимость реализуется путем переполнения буфера в структуре, с последующей перезаписью поля-указателя.

ASLR, DEP и «канарейка». Одновременная работа трех защит делает эксплуатацию уязвимостей еще менее осуществимой, но полностью такую возможность не исключает. При выполнении ряда условий угроза эксплуатации остается актуальной.

- Исполняемый модуль программы не рандомизируется.
- Наличие гаджета-трамплина, который сдвигает указатель стека на определенное значение.
- ELF-файл использует «ленивое» связывание.
- В программе должна присутствовать уязвимость CWE-123. В качестве адреса записи используется ячейка .got секции, в которой располагается адрес вызываемой в дальнейшем библиотечной функции. Записываемое значение – адрес гаджета-трамплина, который передвинет указатель стека на оставшуюся часть ROP-цепочки.
- Наличие данных, зависящих от входных данных на стеке в момент передачи управления на гаджет-трамплин.

Fortify Source. Компилятор gcc, начиная с версии 4.0, поддерживает макрос FORTIFY_SOURCE, активирующий легковесные проверки на переполнение буфера в библиотечных функциях копирования: memcpy, strcpy, sprintf, gets и др. Некоторые проверки осуществляются во время компиляции, выдавая результат в виде предупреждений, остальные проверки происходят во время выполнения и в случае срабатывания аварийно завершают программу. Реализуются проверки времени выполнения путем автоматической замены вызываемых функций на их аналоги, имеющие дополнительный параметр – размер буфера-приемника, в тех случаях, когда этот размер известен.

Помимо того, Fortify Source отслеживает в компилируемом коде наличие проверок возвращаемых значений определенных функций (system, write, open, gets и др.). Функции, работающие с форматной строкой, заменяются аналогами, которые проверяют использование формата %n, отслеживают использование не литеральных форматов и др. Наличие таких проверок нацелено на предотвращение целенаправленной перезаписи адреса возврата, не вызывающей порчи «канарейки».

Компиляторные преобразования для повышения сложности эксплуатации уязвимостей. В программном обеспечении содержатся уязвимости, для каждой найденной уязвимости требуется выпустить обновление безопасности и распространить на все изделия. Если добиться того, чтобы эксплойт разработанный для одного из изделий оказался не

работоспособен на других, либо приводил не к исполнению произвольного кода, а к отказу в обслуживании, то это позволит повысить безопасность. Для достижения этой цели на базе обфусцирующего компилятора, разрабатываемого в ИСП РАН, были реализованы преобразования [15]: перестановки функций, добавления и перемешивания локальных переменных. Совместное применение данных преобразований позволяет создать популяцию диверсифицированных бинарных файлов. В качестве базы для реализации преобразований использовались компиляторы GCC и CLANG/LLVM. С помощью компилятора GCC производилась полная сборка системы CentOS, включая ядро Linux. Влияние на производительность оценивалось на приложении SQLite, пиковое замедление составило 15% при увеличении размера кода на 5%.

Добавление и перемешивание локальных переменных преследовало цель затруднить активацию известных уязвимостей, поскольку изменяется размер и расположение перезаписываемых локальных переменных (буферов). Перестановка функций местами направленно на дополнение системного механизма ASLR и повышение его разрешения до уровня функций в исполняемом модуле для затруднения построения цепочки гаджетов, работающих на других сборках ПО. Для обеспечения максимального уровня защиты рекомендуется использовать предлагаемый подход совместно с ASLR и DEP.

Рандомизация адресного пространства с гранулярностью до функций средствами навесной защиты (технология Selfrando). Дальнейшим развитием идеи рандомизации адресного пространства является уменьшение размера участков памяти, для которых производится рандомизация. Одна из реализаций такого подхода внедрена в Tor браузер с июня 2016 года [16]. Суть идеи состоит в размещении функций в отдельных секциях для получения информации об их границах с использованием специальной опции компилятора. В дальнейшем информация о границах функций, настраиваемых адресах и специальный код начального загрузчика во время компоновки добавляются к образу исполняемого файла, точка входа которого изменяется на адрес начального загрузчика. Далее используется идея, схожая с используемой в упаковщиках исполняемых файлов. После загрузки программы в память система передает управление на код начального загрузчика, который, используя информацию о функциях, производит их размещение в памяти в случайном порядке, настраивает адреса и передает управление на оригинальную точку входа. Для оценки эффективности метода авторы провели эксперимент по оценке энтропии, обеспечиваемой ASLR и их подходом. Для системы Debian 8.4 и компиляторов GCC 6.1.0 и Clang 3.5.0 на 32-х битной версии ОС ASLR обеспечивает изменение 9 бит адреса, на 64-х битной версии – 29 бит. Энтропия, обеспечиваемая Selfrando, зависит от размера модуля программы и количества функций (N): для 10KB кода энтропия составляет 13 x N бит, для 6.6 MB – 22 x N бит, для 92 MB – 26 x N

бит. Влияние на производительность оценивалось с помощью SPEC 2006, геометрическое среднее значение замедления по всем тестам составило 0.71% для GCC. Тестирование производилось в режиме “Identity transformation”, при котором отрабатывает код загрузчика, но все функции остаются в оригинальном порядке.

Защита времени загрузки. Некоторые приемы эксплуатации уязвимостей используют перезапись элементов Global Offset Table (GOT). Для защиты от таких приемов глобальную таблицу смещений размещают в памяти, доступной только на чтение. Чтобы полностью защитить ее от перезаписи, необходимо отказаться от ленивого связывания, использующегося в динамической компоновке. Для этого ELF файл подготавливается, как требующий незамедлительное связывание (BIND_NOW).

2.1 Анализ современных дистрибутивов ОС Linux на предмет защищенности исполняемых файлов

Анализ на предмет защищенности проводился для исполняемых файлов из директории */usr/bin/* в распространенных дистрибутивах ОС Linux. Для анализа использовалась утилита hardening-check из пакета hardening-includes. Данная утилита позволяет проверить, собран ли исполняемый файл в позиционно-независимом коде (ПНК), используются ли безопасные функции Fortify Source и «канарейка», защищена ли секция .got от записи, происходит ли связывание непосредственно в момент загрузки. В табл. 1 приведены результаты анализа некоторых современных дистрибутивов Linux, позволяющие сделать несколько выводов.

Табл. 1. Результаты анализа содержимого /usr/bin некоторых дистрибутивов Linux

Table 1. Analysis results for binaries from /usr/bin for some Linux distributions

Дистрибутив	Дата выпуска	Количество исп. файлов	Без ПНК	Без «канарейки»	Без Fortify Source	Ленивое связывание
Debian 6.0.10 32 разряда	19.07.14	4705	4613 / 98%	4617 / 98%	3899 / 83%	4639 / 99%
Debian 8.3.0 32 разряда	23.01.16	387	311 / 80%	81 / 23%	70 / 20%	311 / 80%
Arch 32 разряда	25.05.16	2846	2671 / 94%	383 / 13%	493 / 17%	2717 / 95%
Ubuntu 14.10 32 разряда	23.10.14	1162	1016 / 87%	242 / 21%	96 / 8%	1036 / 89%
Ubuntu 14.04.1 64 разряда	24.07.14	851	712 / 84%	67 / 8%	48 / 6%	709 / 83%
Ubuntu 16.04.1 64 разряда	21.07.16	1053	891 / 85%	211 / 20%	81 / 8%	881 / 84%

Подавляющее большинство (порядка 85%) исполняемых файлов собирается не в виде позиционно-независимого кода, что позволяет их использовать для поиска гаджетов. При этом следует признать, что не изучалось, какие именно программы остались позиционно зависимыми и насколько критична их компрометация с точки зрения безопасности всей системы. С другой стороны доля программ с позиционно зависимым кодом не меняется в течение последних двух лет, их одинаково много как в 32-х разрядных, так и в 64-х разрядных системах.

Во многих программах используется ленивое связывание, что оставляет возможность перезаписи .got-слотов с целью передачи управления на код полезной нагрузки.

Анализ дистрибутивов Linux показал, что в большинстве современных дистрибутивах присутствуют такие защитные механизмы, как: ASLR, DEP, «канарейка» и FORTIFY_SOURCE. Защитный компиляторный механизм FORTIFY_SOURCE фактически исправляет некоторые ошибки, совершаемые разработчиком, тем самым устраняя потенциальные возможности эксплуатации уязвимостей, основанных на этих дефектах. Таким образом, предлагаемые методы эксплуатации сосредоточены на попытках преодоления таких защитных механизмов, как: ASLR, DEP и «канарейка».

3. Методы использования уязвимостей

Предложенные методы являются развитием подхода к генерации эксплойтов для уязвимости переполнения буфера на стеке, описанного в предыдущей статье [10]. Методы основываются на анализе трасс выполнения, полученных при помощи полносистемного эмулятора. Процесс генерации эксплойта состоит из двух этапов: построения предиката пути и описания предиката безопасности.

3.1 Построение предиката пути

Предикат пути представляет собой набор символьных уравнений и неравенств, описывающий процесс преобразования входных данных программы с момента получения и до аварийного завершения. Для поиска аварийных завершений используется подход на основе анализа прерываний процессора, более подробно рассмотренный в работе [10]. Указание момента получения входных данных, а именно: шага трассы, на котором буфер с входными данными оказывается заполненным, может задаваться аналитиком или определяться автоматически. Для этого используется анализ функций, которые получают данные из различных источников: сеть, файлы, аргументы командной строки. Как правило, такие функции располагаются в известных библиотеках, например, функция *recv* из библиотеки *libc*. Аналитик заранее может подготовить описания известных ему функций получения входных данных и их параметров в различных библиотеках. Эта информация будет

использоваться для автоматического поиска точек получения входных данных.

Построение предиката пути над символьными переменными основывается на отслеживании помеченных данных. Отобранные в результате отслеживания помеченных данных машинные инструкции переводятся в архитектурно-независимое промежуточное представление [11], уже по которому строятся символьные уравнения и неравенства. Как известно, анализу помеченных данных свойственны такие проблемы, как недостаточная помеченность и избыточная помеченность [12].

Недостаточная помеченность возникает, как правило, из-за того, что во время анализа не учитываются некоторые зависимости. В ходе работы программы символьные данные могут оказаться в адресном коде, определяя значение адреса памяти. Дальнейшая интерпретация кода либо предполагает обращение к любой ячейке адресуемой памяти, либо требует ограничения числа возможных адресов, вплоть до конкретизации значения адреса. Такая проблема известна в публикациях, как проблема «символьных адресов» [6].

В рамках предлагаемого подхода символьные пометки не распространяются через адреса, что может приводить к недостаточной помеченности. В свою очередь, недостаточная помеченность может привести к тому, что набор входных данных, полученный в результате решения предиката пути, не проведет программу по тому же самому пути выполнения.

В некоторых случаях последствий недостаточной помеченности можно избежать, добавляя дополнительные ограничения на символьные переменные. Обладая некоторыми сведениями об устройстве программы, аналитик может добавить ограничения на входные данные, параметры функций, а также на произвольные ячейки памяти и регистры. В качестве примера, можно привести ситуацию, когда входные данные считываются при помощи функции *scanf*. Аналитик может добавить такие ограничения, что входные данные не содержат терминальных символов и пробелов.

Избыточная помеченность может привести к росту количества отобранных инструкций, не все из которых оказывают существенное влияние на ход анализа. В основном, такие инструкции содержатся в коде библиотек. Прекращение отслеживания помеченных данных при выполнении кода библиотек может привести к потере нужных зависимостей, например, если копирование помеченных данных происходит с использованием функций копирования. Поэтому предлагается следующий подход. Аналитик указывает список неинтересных для анализа функций. Помимо этого, для каждой функции указывается набор параметров, с которых необходимо снять пометки. Таким образом, в предикат пути не попадут те инструкции внутри функции, на результат выполнения которых могли повлиять выбранные параметры. Примером такой функции может послужить функция *malloc* из

библиотеки *libc*. Использование данного подхода позволяет значительно снизить количество отобранных инструкций.

3.2 Построение предиката безопасности

Набор символьных уравнений и неравенств, который описывает факт проявления уязвимости, называется предикатом безопасности. Как правило, построение предиката безопасности происходит в момент проявления дефекта, т.е. сразу после построения предиката пути. В совокупности с предикатом пути, предикат безопасности составляет полный набор уравнений и неравенств, описывающий условия эксплуатации уязвимости.

Противодействие DEP и ASLR посредством использования ROP. Эксплуатация переполнения буфера на стеке с использованием ROP-цепочек не сильно отличается от классического подхода с передачей управления на шелл-код или трамплин [10]. Для успешной эксплуатации переполнения буфера на стеке необходимо, чтобы адрес возврата из функции указывал на первый гаджет, а остальная часть цепочки располагалась выше по стеку. На рис.1 показано состояние стека в момент эксплуатации переполнения буфера при помощи ROP.

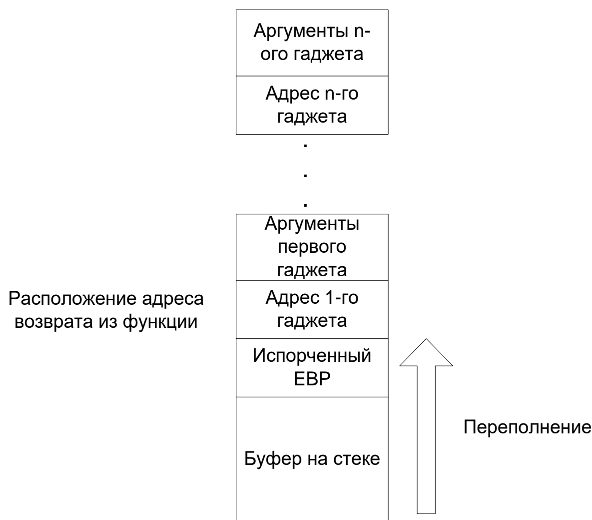


Рис.1. Состояние стека в момент эксплуатации при помощи ROP

Fig. 1. Stack frame in time of explotation using ROP

Таким образом, предикат безопасности будет состоять из набора уравнений, который описывает размещение ROP-цепочки, начиная от ячейки с адресом возврата из функции и далее выше по стеку.

Противодействие «канарейке» посредством эксплуатации уязвимости CWE-123. Как и при классическом переполнении буфера на стеке необходимо добавить набор уравнений, который описывает размещение шелл-кода в выбранной области памяти. Для успешной передачи управления на нужный адрес необходимо добавить два уравнения. Первое уравнение указывает на то, что адрес записи равен адресу ячейки, в которой хранится адрес возврата из функции. Второе уравнение указывает на то, что записываемое значение равно адресу, по которому располагается шелл-код. Объединение эти уравнения с предикатом пути позволит получить набор уравнений, решением которых является эксплойт, способный работать при наличии «канарейки» на стеке.

Противодействие DEP, ASLR и «канарейке». Для того чтобы успешно осуществить эксплуатацию в рамках одновременной работы трех механизмов защит, необходимо выполнение условий, описанных во втором разделе. На рис. 2 показано состояние стека и GOT в момент перехвата потока управления.

Рис.2. Состояние стека и GOT в момент перехвата потока управления

Fig 2. Stack frame and GOT in time of control flow hijack

Уязвимость CWE-123 используется для передачи управления на гаджет-трамплин, который сдвинет указатель стека на нужное смещение. Для этого

требуется добавить два уравнения. Первое уравнение указывает на то, что адрес записи равен ячейки из GOT, в которой хранится адрес вызываемой в дальнейшем библиотечной функции. Второе уравнение указывает на то, что записываемое значение равно адресу гаджета-трамплина. После выполнения гаджета-трамплина указатель стека сместится вверх в область локальных переменных, находящуюся под контролем атакующего. В этой области необходимо разместить основную часть ROP-цепочки. Для этого следует добавить соответствующие уравнения.

4. Реализация методов и результаты практического применения

Предложенные методы реализованы в виде программного инструмента. Метод использует ранее разработанные и реализованные средства: повышение уровня представления трассы машинных команд, а также модель процессора общего назначения. Для решения системы символьных уравнений и неравенств, полученной в результате построения предиката пути и предиката безопасности, используется SMT-решатель Z3 [17].

Разработанное программное средство было протестировано на нескольких примерах.

Недостаточная помеченность. Механизм добавления дополнительных ограничений на символьные данные применялся для противодействия возникающей недостаточной помеченности при генерации эксплойта для уязвимости переполнения буфера на стеке в программе *faad* из одноименного пакета. В качестве гостевой системы для получения трассы выполнения использовался 32-х битный Debian 8.3.0. Защитные механизмы, такие как DEP и ASLR, были отключены. Недостаточная помеченность возникала при копировании функцией *vsscanf* буфера с контролируемыми данными на стек. Во время копирования функция *vsscanf* проверяет наличие в буфере-источнике непечатаемых символов (whitespace). При обнаружении таких символов копирование заканчивается. В трассе выполнения такие проверки реализованы с использованием адресных зависимостей. Для того чтобы компенсировать отсутствие таких проверок, аналитик добавил необходимые ограничения (отсутствие пробелов) на параметр буфера-источника функции *vsscanf*. В результате чего был сгенерирован работоспособный эксплойт.

Избыточная помеченность. Механизм прекращения отслеживания параметров функций применялся для противодействия избыточной помеченности во время генерации эксплойта для уязвимости переполнения буфера на стеке в программе *blast2* из одноименного пакета. В качестве гостевой системы для получения трассы выполнения использовался 32-х битный Debian 6.0.10. Защитные механизмы, такие как DEP и ASLR, были отключены. Во время выполнения программа многократно вызывает функцию *malloc*, параметр которой зависит от входных данных. Таким образом, в

обработку попадают инструкции, относящиеся к работе функции *malloc*, что увеличивает размер полученной системы символьных уравнений. Во избежание такой ситуации аналитик составил описание функции *malloc* с указанием параметра, с которого необходимо снять пометку. Это позволило сократить время генерации эксплойта с 6 часов до 21 минуты.

Противодействие DEP и ASLR с помощью ROP. При генерации эксплойта для уязвимости переполнения буфера на стеке в программе *zsnes* из одноименного пакета использовался метод генерации эксплойта с помощью ROP. Код полезной нагрузки, вызывающий интерпретатор командной оболочки, был составлен в виде ROP-цепочки. Гаджеты были получены из исполняемого файла программы, который не является позиционно-независимым кодом. В качестве гостевой системы для получения трассы выполнения использовался 32-х битный Debian 8.3.0. Защитные механизмы, такие как DEP и ASLR, были включены. В результате был получен эксплойт, способный функционировать в условиях работы DEP и ASLR.

Эксплуатация уязвимости переполнения буфера на стеке в программе уровня ядра ОС. Для проведения эксперимента по эксплуатации уязвимости в программе, которая работает на уровне ядра операционной системы, был подготовлен модельный пример модуля ядра операционной системы Linux. В этом примере присутствовала уязвимость переполнения буфера на стеке. В качестве гостевой системы для получения трассы выполнения использовался 32-х битный Debian 8.3.0, в котором работает ASLR. Для эксплуатации использовался метод генерации эксплойта для переполнения буфера на стеке с использованием трамплинов. Поиск трамплина происходил в образе ядра операционной системы. Таким образом, был сгенерирован работоспособный эксплойт.

Эксплуатация ошибки типа CWE-123. Эксплуатация ошибки типа CWE-123 проводилась на примере из работы [18]. В примере происходит переполнение буфера на стеке с последующей перезаписью вышележащего по стеку указателя, по которому в дальнейшем происходит запись контролируемых значений. Данный пример был модифицирован следующим образом: переполняемый буфер и указатель были объединены в структуру, которую расположили на стеке, и программа была скомпилирована компилятором *gcc* с добавлением опции *-fstack-protector-all*, что привело к добавлению канарейки в уязвимую функцию. В качестве гостевой системы для получения трассы выполнения использовался 32-х битный Debian 8.3.0. Защитные механизмы, такие как DEP и ASLR, были отключены. Применение метода генерации эксплойта для ошибки типа CWE-123 с передачей управления на код полезной нагрузки позволило получить рабочий эксплойт.

Эксплуатация ошибки типа CWE-123 при наличии DEP, ASLR и «канарейки». Для проведения экспериментов по эксплуатации ошибки типа CWE-123 при наличии DEP, ASLR и «канарейки» был разработан модельный пример программы, фрагмент листинга которой приведен ниже.

```
1  typedef struct
2  {
3   char name[64];
4   char *bio;
5  } Person;
6
7  int
8  main(int argc, char *argv[])
9  {
10  int cnt = argc / 2;
11  Person persons[cnt];
12
13  for (int i = 0; i != cnt; ++i)
14  {
15         persons[i].bio = malloc(strlen(argv[2 * i + 2]) + 1);
16         strcpy(persons[i].name, argv[2 * i + 1]);
17         strcpy(persons[i].bio, argv[2 * i + 2]);
18  }
```

Листинг 1. Пример программы с ошибкой CWE-123

Listing 1. Program example with CWE-123 defect

Программа была скомпилирована компилятором *gcc* с опцией *-fstack-protector-all*. Из-за того, что буфер и указатель являются полями структуры, их переразмещение компилятором не происходит, и указатель располагается выше на стеке. Благодаря отсутствию проверки на размер при копировании в строке 16 происходит переполнение буфера, которое приводит к перезаписи указателя. Затем в строке 17 уже по перезаписанному указателю выполняется копирование контролируемых данных, что порождает возникновение ошибки типа CWE-123. Для обхода защит DEP и ASLR стоит воспользоваться технологией ROP, но при наличии «канарейки» нельзя перезаписать адрес возврата из функции *main* путем непосредственного переполнения буфера. Для этого стоит воспользоваться эксплуатацией ошибки типа CWE-123, но из-за работы ASLR неизвестен адрес ячейки, в которой храниться адрес возврата из функции *main*. Запись выполнится в .got-слот функции *strlen*, поскольку размещение GOT не рандомизировано, а сама функция *strlen* будет вызвана на очередной итерации цикла. Записываемым значением служит адрес специального гаджета-трамплина, который сдвигает указатель стека вверх на фиксированное значение. После вызова функции *strlen* управление будет передано на гаджет-трамплин. Гаджет-трамплин подбирается таким образом, что в результате его выполнения указатель стека будет ссылаться на контролируемую область памяти. В этой области размещается основная часть ROP-цепочки, выполняющая код полезной нагрузки. Используя описанный

выше метод, удалось автоматически сгенерировать работоспособный эксплойт.

5. Заключение

В настоящее время в мире активно исследуются возможности автоматической генерации эксплойтов. Инструмент AEG [5] производит поиск уязвимости переполнения буфера и уязвимости форматной строки, используя исходные тексты программы. Эксплуатация найденных уязвимостей происходит уже посредством анализа бинарного кода. Инструмент MAYHEM [6] является развитием AEG, он осуществляет поиск и эксплуатацию, анализируя только бинарный файл программы. Эти инструменты находятся в закрытом доступе, а в публикациях нет информации о том, как происходит противодействие современным защитным механизмам. Система CRAX [7], основанная на полносистемном символьном выполнении, представленном в системе S2E [19], позволяет генерировать эксплойты при наличии входных данных, приводящих к аварийному завершению программы. В публикации по системе CRAX предлагаются подходы к генерации эксплойтов для уязвимостей переполнения буфера на стеке и на куче, а также уязвимости форматной строки. Описываются способы обхода защитных механизмов DEP и ASLR. Реализация системы, представленная в открытых источниках, позволяет генерировать эксплойты только для переполнения буфера на стеке без учета работы защитных механизмов. Система REX [9] дает возможность генерировать эксплойты для уязвимостей переполнения буфера на стеке в программах для операционной системы Linux при работе DEP и ASLR.

В статье представлен метод генерации входных данных, подтверждающих наличие в программе критических ошибок переполнения буфера на стеке и ошибки типа CWE-123. Данный метод основан на символьном выполнении бинарного кода и учитывает работу современных защитных механизмов, таких как DEP, ASLR и «канарейка». Применение предложенного метода позволит выявлять среди зафиксированных аварийных завершений крайне важные ситуации, когда использование дефекта приводит к выполнению произвольного кода, несмотря на работу современных защитных механизмов.

Для рассмотренных в данной статье примеров составлялись ROP-цепочки, что потребовало минимальной автоматизации, а именно классификации найденных в программе ROP-гаджетов. Предложенный метод реализован в виде программного инструмента.

Перспективными направлениями дальнейших исследований являются исследование вопросов, связанных с уязвимостями переполнения буфера на куче, а также исследование возможности перехвата потока данных.

Список литературы

[1]. One A. Smashing the stack for fun and profit. Phrack magazine, v. 7, №. 49, 1996, pp. 14-16.
[2]. Durden T. Bypassing pax aslr protection. Phrack Magazine, v. 59, №. 9, 2002, pp. 9.
[3]. Nergal. The advanced return-into-lib (c) exploits: PaX case study. Phrack Magazine, Volume 58, Issue 4, 2001.
[4]. Bulba K. Bypassing stackguard and stackshield. 2000.
[5]. T. Avgerinos, S. K. Cha, Alexandre Rebert, Edard J. Schwartz, Maverick Woo, and D.Brumley. AEG: Automatic exploit generation. Commun. ACM, №2, 2014.
[6]. Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert and David Brumley. Unleashing MAYHEM on Binary Code. IEEE Symposium on Security and Privacy, 2012.
[7]. Huang S. K. et al. Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations. Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on., IEEE, 2012, pp. 78-87.
[8]. Hovav Shacham. The Geometry of Innocent Flash on the Bone: Return-into-libc without Function Calls (on the x86). 2007 ACM Conference on Computer and Communications Security (CCS), Proceedings of CCS 2007, pp. 552-561.
[9]. Shoshitaishvili Y. et al. SOK:(State of) The Art of War: Offensive Techniques in Binary Analysis. 2016 IEEE Symposium on Security and Privacy (SP), IEEE, 2016, pp. 138-157.
[10]. Padaryan V. A., Kaushan V. V., Fedotov A. N. Automated exploit generation for stack buffer overflow vulnerabilities. Programming and Computer Software, v. 41, №. 6, 2015, pp. 373-380. DOI: 10.1134/S0361768815060055.
[11]. Падарян В. А., Соловьев М. А., Кононов А. И. Моделирование операционной семантики машинных инструкций. Программирование, № 3, 2011 г., стр. 50-64.
[12]. J. Kim, T. Kim and E. G. Im. Survey of dynamic taint analysis. 2014 4th IEEE International Conference on Network Infrastructure and Digital Content, Beijing, 2014, pp. 269-272.
[13]. Ubuntu security features. https://wiki.ubuntu.com/Security/Features
[14]. Malloc Des-Maleficarum. Phrack magazine, v. 13, issue 66, 2009.
[15]. Nurmukhametov, AR; Kurmangaleev, Sh F; Kaushan, VV; Gaissaryan, SS. Application of compiler transformations against software vulnerabilities exploitation. Programming and Computer Software, v. 41, № 4, 2015, pp. 231-236. DOI: 10.1134/S0361768815040052.
[16]. Mauro Conti, Stephen Crane, Tommaso Frassetto, Andrei Homescu, Georg Koppen, Per Larsen, Christopher Liebchen, Mike Perry, Ahmad-Reza Sadeghi. Selfrando: Securing the Tor Browser against De-anonymization Exploits. In Proceedings of the 16th Privacy Enhancing Technologies Symposium (PETS 2016), in press, Darmstadt, Germany, July 19-22, 2016.
[17]. Nikolaj Bjorner, Leonardo de Moura. Z3: Applications, Enablers, Challenges and Directions. Sixth International Workshop on Constraints in Formal Verification Grenoble, 2009.
[18]. Heelan, S. Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities. M.Sc. thesis. University of Oxford, Oxford, U.K., Sept. 3, 2009.
[19]. Chipounov V., Kuznetsov V., Candea G. S2E: a platform for in-vivo multi-path analysis of software systems. ACM SIGPLAN Notices, v. 46, №. 3, 2011, pp. 265-278.

[20]. Vanegue J., Heelan S., Rolles R. SMT Solvers in Software Security. WOOT, 2012, pp. 85-96.
[21]. CWE-123, https://cwe.mitre.org/data/definitions/123.html

Software defect severity estimation in presence of modern defense mechanisms*

[1]*A.N. Fedotov <fedotoff@ispras.ru>*
[1,2]*V.A. Padaryan <vartan@ispras.ru>*
[1]*V.V. Kaushan <korpse@ispras.ru>*
[1]*Sh.F. Kurmangaleev <kursh@ispras.ru>*
[1]*A.V. Vishnyakov <vishnya@ispras.ru>*
[1]*A.R. Nurmukhametov <oleshka@ispras.ru>*
[1]*Institute for System Programming of the Russian Academy of Sciences,*
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia
[2]*Lomonosov Moscow State University,*
GSP-1, Leninskie Gory, Moscow, 119991, Russia

Abstract. This paper introduces a refined method for automated exploitability evaluation of found program bugs. During security development lifecycle a significant number of crashes is detected in programs. Because of limited resources, bug fixing is time consuming and needs prioritization. It should be the matter of highest priority to fix exploitable bugs. Automated exploit generation technique is used to solve this problem in practice. Generated exploit confirms the presence of a critical vulnerability. However, state-of-the-art publications omit modern defense mechanisms preventing exploitation. It results in lowering of an evaluation quality. This paper considers modern vulnerability exploitation prevention mechanisms. An evaluation of their prevalence and efficiency is also presented. The method can be applied to program binaries and doesn't require any debug information. Proposed method is based on symbolic interpretation of traces obtained by a full-system emulator. Our method can demonstrate a real exploitability for stack buffer overflow vulnerability with write-what-where condition even when DEP, ASLR, and "canary" operate together. The implemented method capabilities were shown on model examples and real programs.

Keywords: critical vulnerability; binary code; symbolic execution.

DOI: 10.15514/ISPRAS-2016-28(5)-4

For citation: Fedotov A.N., Padaryan V.A., Kaushan V.V., Kurmangaleev Sh.F., Vishnyakov A.V., Nurmukhametov A.R. Software defect severity estimation in presence of modern defense mechanisms. *Trudy ISP* RAN/Proc. ISP RAS, vol. 28, issue 5, 2016. pp. 73-92 (in Russian). DOI: 10.15514/ISPRAS-2016-28(5)-4

* The paper was supported by RFBR grant # 16-29-09632

References

[1]. One A. Smashing the stack for fun and profit. Phrack magazine, v. 7, №. 49, 1996, pp. 14-16.
[2]. Durden T. Bypassing pax aslr protection. Phrack Magazine, v. 59, №. 9, 2002, pp. 9.
[3]. Nergal. The advanced return-into-lib (c) exploits: PaX case study. Phrack Magazine, Volume 58, Issue 4, 2001.
[4]. Bulba K. Bypassing stackguard and stackshield. 2000.
[5]. T. Avgerinos, S. K. Cha, Alexandre Rebert, Edard J. Schwartz, Maverick Woo, and D.Brumley. AEG: Automatic exploit generation. Commun. ACM, №2, 2014.
[6]. Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert and David Brumley. Unleashing MAYHEM on Binary Code. IEEE Symposium on Security and Privacy, 2012.
[7]. Huang S. K. et al. Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations. Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on., IEEE, 2012, pp. 78-87.
[8]. Hovav Shacham. The Geometry of Innocent Flash on the Bone: Return-into-libc without Function Calls (on the x86). 2007 ACM Conference on Computer and Communications Security (CCS), Proceedings of CCS 2007, pp. 552-561.
[9]. Shoshitaishvili Y. et al. SOK:(State of) The Art of War: Offensive Techniques in Binary Analysis. 2016 IEEE Symposium on Security and Privacy (SP), IEEE, 2016, pp. 138-157.
[10]. Padaryan V. A., Kaushan V. V., Fedotov A. N. Automated exploit generation for stack buffer overflow vulnerabilities. Programming and Computer Software, v. 41, №. 6, 2015, pp. 373-380. DOI: 10.1134/S0361768815060055.
[11]. Padaryan V.A., Solov'ev M.A., Kononov A.I. Simulation of operational semantics of machine instructions. Programming and Computer Software, May 2011, Volume 37, Issue 3, pp 161-170, DOI 10.1134/S0361768811030030.
[12]. J. Kim, T. Kim and E. G. Im. Survey of dynamic taint analysis. 2014 4th IEEE International Conference on Network Infrastructure and Digital Content, Beijing, 2014, pp. 269-272.
[13]. Ubuntu security features. https://wiki.ubuntu.com/Security/Features
[14]. Malloc Des-Maleficarum. Phrack magazine, v. 13, issue 66, 2009.
[15]. Nurmukhametov A.R.; Kurmangaleev Sh.F.; Kaushan V.V.; Gaissaryan S.S. Application of compiler transformations against software vulnerabilities exploitation. Programming and Computer Software, v. 41, № 4, 2015, pp. 231-236. DOI: 10.1134/S0361768815040052
[16]. Mauro Conti, Stephen Crane, Tommaso Frassetto, Andrei Homescu, Georg Koppen, Per Larsen, Christopher Liebchen, Mike Perry, Ahmad-Reza Sadeghi. Selfrando: Securing the Tor Browser against De-anonymization Exploits. In Proceedings of the 16th Privacy Enhancing Technologies Symposium (PETS 2016), in press, Darmstadt, Germany, July 19-22, 2016.
[17]. Nikolaj Bjorner, Leonardo de Moura. Z3: Applications, Enablers, Challenges and Directions. Sixth International Workshop on Constraints in Formal Verification Grenoble, 2009.
[18]. Heelan, S. Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities. M.Sc. thesis. University of Oxford, Oxford, U.K., Sept. 3, 2009.
[19]. Chipounov V., Kuznetsov V., Candea G. S2E: a platform for in-vivo multi-path analysis of software systems. ACM SIGPLAN Notices, v. 46, №. 3, 2011, pp. 265-278.

[20]. Vanegue J., Heelan S., Rolles R. SMT Solvers in Software Security. WOOT, 2012, pp. 85-96.
[21]. CWE-123, https://cwe.mitre.org/data/definitions/123.html

Применение диверсифицирующих и обфусцирующих преобразований для изменения сигнатуры программного кода[*]

А.Р. Нурмухаметов <oleshka@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. Развитие методов обнаружения вредоносных приложений привело к развитию специальных технологий, которые помогают вредоносным приложениям оставаться незамеченными. Многие из них направлены на изменение сигнатуры программного кода. Для академического изучения этих методов и кода, получаемого с их помощью, необходимо разработать инфраструктуру для автоматизированного изменения сигнатуры имеющейся программы. В данной работе приводится результаты работы по разработке и реализации такого автоматизированного инструмента на базе компиляторной инфраструктуры LLVM и инфраструктуры инструментации и трансформации бинарного кода Syzygy. В рамках этих инфраструктур были реализованы диверсифицирующие и обфусцирующие преобразования, которые были направлены на изменение сигнатуры исполняемого файла. Для работы этих инструментов требуется соответственно наличие исходного кода или отладочной информации. Были разработаны и реализованы следующие преобразования: вставка мертвого кода, перестановка местами инструкций, перестановка местами базовых блоков, перестановка местами функций в модуле, замена инструкций на эквивалентные, шифрование константных буферов данных. По результатам проведенных работ была продемонстрирована работоспособность данного подхода на реальных примерах, а также выявлены недостатки предложенного подхода и пути дальнейшего развития.

Ключевые слова: диверсификация; обфускация; сигнатурный поиск; вредоносные приложения.

DOI: 10.15514/ISPRAS-2016-28(5)-5

Для цитирования: Нурмухаметов А.Р. Применение диверсифицирующих и обфусцирующих преобразований для изменения сигнатур программного кода. Труды ИСП РАН, том 28, вып. 5, стр. 93-104, 2016 г. DOI: 10.15514/ISPRAS-2016-28(5)-5

[*] Работа поддержана грантом РФФИ 14-01-00462 А

1. Введение

Изучение методов защиты вредоносных приложений от средств обнаружения – важная область компьютерной безопасности. В то время как многие другие области тестирования на проникновения хорошо изучены и их методы документированы, то о тестировании и обходе защит от вредоносных приложений опубликовано не так много работ. До настоящего времени инциденты, связанные с заражениями внутренних сетей крупных организаций, часто встречаются и приносят ощутимые убытки этим организациям. Кроме этого, такие виды заражений являются подходящим средством для проведения целенаправленных атак на конкретные организации. Вредоносные приложения используются как для вывода из строя элементов инфраструктуры, так и для получения доступа к непубличной информации. Исходя из этого, важно уметь оценивать эффективность защитных механизмов программного обеспечения, направленного на обнаружение вредоносных приложений.

Обнаруженное вредоносное приложение не представляет собой никакой опасности. Поэтому активно развиваются методы сокрытия такого кода от систем обнаружения. Для решения этой трудной задачи применяются следующие технологии: полиморфизм, метаморфизм, упаковщики, крипторы [10-14]. Большинство из этих методов сосредоточены на том, чтобы снабдить вредоносное приложение дополнительной функциональностью, позволяющей изменять себя во время распространения или работы для ухода от обнаружения. В данной работе исследуется возможность изменения сигнатур программного кода с помощью обфусцирующих и диверсифицирующих компиляторных преобразований.

Методы обнаружения вредоносных приложений основаны на сигнатурном поиске, эмуляции кода, эвристических анализах. Сигнатурный поиск [3, 30] основывается на поиске известных шаблонов вредоносного кода в бинарном файле. Хотя этот метод имеет свои ограничения, но также он имеет неоспоримые преимущества: низкое количество ложных срабатываний и высокая скорость работы. Благодаря своей простоте и аккуратности сигнатурный поиск имеет широкое применение для обнаружения вредоносных приложений. Сигнатурные методы плохо применимы для обнаружения новых видов вредоносных приложений, поскольку требуют предварительного создания сигнатуры и добавления ее в базу знаний инструмента поиска, и доставки всем его пользователям.

В данной работе описывается подход, основанный на применении методов диверсификации и обфускации и позволяющий разработать инструмент для автоматизированного изменения сигнатуры приложения в исследовательских целях. Предложенные методы позволяют сгенерировать потенциально неограниченное количество различных версий бинарного файла, которые имеют различные сигнатуры, но одинаковую функциональность. Кроме того, приводятся результаты экспериментальной проверки, обсуждаются

преимущества и недостатки использованного подхода. Предлагаемый подход требует для защиты наличия исходного кода приложения или отладочной информации.

Данная работа состоит из шести частей. Первая глава представляет собой введение, в котором обосновывается актуальность данной работы. Во второй главе приводится обзор методов защиты вредоносных приложений. В третьей главе приводится описание предлагаемого подхода. В четвертой главе рассказывается про используемые преобразования и обсуждаются детали реализации. В пятой главе приведены результаты экспериментальной проверки разработанных инструментов. В шестой главе подводятся итоги и намечаются планы дальнейшего развития.

2. Обзор методов защиты

В данном разделе приводится описание методик, применяемых при разработке вредоносного программного обеспечения для обхода методов обнаружения, а также обсуждаются сильные и слабые стороны каждого способа с точки зрения способов защиты.

Крипторы. Широко применяются различные крипторы [9] для обхода сигнатурных средств поиска. Общая идея их применения заключается в использовании шифрования кода вредоносного приложения. Шифрование применяется в качестве обратимого преобразования, для которого криптографическая стойкость, как правило, не является необходимой в контексте изменения сигнатуры бинарного кода. Зашифрованный двоичный код снабжается специальным расшифровщиком. При запуске управление передается на код расшифровщика, который производит дешифрование непосредственно в памяти в процессе выполнения. Код расшифровщика остается неизменным, поэтому средства сигнатурного поиска могут составлять характерные сигнатуры для расшифровщиков и производить их поиск в подозрительном программном обеспечении.

Олигоморфизм. Последовательным шагом в развитии предыдущей технологии является техника олигоморфизма [9, 10]. Она характеризуется изменением кода декриптора при самовоспроизведении от одной версии к другой. Простейший вариант – наличие различных вариантов декриптора, написанных вручную. В процессе генерации новых поколений происходит случайный выбор декриптора. В работе [11] приводится пример, использующий различные декрипторы. Данный подход ограничен тем, что средство сигнатурного поиска может иметь в своем распоряжении шаблоны для самых популярных декрипторов и успешно их обнаруживать. **Полиморфизм.** Техника полиморфизма [9, 10, 11] является логичным развитием идей, на которых основаны крипторы и олигоморфные вредосные приложения. Она обладает способностью генерировать больше разных декрипторов. Цель состоит в том, чтобы генерировать уникальный код для каждой новой версии, не оставляя общей сигнатуры. Применяются методы

трансформации программного кода: замена инструкций на эквивалентные, перестановка местами инструкций, переименование регистров, вставка мертвого кода для достижения этой цели. Для противодействия такому подходу используется метод эмуляции кода. Во время эмуляции происходит расшифровка кода и сигнатурный поиск позволяет найти характерные куски кода.

Метаморфизм. Техника метаморфизма [9, 11, 12, 23] расширяет возможности предыдущих этапов и генерирует не только новый декриптор, но и полностью новое тело вредоносного приложения при сохранении его поведения. Основой для формирования следующего поколения является код в некотором его представлении, доступном для анализа и трансформации (зачастую это некоторое промежуточное представление). Данный код содержит всю необходимую последовательность действий и транслируется в исполняемый двоичный код. При этом процесс трансляции включает в себя некоторую неопределенность, которая позволяет генерировать различные сигнатуры. Для обнаружения таких вредоносных приложений используется поведенческий и эвристический анализ. Существенным минусом эвристического анализа является его точность, т.е. количество ложных срабатываний.

Протекторы. Другим примером защиты являются протекторы [15-17]. Они представляют собой виртуальные машины, построенные на основе некоторых случайно-генерированных машинных архитектур, могут содержать приемы антиотладки и методы запутывания бинарного кода. Самыми известными примерами являются: VMProtect [18], Themida [19], ASProtect [20]. Опишем подробнее подход, осуществлённый при реализации одного из протекторов. Он основан на построении виртуальной машины некоторого несуществующего процессора и преобразовании обычного кода в код для этого процессора с последующим выполнением его на этой виртуальной машине. Эта технология была реализована на основе LLVM и QEMU. Случайным образом машинные коды инструкций x86 менялись друг на друга с соблюдением ряда нетривиальных ограничений на взаимозаменяемость. Для полученной архитектуры набора инструкций автоматически генерировались свои файлы описания целевой архитектуры компилятора LLVM, после чего он собирался для новой архитектуры. Кроме этого, вносились соответствующие изменения в QEMU для поддержания возможности эмуляции новой архитектуры. Исходный код вредоносного приложения транслировался полученным компилятором, снабжался эмулятором и логикой, запускающей данный код на эмуляторе. Авторы статьи [20] приводят цифры, показывающие практически неограниченную мощность диверсификации сигнатуры исходного кода данного подхода.

3. Описание предлагаемого подхода

Методы трансформации программ, используемые компиляторами, хорошо изучены и развиты в инструментальных средствах. Существующие

промышленные компиляторные инфраструктуры Clang/LLVM [1, 22], GCC [2] содержат в себе большое количество преобразований, направленных на оптимизацию программного кода. Они используют собственные внутренние представления программы.

Компиляторные инфраструктуры предоставляют возможность для удобной реализации новых преобразований над кодом программы в виде промежуточного представления. Преобразования запускаются друг за другом. Каждое из них изменяет промежуточное представление, после чего запускается следующее преобразование. Автоматическое изменение кода программы внутри компиляторных инфраструктур с целью изменения сигнатуры представляется наиболее логичным подходом.

В данной работе предлагается использовать компиляторную инфраструктуру для реализации специальных преобразований над промежуточным представлением, которые бы приводили к изменению кода приложения таким образом, чтобы изменить характерные признаки его сигнатуры. За основу такого решения был взят разработанный обфусцирующий компилятор [26]. Стоит отметить, что данный компилятор разработан на основе Clang/LLVM. Это накладывает некоторые ограничения на его использование. Основное ограничение заключается в недостаточной поддержке платформозависимых особенностей, например, ограниченность поддержки кода, специфичного для Microsoft Visual Studio Compiler. Данная компиляторная инфраструктура является закрытой и не предоставляет возможности для своего расширения.

Для изменения вредоносных приложений под Windows, не компилируемых Clang/LLVM, в данной работе предлагается использовать инструмент Syzygy от Google [24]. Syzygy - набор утилит для послесборочной инструментации и оптимизации 32-битных приложений для Windows в условиях наличия отладочной информации. Выполняет задачи инструментации кода для подсчета метрик, покрытия и профилирования кода. Собранная информация используется для оптимизации запуска приложения и поиска ошибок с памятью. Часть преобразований из обфусцирующего компилятора перенесена в Syzygy для обеспечения возможности генерации различных исполняемых файлов. Отметим, что дизассемблирование в отсутствие отладочной информации является неразрешимой задачей в общем случае, поэтому наличие отладочной информации необходимо для работы этого инструмента.

4. Используемые методы трансформации кода

Диверсифицирующие и обфусцирующие трансформации [7, 8, 10] напоминают компиляторные по своему устройству, но преследуют другие цели. Цель оптимизирующих преобразований – увеличить скорость работы программы или уменьшить занимаемое место. Цель диверсифицирующих преобразований – изменить код приложения, не изменяя его функциональности. Целью обфусцирующих преобразований является запутывание приложения таким образом, чтобы осложнить понимание

принципов работы алгоритмов приложения и структур данных. Диверсифицирующие и обфусцирующие трансформации зависят от случайной величины, получаемой от генератора случайных чисел. Все описываемые ниже преобразования были реализованы в рамках обфусцирующего компилятора, разработанного в ИСП РАН [27, 28]. Часть из них была также реализована в инструменте бинарной инструментации Syzygy.

Вставка недостижимого кода. *Недостижимый код* – это код, который не выполняется в программе ни при каких входных данных. Вставка недостижимого кода может быть реализована различными способами [25]. Недостижимый код вставляется в граф потока управления в виде нового базового блока. Вход в этот базовый блок закрывается непрозрачным предикатом [26]. Это преобразование может быть использовано с целью увеличения метрики сигнатурного различия оригинального и трансформированного приложения.

Вставка бесполезных инструкций. Данное преобразование привносит в код программы инструкции, которые, в отличии от недостижимого кода выполняются, но никак не влияют выходные данные или наблюдаемое поведение приложения [29]. Количество привносимых инструкций может быть любым, как и их сложность, поэтому данной трансформацией можно добиться континуального множетсва вариантов программы. В обоих инструментах была реализована вставка инструкций и их последовательностей, которые ничего не делают: nop; xchg eax, eax; xchg eax, ebx, xchg eax, ebx.

Замена инструкций на эквивалентные. Существуют различные машинные инструкции для выполнения одних и тех же действий, поэтому возможно заменять инструкции друг на друга. Данное преобразование было реализовано внутри компиляторной инфраструктуры LLVM на уровне машинно-зависимого кода как peephole оптимизации. Трансформация использует набор шаблонов для замены инструкций. Большая часть шаблонов связана с заменой логико-арифметических выражений на эквивалентные, но выраженные через другие операции.

Перестановка местами инструкций. С помощью стандартного планировщика инструкций перестановка местами инструкций была реализована внутри компиляторной инфраструктуры LLVM. Кроме того, данное преобразование было реализовано и в инфраструктуре Syzygy. Для этого был реализован дополнительный анализ зависимостей между инструкциями. Результаты данного анализа позволяют определять вид зависимостей между инструкциями и на основании этого определять те инструкции, которые можно менять друг с другом местами внутри одного базового блока.

Перестановка базовых блоков. Базовый блок – линейная последовательность инструкций, оканчивающаяся единственной инструкцией передачи управления следующему базовому блоку. Последовательность

раскладки базовых блоков в образе программы можно менять, что позволяет менять сигнатуру приложения. Перестановка базовых блоков в случайном порядке в пределах одной функции была реализована внутри инструмента Syzygy.

Перестановка местами функций в модуле. Перестановка местами функций дает дополнительную возможность по изменению сигнатуры программы на более крупнозернистом уровне по сравнению с перестановкой базовых блоков. [21]. Данное преобразование было реализовано внутри компиляторной инфраструктуры LLVM. Функции в пределах одного модуля трансляции переставлялись в случайном порядке следования.

Шифрование константных буферов. Константные буферы, хранящиеся в памяти процесса, могут быть источником характерных сигнатур и причиной обнаружения. Преобразование [26], маскирующее такие буферы, предназначено для изменения их сигнатуры. Шифрование выполняется следующим образом: все константные буферы кроме тех, для которых невозможно гарантировать консервативность, шифруются; в модуль добавляются функции расшифровки и шифровки. Перед каждым использованием буфера вставляется расшифровывающая функция, а после шифрующая. Данное преобразование было реализовано в обфусцирующем компиляторе, разрабатываемом в ИСП РАН [27, 28].

5. Результаты

Для экспериментальной проверки работоспособности описанного подхода были проведены исследования по запутыванию реальных образцов вредоносных приложений. В данной работе в качестве средства сигнатурного поиска использовался проект с открытым исходным кодом Clam [5]. Были взяты приложения из открытого набора вредоносных приложений с исходным кодом theZoo [6], который содержит в себе примеры под Windows. Стоит отметить, что не все примеры из этого набора успешно собираются и обнаруживаются средствами сигнатурного поиска. Для двух из них ‘NullBot’, ‘xTBot’ удалось получить запутанные образцы с помощью инструмента на основе Syzygy, которые не обнаруживались средством сигнатурного поиска. Для проверки обфусцирующего компилятора были использованы примеры вредоносных приложений с исходным кодом на языке C для Linux с ресурса [4]. К сожалению, из двадцати примеров только один собирается без ошибок и после оригинальной сборки обнаруживается средством сигнатурного поиска, а именно – vit. Запутывание его обфусцирующим компилятором привело к исчезновению из его кода характерной сигнатуры.

В ходе экспериментального тестирования разработанных преобразований был выявлен их недостаток. Он заключается в том, что данные преобразования носят случайный характер. Они производятся над теми местами в коде, над которыми их можно осуществить. Это приводит к тому, что, преобразования могут давать разный эффект в зависимости от начальной затравки. Например,

вредоносное приложение NullBot было запутано сто раз с разными значениями начальной затравки для линейного генератора псевдослучайных чисел. Характерная сигнатура обнаруживалась в этом приложении только в 19 случаях, тогда как в 81 случае ее в коде трансформированного приложения не присутствовало.

Направленность диверсифицирующих и обфусцирующих преобразований может быть исправлена путем создания специальных метрик, численно показывающих степень различия двух исполняемых файлов с точки зрения сигнатурного поиска.

6. Заключение

В данной работе описывается подход к разработке инструментального средства и соответствующей инфраструктуры, предназначенного для изучения возможных способов обфускации и диверсификации, которыми могут пользоваться вредоносные приложения. Данная инфраструктура может быть полезна для последующего поиска способов противодействия. Проведен обзор существующих методов обхода сигнатурных типов защиты. Предложен и реализован набор преобразований, направленных на достижение поставленной цели. Приведены результаты экспериментальной проверки предложенного подхода. Они показывают, что диверсифицирующие и обфусцирующие преобразования могут применяться для решения задачи изменения сигнатуры программного кода. Кроме того, обнаружено ограничение представленного подхода – ненаправленность проводимых преобразований. Обсуждаются способы исправления этих недостатков и перспективные пути развития предлагаемого подхода.

Список литературы

[1]. LLVM Compiler Infrastructure. http://llvm.org/
[2]. GCC Compiler Infrastructure. https://gcc.gnu.org/
[3]. Radare2. http://radare.org/r/
[4]. Malware Source Collection. http://vxheaven.org.
[5]. Clam. http://www.clamav.net/
[6]. Malware Open Source Collection. https://github.com/ytisf/theZoo
[7]. Source-Free Binary Mutation for Offense and Defense. V. Mohan. 2014.
[8]. Algorithmic Diversity for Software Security. https://arxiv.org/abs/1312.3891
[9]. You and K. Yim, "Malware Obfuscation Techniques: A Brief Survey," Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on, Fukuoka, 2010, pp. 297-300.
[10]. Ashu Sharma and S K Sahay. Article: Evolution and Detection of Polymorphic and Metamorphic Malwares: A Survey. International Journal of Computer Applications 90(2):7-11, March 2014.
[11]. Rad, B., Masrom, M. and Ibrahim, S. “Camouflage in Malware: From Encryption to Metamorphism”, International Journal of Computer Science and Network Security, 2012, 12: 74-83.

[12]. P. OKane, S. Sezer and K. McLaughlin, "Obfuscation: The Hidden Malware," in IEEE Security & Privacy, vol. 9, no. 5, pp. 41-47, Sept.-Oct. 2011. doi: 10.1109/MSP.2011.98
[13]. Tom Brosch, Maik Morgenstern AV-Test GmbH. Runtime Packers: The Hidden Problem? Black Hat USA'06. https://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Morgenstern.pdf
[14]. Ultimate Packer for eXecutables. http://upx.sourceforge.net/
[15]. ASPack. http://www.asprotect.ru/aspack.html
[16]. FSG. https://exelab.ru/_dl-nLh/pack/fsg20.rar
[17]. VMProtect. http://vmpsoft.com/
[18]. Themida. http://www.oreans.com/themida.php
[19]. ASProtect. http://www.asprotect.ru/asprotect64.html
[20]. AlphaPack Protector Report. https://github.com/graulito/alphapack
[21]. Application of Compiler Transformation Against Software Vulnerabilities Exploitation. A. Nurmukhametov, Sh. Kurmangaleev, V. Kaushan, S. Gaisaryan. Programming and Computer Software. 2015. V. 41, № 4. P. 231-236. Doi: 10.1134/S0361768815040052.
[22]. Lattner C. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL.
[23]. Philippe Beaucamps. Advanced Metamorphic Techniques in Computer Viruses. International Conference on Computer, Electrical, and Systems Science, and Engineering - CESSE'07, Nov 2007, Venice, Italy. 2007.
[24]. Syzygy Transformation Toolchain. url: https://github.com/google/syzygy/
[25]. Tamboli Teja. Metamorphic Code Generation from LLVM IR Bytecode: Master's thesis. San Jose State University. San Jose. 2013.
[26]. В. Иванников, Ш. Курмангалеев, А. Белеванцев [и др.]. Реализация запутывающих преобразований в компиляторной инфраструктуре LLVM. Труды ИСП РАН, том 26, вып. 1, 2014, стр. 327-342. DOI: 10.15514/ISPRAS-2014-26(1)-12.
[27]. Ш. Ф. Курмангалеев, В. П. Корчагин, В. В. Савченко [и др.]. Построение обфусцирующего компилятора на основе инфраструктуры LLVM. Труды ИСП РАН, том 23, 2012, стр. 77-92. DOI: 10.15514/ISPRAS-2012-23-5.
[28]. Курмангалеев Ш. Ф., Корчагин В. П., Матевосян Р. А. Описание подхода к разработке обфусцирующего компилятора Труды ИСП РАН, том 23, 2012, стр. 67-76. DOI: 10.15514/ISPRAS-2012-23-4.
[29]. Software Tamper Resistance: Obstructing Static Analysis of Programs: Tech. Rep.: Chenxi Wang, Jonathan Hill, John Knight [и др.]. Charlottesville, VA, USA: 2000.
[30]. Clam. http://www.clamav.net/

The Application of Compiler-based Obfuscation and Diversification for Program Signature Modification.

A.R. Nurmukhametov <oleshka@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. Development of malware detection techniques leads to the evolution of anti-detection techniques. In this paper we discuss possibility of creating an automatic tool for signature modification. In this article we describe our experience in designing and development of such tool. For signature modification in Linux programs we implemented a tool based on LLVM compiler infrastructure and for Windows programs we used post-link instrumentation and optimization tool Syzygy. The former approach requires program source code, while the latter assumes only the presence of debug information. Diversifying and obfuscating transformations were implemented in both cases with the aim of changing the signature of program to prevent matching them the known patterns. Implemented transformations are bogus code insertion, function permutation, instruction substitution, ciphering of constant buffer. As a result we demonstrate proof-of-concept examples which confirm that it is possible to automatically change of program signature for avoiding detection by signature-based analysis. Furthermore we explain drawbacks of this technique and discuss the further ways of development.

Keywords: diversification; obfuscation; signature analysis.

DOI: 10.15514/ISPRAS-2016-28(5)-5

For citation: Nurmukhametov A.R. The Application of Compiler-based Obfuscation and Diversification for Program Signature Modification. Trudy ISP RAN/Proc. ISP RAS, 2016, vol. 28, issue 5, 2016, pp. 93-104 (in Russian). DOI: 10.15514/ISPRAS-2016-28(5)-5

References

[1]. LLVM Compiler Infrastructure. http://llvm.org/
[2]. GCC Compiler Infrastructure. https://gcc.gnu.org/
[3]. Radare2. http://radare.org/r/
[4]. Malware Source Collection. http://vxheaven.org.
[5]. Clam. http://www.clamav.net/
[6]. Malware Open Source Collection. https://github.com/ytisf/theZoo
[7]. Source-Free Binary Mutation for Offense and Defense. V. Mohan. 2014.
[8]. Algorithmic Diversity for Software Security. https://arxiv.org/abs/1312.3891
[9]. You and K. Yim, "Malware Obfuscation Techniques: A Brief Survey," Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on, Fukuoka, 2010, pp. 297-300.

[10]. Ashu Sharma and S K Sahay. Article: Evolution and Detection of Polymorphic and Metamorphic Malwares: A Survey. International Journal of Computer Applications 90(2):7-11, March 2014.
[11]. Rad, B., Masrom, M. and Ibrahim, S. "Camouflage in Malware: From Encryption to Metamorphism", International Journal of Computer Science and Network Security, 2012, 12: 74-83.
[12]. P. OKane, S. Sezer and K. McLaughlin, "Obfuscation: The Hidden Malware," in IEEE Security & Privacy, vol. 9, no. 5, pp. 41-47, Sept.-Oct. 2011. doi: 10.1109/MSP.2011.98
[13]. Tom Brosch, Maik Morgenstern AV-Test GmbH. Runtime Packers: The Hidden Problem? Black Hat USA'06.https://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Morgenstern.pdf
[14]. Ultimate Packer for eXecutables. http://upx.sourceforge.net/
[15]. ASPack. http://www.asprotect.ru/aspack.html
[16]. FSG. https://exelab.ru/_dl-nLh/pack/fsg20.rar
[17]. VMProtect. http://vmpsoft.com/
[18]. Themida. http://www.oreans.com/themida.php
[19]. ASProtect. http://www.asprotect.ru/asprotect64.html
[20]. AlphaPack Protector Report. https://github.com/graulito/alphapack
[21]. Application of Compiler Transformation Against Software Vulnerabilities Exploitation. A. Nurmukhametov, Sh. Kurmangaleev, V. Kaushan, S. Gaisaryan. Programming and Computer Software. 2015. V. 41, № 4. P. 231-236. Doi: 10.1134/S0361768815040052.
[22]. Lattner C. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL.
[23]. Philippe Beaucamps. Advanced Metamorphic Techniques in Computer Viruses. International Conference on Computer, Electrical, and Systems Science, and Engineering - CESSE'07, Nov 2007, Venice, Italy. 2007.
[24]. Syzygy Transformation Toolchain. url: https://github.com/google/syzygy/
[25]. Tamboli Teja. Metamorphic Code Generation from LLVM IR Bytecode: Master's thesis. San Jose State University. San Jose. 2013.
[26]. Ivannikov V., Kurmangaleev S., Belevantsev A., Nurmukhametov A., Savchenko V., Matevosyan H., Avetisyan A. Implementing Obfuscating Transformations in the LLVM Infrastructure. Trudy ISP RAN/Proc. ISP RAS, vol. 26, issue 1, 2014. pp. 327-342. (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-12
[27]. Kurmangaleev S.F., Korchagin V.P., Savchenko V.V., Sargsyan S.S. Building obfuscation compiler based on LLVM infrastructure. Trudy ISP RAN/Proc. ISP RAS, vol. 23, 2012, pp. 77-92 (in Russian). DOI: 10.15514/ISPRAS-2012-23-5
[28]. Kurmangaleev S.F., Korchagin V.P., Matevosyan H.A. Description of the approach to development of the obfuscating compiler. Trudy ISP RAN/Proc. ISP RAS, vol. 23, 2012, pp. 67-76 (in Russian). DOI: 10.15514/ISPRAS-2012-23-4
[29]. Software Tamper Resistance: Obstructing Static Analysis of Programs: Tech. Rep.: Chenxi Wang, Jonathan Hill, John Knight [и др.]. Charlottesville, VA, USA: 2000.
[30]. Clam Antivirus Software. http://www.clamav.net/

Формализация определения ошибок при статическом символьном выполнении

В.К. Кошелев <vedun@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. Данная работа посвящена формализации понятия ошибочной ситуации при статическом анализе исходного кода, основанном на символьном выполнении. При использовании методов символьного выполнения для статического анализа необходимо пересматривать критерий выдачи ошибок, так как оригинальный критерий приводит к чрезмерному числу ложных срабатываний. Для решения этой проблемы предлагаются альтернативные определения ошибочной ситуации, сообщающие об ошибке только в том случае, когда она происходит на некотором множестве значений входных переменных. Примерами таких множеств являются, например, множество значений входных переменных, при которых управление пройдет через заданную точку программы, или множество значений, при которых управление пройдет по заданному пути в графе потока управления. В данной работе рассматриваются различные способы задания таких множеств начальных значений. Анализируются полученные таким образом определения ошибочных ситуаций. Приводятся алгоритмы, обнаруживающие данные ошибочные ситуации, а также доказывается их соответствие. Рассматривается практическое применение приведённых определений ошибочных ситуаций, а именно: классификация срабатываний инструментов статического анализа; учет неизвестных контракта вызова анализируемой функции; использование определения ошибочной ситуации в качестве запроса к SMT-решателю для поиска точного решения в соответствии с данными определением.

Ключевые слова: статический анализ; определение ошибочной ситуации; символьное выполнение.

DOI: 10.15514/ISPRAS-2016-28(5)-6

Для цитирования: Кошелев В.К. Формализация определения ошибок при статическом символьном выполнении. Труды ИСП РАН, том 28, вып. 5, 2016, стр. 105-118. DOI: 10.15514/ISPRAS-2016-28(5)-6

1. Введение

В настоящее время при разработке программного обеспечения широко используются методы автоматического поиска дефектов в программном коде. Одним из таких методов является статический анализ исходного кода

программы. Преимуществом статического анализа является возможность обнаруживать дефекты на путях выполнения, не покрытых при тестировании.

К сожалению, в общем случае задачи поиска дефектов при помощи статического анализа неразрешимы в силу теоремы Райса. На практике статические анализаторы ищут частные случаи ошибочных ситуаций. От того, ситуации какого типа обнаруживает статический анализатор, зависит как число пропущенных ошибок, так и количество ложных срабатываний. Анализаторы, не выдающие ложных срабатываний, способны обнаружить лишь простые дефекты. Не пропускающие ошибок анализаторы, напротив, оказываются неприменимыми для анализа промышленных программ из-за большого числа ложных срабатываний. Для эффективного использования статического анализа для промышленных программ необходимо использовать определение ошибочной ситуации, позволяющее обнаружить широкий спектр ошибочных ситуаций, сохраняя при этом высокий процент истинных срабатываний.

Один из сценариев использования статического анализатора предполагает регулярный анализ проектов во время разработки. Для исполнения этого сценария к статическому анализу предъявляется ряд требований. Во-первых, статический анализатор должен быть способен проводить анализ функции, не имея информации о точках её вызова. Это необходимо как для анализа ещё недописанного кода, так и для анализа библиотек. Во-вторых, при определении понятия ошибочной ситуации анализ должен учитывать наличие в коде вызовов неизвестных функций.

К сожалению, в существующих работах по статическому анализу [1] [2] [3], авторы акцентируют внимание на построении анализа, оставляя за скобками определение ошибочной ситуации. Целью данной работы является разработка формальных определений ошибочных ситуаций, позволяющих выдавать предупреждения с различной степенью уверенности относительно неизвестных значений.

Один из вариантов определения ошибочной ситуации, учитывающей контексты вызова и внешние функции, был рассмотрен в работе [4], посвящённой поиску внутрипроцедурной ошибки переполнения буфера. Данная работа формализует и расширяет определение ошибочной ситуации, использующееся в работе [4] для произвольных детекторов ошибок, основанных на методе символьного выполнения. Рассматриваются примеры реальных детекторов ошибок в программах на языке C#, включая доказательство соответствия с данными определениями ошибочных ситуации.

2. Статический анализ с помощью символьного выполнения

Символьное выполнение [5] широко используется для автоматической генерации тестов. В этом случае выполнение начинается с точки входа

программы, а входные данные программы считаются символьными. От символьного выполнения требуется подобрать входные данные, на которых произойдет ошибка. Пусть e – шаг символьного выполнения, P_e – предикат пути для шага e, а Err_e - условие возникновения ошибки на шаге e, тогда наличие ошибки на шаге e задается следующей формулой:

(1) $\exists \bar{\boldsymbol{s}} : P_e(\bar{\boldsymbol{s}}) \wedge Err_e(\bar{\boldsymbol{s}})$, $\bar{\boldsymbol{s}}$ – вектор символьных переменных

Однако задача статического анализа существенно отличается от задачи автоматической генерации ошибочных входных данных. Ключевым отличием является отсутствие необходимости в предоставлении набора данных, на которых произойдет ошибка.

Отсутствие этой необходимости позволяет рассматривать каждую функцию, представленную в программе, в качестве точки входа. В этом случае аргументы анализируемой функции и состояние памяти в момент её вызова параметризируются набором символьных переменных. Данный подход используется в ряде работ, включая [1], [2], [3]. Однако, в этом случае условие наличие ошибки (1) не может быть использовано ввиду чрезмерного количества ложных срабатываний. Рассмотрим следующий пример.

```
1) public class A {
2)    public int X;
3)    public static int Foo(A a, bool five) {
4)      if (five)
5)         return a.X;
6)      else
7)         return 5;
8)    }
9) }
```

Напишем условие ошибки NullReferenceException для точки (5).

$$\exists a, five : five \wedge a = null$$

Данная формула имеет решение $a = null, five = true$, однако функция Foo может иметь неявные предусловия, запрещающее переменной a иметь значение $null$. На практике, использование условия ошибки (1) для поиска NullReferenceException приведет к обнаружению ошибки практически в каждой функции.

Статический анализ приходится проводить, не имея полной информации о контракте анализируемой функции, включая её точное предусловие. Из-за чего, предположение о том, что неизвестные переменные никак не связаны между собой, слишком часто нарушается на практике.

Для учета неизвестного контракта функции предлагается ввести гибкое определение ошибки в конкретной точке программы, позволяющее в зависимости от целей анализатора описывать различные классы ошибочных ситуаций.

Предполагается, что анализ проводится над некоторым набором путей выполнения. Множество таких путей удобно описывать при помощи графа развертки. Ациклический ориентированный связанный граф с выделенными истоком и стоком $G' = \langle V', E', v'_{entry}, v'_{exit} \rangle$, будем называть графом развертки для ГПУ $G = \langle V, E, v_{entry}, v_{exit} \rangle$, если найдется функция соответствия $\psi : V' \to V$, такая что выполнено:

$$(2)\ \psi(v'_{entry}) = v_{entry} \qquad \psi(v'_{exit}) = v_{exit}$$

$$(3)\ \langle v', u' \rangle \in E' \to \langle \psi(v'), \psi(u') \rangle \in E$$

$$(4)\ \langle v', u' \rangle \in E' \wedge \langle v', w' \rangle \in E' \wedge u' \neq w' \ \to \psi(u') \neq \psi(w')$$

Графом развертки, например, является дерево выполнений программы, проходящих по обратным рёбрам не более k-раз. Так как размер такого дерева экспоненциально зависит от размера исходной программы, на практике используются ациклические графы. Пример алгоритма построения ациклического графа развертки приведен в работе [6].

Будем говорить, что путь $l = \langle v_0 = v_{entry}, v_1, v_2, \ldots, v_n \rangle$ от истока на ГПУ принадлежит к графу развертки G' (обозначим как $\psi(l) \in G'$), если найдется такой путь $l' = \langle v'_0 = v'_{entry}, v'_1, v'_2, \ldots, v'_n \rangle$ в G', что $\forall_{i \in [0,n]} \psi(v'_i) = v_i$. Заметим, что исходя из свойства (4), если путь l' существует, то он единственный.

Тогда символьное выполнение должно производить поиск ошибок среди путей $l \in G$, таких, что $\psi(l) \in G'$. Учитывая введённую параметризацию, ход выполнения в рассматриваемой модели полностью определяется значением символьных переменных.

3. Определения ошибочных ситуаций

Пусть Σ – множество значений вектора символьных переменных $\bar{\boldsymbol{s}}$. Тогда рассмотрим множество $\{\Sigma_i\} \subseteq 2^{\Sigma}$, где $\Sigma_i \subseteq \Sigma$. Элемент Σ_i назовём абстракцией, а $\{\Sigma_i\}$ – множеством абстракций. Будем говорить, что в программе в точке B содержится ошибка, если найдется такая абстракция Σ_i, что на всех наборах значений переменных $\sigma \in \Sigma_{\text{i}}$ произойдет ошибка в точке B.

Обозначим как $P_B^{\Sigma_i}(\bar{\boldsymbol{s}})$ формулу от символьных переменных, задающую условие того, что $\bar{\boldsymbol{s}}$ принадлежит абстракции Σ_i и управление дойдет до точки B. Как $Err_B(\bar{\boldsymbol{s}})$ обозначим условие того, что в точке B произойдет ошибка. Тогда дадим определение ошибки для абстракции Σ_i следующим образом следующим образом:

$$(5)\quad Err_B^{\Sigma_i} = \left(\exists \bar{\boldsymbol{s}} \left(P_B^{\Sigma_i}(\bar{\boldsymbol{s}})\right)\right) \wedge \forall \bar{\boldsymbol{s}} \left(P_B^{\Sigma_i}(\bar{\boldsymbol{s}}) \rightarrow Err_B(\bar{\boldsymbol{s}})\right)$$

Тогда наличие ошибки в точке B определим, как существования абстракции, на которой произойдёт ошибка:

$$(6)\ \exists \Sigma_i :\ Err_B^{\Sigma_\mathrm{i}}$$

Данную формулу будем называть общим определением ошибки, $\langle \{\Sigma_i\}, Err_B \rangle$ - определением ошибки, полученным подстановкой $\{\Sigma_i\}$ и Err_B в общее определение ошибки. В зависимости от выбора множества абстракций $\{\Sigma_i\}$ будут различаться множества обнаруживаемых ошибок. Рассмотрим два множества абстракций $\boldsymbol{\Sigma}' = \{\Sigma_i'\}$ и $\boldsymbol{\Sigma}'' = \{\Sigma_j''\}$, $\boldsymbol{\Sigma}', \boldsymbol{\Sigma}'' \in 2^{\Sigma}$ таких, что:

$$(7)\ \forall i\ \exists \bar{\boldsymbol{j}}: \Sigma_i' = \bigcup\nolimits_{j \in \bar{\boldsymbol{j}}} \Sigma_j''$$

Пусть $Err_B^{\boldsymbol{\Sigma}'}$- множество ошибок в точке B для множества абстракций $\boldsymbol{\Sigma}'$, $Err_B^{\boldsymbol{\Sigma}''}$ - для $\boldsymbol{\Sigma}''$, тогда если (7) верно, то:

$$(8)\ \forall B :\ Err_B^{\boldsymbol{\Sigma}'} \subseteq Err_B^{\boldsymbol{\Sigma}''}$$

Для доказательства (8) заметим, что если верна $Err_B^{\Sigma_\mathrm{i}'}$, то для всех $\Sigma_j'' \subseteq \Sigma_i'$, при условии достижимости $\mathrm{P}_\mathrm{B}^{\Sigma_j''}(\bar{\boldsymbol{s}})$, будет верна и $Err_B^{\Sigma_j''}$. Так как $P_B^{\Sigma_i'}(\bar{\boldsymbol{s}})$ – выполнима, а $\Sigma_i' = \bigcup_{j \in \bar{\jmath}} \Sigma_j''$, то среди Σ_j'' найдется такой $\Sigma_{j'}''$, что $P_B^{\Sigma_{j'}''}(\bar{\boldsymbol{s}})$ – выполнима. Тогда подставляя $i = j'$ в (6) получим верную формулу.

Таким образом, определение ошибки (6) позволяет проводить сравнение различных методов обнаружения ошибок. Рассмотрим несколько используемых на практике вариантов задания абстракций $\{\Sigma_i\}$.

Рассмотрим множество Σ, пронумеруем все его элементы $\Sigma = \{\sigma_i\}$. Зададим $\Sigma_i = \sigma_i$. Тогда для данного варианта разбиения формула (6) принимает вид:

$$(9)\ \exists \sigma_i: P_B(\sigma_i) \wedge Err_B(\sigma_i)$$

Формула (9) эквивалента формуле (1), т.к. задает значения символьных переменных параметризации. Таким образом формула (6) для разбиения $\Sigma_i = \sigma_i$ формулирует задачу символьного выполнения для автоматической генерации тестов.

С другой стороны, в качестве множества абстракции можно взять один элемент $\Sigma_1 = \Sigma$. Тогда формула (6) примет вид:

$$(10)\ \exists s\left(P_B(\bar{\boldsymbol{s}})\right) \wedge \forall s\ \left(P_B(\bar{\boldsymbol{s}}) \rightarrow Err_B(\bar{\boldsymbol{s}})\right)$$

Формула (10) утверждает, что если точка B такова, что она достижима хотя бы на одном конкретном состоянии и в ней всегда происходит ошибка, то точка B ошибочна. Данное определение является самым строгим из рассматриваемых, поэтому ему свойственен пропуск реальных ошибок.

По числу обнаруживаемых ошибок между абстракциями $\Sigma_i = \sigma_i$ и $\Sigma_i = P_B$ находится абстракция путей выполнения. Идея данной абстракции основана на предположении, что контракт функции не запрещает пути выполнения в графе развертки. Иными словами, для каждого пути в графе развертки найдется способ запустить рассматриваемую функцию таким образом, чтобы управление прошло именно на данном пути.

Для построения абстракций путей выполнения, каждому булевому выражению в графе развертки сопоставим свою булеву переменную $\{b_j\}$. По построению, каждая из булевых переменных зависит параметризации, т.е. $b_j = b_j(\bar{\boldsymbol{s}})$, тогда:

$$(11)\ \Sigma_i = \bigwedge_j \begin{cases} b_j, \text{если} \left\lfloor \frac{i}{2^j} \right\rfloor \equiv 0\ (mod\ 2) \\ \neg b_j,\ \text{иначе} \end{cases} = \bar{\boldsymbol{b}}_i$$

Таким образом, каждая Σ_i задает значения всех булевых выражений, которые в свою очередь однозначно определяет путь выполнения в графе развертки. Формула (6) для данной абстракции имеет следующий вид:

$$(12)\ \exists \bar{\boldsymbol{b}}: \left(\exists \bar{\boldsymbol{s}} \left(P_B^{\bar{\boldsymbol{b}}}(\bar{\boldsymbol{s}})\right)\right) \wedge \forall \bar{\boldsymbol{s}}\ \left(P_B^{\bar{\boldsymbol{b}}}(\bar{\boldsymbol{s}}) \rightarrow\ Err_B(\bar{\boldsymbol{s}})\right)$$

Где $P_B^{\bar{\boldsymbol{b}}}(\bar{\boldsymbol{s}}) = \left(\forall i\ b_i = b_i(\bar{\boldsymbol{s}})\right) \wedge P_B$

Заметим, что для введённых абстракций верно следующее соотношение:

$$(13)\ Err^{\Sigma_i = P_B} \subseteq Err^{\Sigma_\mathrm{i} = \bar{\boldsymbol{b}}_i} \subseteq Err^{\Sigma_i = \sigma_i}$$

Между абстракциями $\Sigma_i = P_B$ и $\Sigma_i = \bar{\boldsymbol{b}}_i$ находятся абстракции k критических точек. При использовании абстракции k критических точек точка B считается ошибочным в том случае, если найдутся k точек графа развертки таких, что любой путь проходящий через них и через B приводит к ошибке. Как можно заметить, при достаточно большом числе k, данная абстракция будет эквивалентна абстракции $\Sigma_i = \bar{\boldsymbol{b}}_i$. Рассмотрим случай $k = 1$. Пронумеруем все точки в графе развертки – $\{B_i\}$, тогда $\Sigma_i = P_{B_i}$, а $P_B^{B_i} = P_{B_i} \wedge P_B$. Тогда общая формула ошибки записывается следующим образом:

$$(14)\ \exists B_i: \left(\exists \bar{\boldsymbol{s}} \left(P_B^{B_i}(\bar{\boldsymbol{s}})\right)\right) \wedge \forall \bar{\boldsymbol{s}}\ \left(P_B^{B_i}(\bar{\boldsymbol{s}}) \rightarrow\ Err_B(\bar{\boldsymbol{s}})\right)$$

Учитывая последнюю введённую абстракцию, напишем итоговое соотношение:

$$(15)\ Err^{\Sigma_i = \Sigma} \subseteq Err^{\Sigma_i = P_{B_i}} \subseteq Err^{\Sigma_\mathrm{i} = \bar{\boldsymbol{b}}_i} \subseteq Err^{\Sigma_i = \sigma_i}$$

4. Примеры ошибочных ситуаций

Рассмотрим на примерах разницу между представленными выше определениями ошибок.

```
1) public string Example1(object obj)
2) {
3)     obj = null;
4)     return obj.ToString();
5) }
```

```
1) public string Example2(object obj, bool a)
2) {
3)     if (a) {
4)         obj = null;
5)     }
6)     return obj.ToString();
7) }
```

```
1) public string Example3(object obj, bool a, bool b)
2) {
3)     if (a) {
4)         obj = null;
5)     }
6)     if (b) {
7)          return obj.ToString();
8)     }
9)     return null;
10)}
```

```
1) public string Example4(object obj, bool a, bool b, bool c)
2) {
3)    if (a) {
4)        obj = null;
5)    }
6)    if (b) {
7)         obj = new object();
8)    }
9)    if (c) {
10)          return obj.ToString();
11)    }
12)   return null;
13)}
```

```
1) public string Example5(object obj)
2) {
3)    return obj.ToString();
4) }
```

Листинг 1. Примеры различных ошибочных ситуаций.

Listing 1. Examples of defects.

На листинге 1 приведены пять различных потенциально ошибочных ситуаций. В качестве условия ошибки рассмотрим условие $obj == null$. Тогда Example1 будет являться ошибочным для всех абстракций.

Example2 является ошибочным для всех абстракций, кроме $\Sigma_i = \Sigma$. Однако, если в качестве условия ошибки использовать не условие равенства $null$ в точке разыменования (6), а условие разыменования в точке присваивания (4), то определение ошибки при $\Sigma_i = \Sigma$ также обнаружит ошибку.

Example3 также является ошибочным для всех абстракций, кроме $\Sigma_i = \Sigma$, однако в отличие от Example2 определение $\Sigma_i = \Sigma$ не обнаружит ошибку в данном случае при переносе ошибочной точки в присваивание.

В Example4 ошибку обнаружат только абстракции $\Sigma_i = \overline{\boldsymbol{b}}_i$ и $\Sigma_i = \sigma_i$. Абстракция $\Sigma_i = P_{B\,i}$ не найдет ошибку, т.к. критической точкой для данной абстракции является точка (4), однако между точками (4) и (10) существует путь (4)-(7)-(10) не содержащий ошибок. Следовательно, по определению (14) данная ситуация не является ошибочной.

Наконец, в Example5 ошибку обнаружит только абстракция $\Sigma_i = \sigma_i$.

5. Примеры алгоритмов поиска ошибочных ситуаций

Рассмотрим классификацию алгоритмов поиска ошибок относительно введённых определений. Рассмотрим некоторый алгоритм поиска ошибок A, обозначим за Err_A множество обнаруживаемых им ошибок для данного графа развертки. Тогда для упорядоченного набора абстракций

$$(16)\ Err^{\Sigma_i^0} = \emptyset \subseteq Err^{\Sigma_i^1} \subseteq Err^{\Sigma_i^2} \subseteq \cdots Err^{\Sigma_i^N} \subseteq Err^{\Sigma_i = \sigma_i}$$

будем говорить, что A относится к классу $Err^{\Sigma_\text{i}^k}$, если $Err_A \subseteq Err^{\Sigma_\text{i}^k}$, но $Err_A \nsubseteq Err^{\Sigma_\text{i}^{k-1}}$. Заметим, что в данном случае допускается пропуск ошибок класса $Err^{\Sigma_\text{i}^k}$, однако алгоритм A должен находить некоторый набор ошибок, таких что они принадлежат $Err^{\Sigma_\text{i}^k}$, но не содержатся в $Err^{\Sigma_\text{i}^{k-1}}$.

Заметим, что набор ошибок, обнаруживаемых алгоритмом A, может не совпадать с набором выдаваемых им предупреждений. Для предупреждения w_A будем говорить, что оно является ошибкой, если $w_A \in Err^{\Sigma_i = \sigma_i}$.

Рассмотрим построенные во второй главе абстракции:

$$(17)\ Err^{\Sigma_i^0} = \emptyset \subseteq Err^{\Sigma_i=\Sigma} \subseteq Err^{\Sigma_i=P_{B_i}} \subseteq Err^{\Sigma_\mathrm{i}=\overline{\boldsymbol{b}}_i} \subseteq Err^{\Sigma_i=\sigma_i}$$

Примером алгоритма поиска ошибок, относящихся к классу $\Sigma_i = \Sigma$ является анализ критических рёбер, реализованный в анализаторе Svace [7]. В качестве условия возникновения ошибки для поиска разыменования нулевого указателя в программах на C/C++ используется следующее условие:

$$(18) Err_x = (x = null) \wedge (dereference(x))$$

Где $dereference(x)$ – условие того, что символьное значение x будет использовано в операции разыменование далее в графе развертки. Условие $dereference(x)$ может быть вычислено с помощью обратного анализа графа развертки.

Алгоритм, использующийся в анализаторе Svace для поиска разыменования нулевого указателя, основан на вычислении для каждой точки условий $null(x) = \{T, F\},\ deref(x) = \{T, F\}$. Условие $null(x)$ истинно в том случае, если в данной точке символьная переменная x всегда равна $null$. Соответственно, $deref(x)$ истинен в том случае, если символьная переменная x будет гарантированно разыменована. Тогда, если найдется такая точка B в графе развертки G', что для какой-либо переменной x одновременно верны $null(x) \wedge deref(x)$, то в точке B происходит ошибка.

Покажем, что данный алгоритм относится к классу $\Sigma_i = \Sigma$. Для этого достаточно показать, что если данный алгоритм находит ошибку, то она соответствует определению $\langle \Sigma_i = \Sigma, Err_x \rangle$. Действительно, если для точки B верно одновременно $null(x) \wedge deref(x)$, то на любом пути выполнения будет верно $Err_x = (x = null) \wedge (dereference(x))$. Однако определение $\langle \Sigma_i = \Sigma, Err_x \rangle$ предполагает, что точка B достижима из точки входа. Данный алгоритм не проводит анализ достижимости, поэтому он будет выдавать предупреждения даже в случае недостижимого кода. Предупреждения в недостижимом коде не содержатся в $Err^{\Sigma_i=\sigma_i}$, поэтому они не включаются в общее множество ошибок алгоритма, следовательно ошибки данного алгоритма включены в $\langle \Sigma_i = \Sigma, Err_x \rangle$.

Рассмотрим алгоритм поиска ошибки разыменования нулевого указателя по определению $\langle \Sigma_\mathrm{i} = \overline{\boldsymbol{b}}_i, Err_{Deref(x)} = (x = null) \rangle$. Пусть для каждой вершины графа развертки посчитаны условия $Null_B^x(\overline{\boldsymbol{b}})$, такие, что $Null_B^x(\overline{\boldsymbol{b}}) \to x = null$. Запись $Null_B^x(\overline{\boldsymbol{b}})$ означает, что условие ошибки зависит от значений булевых выражений

Алгоритм выдает в точке B ошибку в том случае, если формула разрешима:

$$(19)\ Null_B^x(\overline{\boldsymbol{b}}_i(\overline{\boldsymbol{s}})) \wedge P_B(\overline{\boldsymbol{s}}).$$

Покажем, что все ошибки найденные данным алгоритмом являются ошибками по определению определения $\langle \Sigma_\mathrm{i} = \overline{\boldsymbol{b}}_i, Err_{Deref(x)} = (x = null) \rangle$.

Так как формула $Null_B^x(\overline{\boldsymbol{b}}(\overline{\boldsymbol{s}}\,)) \wedge P_B(\overline{\boldsymbol{s}}\,)$ разрешима, то найдется такое $\overline{\boldsymbol{s}}'$, что $Null_B^x(\overline{\boldsymbol{b}}(\overline{\boldsymbol{s}}'\,)) \wedge P_B(\overline{\boldsymbol{s}}'\,)$ - верно. Пусть $\overline{\boldsymbol{b}}' = \overline{\boldsymbol{b}}(\overline{\boldsymbol{s}}')$, тогда подставив в формулу (12) $\overline{\boldsymbol{b}} = \overline{\boldsymbol{b}}'$ получим:

$$(20) \left(\exists \overline{\boldsymbol{s}} \left(\left(\forall i\ b_i' = b_i'(\overline{\boldsymbol{s}})\right) \wedge P_B(\overline{\boldsymbol{s}}\,)\right)\right) \wedge \forall \overline{\boldsymbol{s}} \left(\left(\forall i\ b_i' = b_i'(\overline{\boldsymbol{s}})\right) \wedge P_B(\overline{\boldsymbol{s}}\,) \rightarrow x = null\right)$$

Заметим, что первый конъюнкт формулы (20) верен, т.к. взяв $\overline{\boldsymbol{s}} = \overline{\boldsymbol{s}}'$ получим верное равенство. Осталось доказать, что:

$$(21)\ \forall \overline{\boldsymbol{s}} \left(\left(\forall i\ b_i' = b_i(\overline{\boldsymbol{s}})\right) \wedge P_B(\overline{\boldsymbol{s}}\,) \rightarrow x = null\right)$$

Для этого докажем следующее утверждение:

$$(22) \left(\forall i\ b_i' = b_i(\overline{\boldsymbol{s}})\right) \wedge P_B(\overline{\boldsymbol{s}}\,) \rightarrow\ Null_B^x(\overline{\boldsymbol{b}}) \wedge P_B(\overline{\boldsymbol{s}}\,)$$

Так как $\overline{\boldsymbol{s}}'$ является решением для (20), а $\overline{\boldsymbol{b}}' = \overline{\boldsymbol{b}}(\overline{\boldsymbol{s}}')$, то $Null_B^x(\overline{\boldsymbol{b}}')$ - верно. Условие $\left(\forall i\ b_i' = b_i(\overline{\boldsymbol{s}})\right)$ означает, что $\overline{\boldsymbol{b}}' = \overline{\boldsymbol{b}}(\overline{\boldsymbol{s}}')$, отсюда получаем, что $\left(\forall i\ b_i' = b_i(\overline{\boldsymbol{s}})\right) \rightarrow\ Null_B^x(\overline{\boldsymbol{b}})$. Следовательно, исходная импликация (22) верна. Учитывая, что из $Null_B^x(\overline{\boldsymbol{b}}) \wedge P_B(\overline{\boldsymbol{s}}\,) \rightarrow x = null$, получаем, что формула (21) также верна. Таким образом доказано, что результаты данного алгоритма включены в $\langle \Sigma_\mathrm{i} = \overline{\boldsymbol{b}}_i, Err_{Deref(\,x)} = (x = null)\rangle$.

Количество срабатываний, выдаваемых алгоритмом напрямую зависит от вида условия $Null_B^x$. Вообще говоря, условие $Null_B^x$ может быть построено точно, т.е. результаты такого алгоритма совпадут с $\langle \Sigma_\mathrm{i} = \overline{\boldsymbol{b}}_i, Err_{Deref(\,x)x} = (x = null)\rangle$. На практике, в инструментах Svace [7] и SharpChecher [6], используются неполные реализации, пропускающие часть срабатываний по сравнению $\langle \Sigma_\mathrm{i} = \overline{\boldsymbol{b}}_i, Err_{Deref(\,x)x} = (x = null)\rangle$. Однако, в соответствии с классификацией данные реализации всё равно относятся к $\langle \Sigma_\mathrm{i} = \overline{\boldsymbol{b}}_i, Err_{Deref(\,x)x} = (x = null)\rangle$, т.к. обнаруживают ошибки в Example4.

В качестве примера алгоритма поиска $Err^{\Sigma_i = P_{B}{}_i}$, может быть использован алгоритм поиска $Err^{\Sigma_\mathrm{i} = \overline{\boldsymbol{b}}_i}$, который в случае обнаружения ошибки дополнительно проверяет её принадлежность к классу $Err^{\Sigma_i = P_{B}{}_i}$.

5. Применения предложенной классификации

Предложенный метод построения определения ошибок может быть использован для введения определений с целью формализации новых подклассов ошибочных ситуаций. Примером такого класса ошибок являются ошибки, происходящие вне зависимости от условия заданного перехода. Мотивацией к введению такого класса ошибок является наличие ситуаций, в которых переход зависит от возвращаемого значения неизвестной функции.

Если ошибка возможно только на одной ветке такого перехода, то ошибку предлагается не выдавать.

Набор различных определений ошибочных ситуаций можно использовать для анализа и сравнения алгоритмов поиска ошибок. Наличие формальных определений ошибочных ситуаций позволяет проводить формальное доказательство соответствия предложенных алгоритмов анализа программ.

Кроме того, предложенные определения ошибок могут сами использованы как алгоритмы поиска ошибок. Построенные формулы, описывающие ошибочные ситуации, могут быть переданы сторонним SMT-решателям, которые проверят их на совместность. В случае, если формула является совместной, SMT-решатель предоставит явное решение, описывающее абстракцию Σ_i, на которой происходит ошибка. Данная абстракция может быть предъявлена пользователю, в качестве примера ошибочной ситуации.

Использование определений в качестве алгоритма поиска ошибок было применено в анализаторе Svace для поиска переполнения буфера. Предварительное тестирование данного подхода показало его применимость для обнаружения реальных ошибок в промышленных программах.

Наконец, использование разных определений ошибочных ситуаций может применяться для ранжирования предупреждений. Благодаря тому, что ошибочные ситуации вложены друг в друга, результаты работы алгоритма могут быть классифицированы по определениям ошибок, как наименьшее по включению определение, содержащее данную ошибку. Данная классификация является осмысленной, поскольку, как правило, меньшие по включению определения имеют более высокий процент истинных срабатываний.

6. Заключение

В данной работе рассмотрен вопрос формализации ошибочных ситуаций для статического символьного выполнения. Рассмотрена общая формула ошибочных ситуаций. Разобраны частные определения ошибочных ситуаций, а также алгоритмы их поиска, использующиеся в анализаторах кода Svace и SharpChecker. Разработана классификация алгоритмов поиска ошибок на основе набора определений ошибочных ситуаций. Для рассматриваемых алгоритмов поиска ошибок проведена их классификация, включая доказательство соответствия. Приведены примеры использования данной формализации на практике.

Список литературы

[1]. Y. Xie, A. Aiken. Saturn: A Scalable Framework for Error Detection Using Boolean Satisfiability ACM Trans. Program. Lang. Syst. 2007. Vol. 29, no. 3.

[2]. F. Ivančić, G. Balakrishnan, A. Gupta at al. Scalable and scope-bounded software verification in Varvel. Automated Software Engineering. 2015. Vol. 22, no. 4. pp. 517–559.

[3]. Babic D., Hu A.J. Calysto. Software Engineering, 2008. ICSE ’08. ACM/IEEE 30th International Conference on. 2008. May. pp. 211–220.
[4]. В.К. Кошелев, И.А. Дудина, В.И. Игнатьев, А.И. Борзилов. Чувствительный к путям поиск дефектов в программах на языке C# на примере разыменования нулевого указателя. Труды ИСП РАН, том 27, вып. 5, 2015 г., стр. 59-86. DOI: 10.15514/ISPRAS-2015-27(5)-5
[5]. J. King. Symbolic Execution and Program Testing. Commun. ACM. 1976. Vol. 19, no. 7. pp. 385–394.
[6]. В. К. Кошелев, В. Н. Игнатьев, А. И. Борзилов. Инфраструктура статического анализа программ на языке C#. Труды ИСП РАН, том 28, вып. 1, 2016 г., стр. 21-40. DOI: 10.15514/ISPRAS-2016-28(1)-2
[7]. В.П. Иванников А.А. Белеванцев А.Е. Бородин В.Н. Игнатьев Д.М. Журихин А.И. Аветисян М.И. Леонов. Статический анализатор Svace для поиска дефектов в исходном коде программ. Труды ИСП РАН, том. 26, вып. 1. pp. 231–250. DOI: 10.15514/ISPRAS-2014-26(1)-7
[8]. И.А. Дудина, В.К. Кошелев, А.Е. Бородин. Поиск ошибок доступа к буферу в программах на языке C/C++. Труды ИСП РАН, том 28, вып. 4, 2016 г., стр. 149-168. DOI: 10.15514/ISPRAS-2016-28(4)-9

Formalization of Error Criteria for static symbolic execution

V.K. Koshelev <vedun@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

Abstract. This paper is devoted to the formalization of the error criteria for program static analysis, based on symbolic execution. Using the original error criteria of symbolic execution approach in program static analysis leads to an excessive number of false positives. To solve this problem, we propose an alternative definition of the error criteria. Proposed definition reports errors only if they occur on a certain set of input variables. Examples of such sets are the set of values of input variables in which control will pass through a given point of the program, or set of values in which the controls take place along a given path in the control flow graph. This paper discusses the various ways to specify such sets of initial values, including analysis of the final error criteria. We overview algorithms corresponding to the error criteria and prove their correctness. Finally, we consider the practical applications of the given error criteria, which include classification of the warnings generated by static analysis tools; taking into account unknown function contracting, especially preconditions; using the proposed error criteria as formulas for a SMT-solver. The latest application allows to get the precise solution of the particular error criteria, including the error trace.

Keywords: static analysis, error criteria, symbolic execution.

DOI: 10.15514/ISPRAS-2016-28(5)-6

For citation: V.K. Koshelev. Formalization of Error Criteria for static symbolic execution. Trudy ISP RAN/Proc. ISP RAS, 2016, vol. 28, issue 5, 2016, pp. 105-118 (in Russian). DOI: 10.15514/ISPRAS-2016-28(5)-6

References

[1]. Y. Xie, A. Aiken. Saturn: A Scalable Framework for Error Detection Using Boolean Satisfiability ACM Trans. Program. Lang. Syst. 2007. Vol. 29, no. 3.
[2]. F. Ivančić, G. Balakrishnan, A. Gupta at al. Scalable and scope-bounded software verification in Varvel. Automated Software Engineering. 2015. Vol. 22, no. 4. pp. 517–559.
[3]. Babic D., Hu A.J. Calysto. Software Engineering, 2008. ICSE ’08. ACM/IEEE 30th International Conference on. 2008. May. pp. 211–220.

[4]. V. Koshelev, I. Dudina, V. Ignatyev, A. Borzilov. [Path-Sensitive Bug Detection Analysis of C# Program Illustrated by Null Pointer Dereference]. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 5, 2015. pp. 59-86 (in Russian). DOI: 10.15514/ISPRAS-2015-27(5)-5

[5]. J. King. Symbolic Execution and Program Testing. Commun. ACM. 1976. Vol. 19, no. 7. pp. 385–394

[6]. V. Koshelev, V. Ignatyev, A. Borzilov. C# static analysis framework. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 1, 2016, pp. 21-40 (in Russian). DOI: 10.15514/ISPRAS-2016-28(1)-2

[7]. V.P. Ivannikov, A.A. Belevantsev, A.E. Borodin, V.N. Ignatiev, D.M. Zhurikhin, A.I. Avetisyan, M.I. Leonov. Static analyzer Svace for finding of defects in program source code. Trudy ISP RAN/Proc. ISP RAS, vol. 26, issue 1, 2014, pp. 231-250 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-7

[8]. I. Dudina, V. Koshelev, A. Borodin. [Statically detecting buffer overflows in C/C++ Proceedings of the Institute for System Programming]. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 4, 2016, pp. 149-168 (in Russian).

Обнаружение ошибок доступа к буферу в программах на языке C/C++ с помощью статического анализа

И.А. Дудина <eupharina@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25.
Московский государственный университет имени М.В. Ломоносова,
119991, Россия, Москва, Ленинские горы, д. 1

Аннотация. В данной работе рассматривается метод поиска межпроцедурных ошибок доступа к буферу с помощью статического анализа. В основе рассматриваемого подхода лежит разработанный ранее алгоритм внутрипроцедурного анализа на базе символьного исполнения с объединением состояний, который является чувствительным к путям и учитывает взаимосвязи между переменными, такие как сравнения, арифметические операции и инструкции приведения типа. В работе предложено формальное определение межпроцедурного дефекта и рассмотрены некоторые типы межпроцедурных ошибок доступа к буферу. Межпроцедурный анализ реализован с помощью метода резюме, что позволяет в некоторой степени добиться контекстной чувствительности. Показано, как можно расширить внутрипроцедурный алгоритм для отслеживания межпроцедурных связей между переменными. Кроме этого, приведен алгоритм построения двух типов достаточных условий наличия ошибки доступа к буферу в функции, которые сохраняются в резюме и проверяются при вызове этой функции. Описанный подход был реализован в инструменте статического анализа Svace. На проекте Android 5.0.2 было получено 351 предупреждение об ошибке доступа к буферу, среди которых 64% оказались истинными, при этом существенного замедления анализа не произошло.

Ключевые слова: статический анализ; поиск дефектов; переполнение буфера; чувствительность к путям; контекстная чувствительность; межпроцедурный анализ; символьное исполнение.

DOI: 10.15514/ISPRAS-2016-28(5)-7

Для цитирования: И.А. Дудина. Обнаружение ошибок доступа к буферу в программах на языке C/C++ с помощью статического анализа. Труды ИСП РАН, том 28, вып. 5, 2016, стр. 119-134. DOI: 10.15514/ISPRAS-2016-28(5)-7

1. Введение

Поиск ошибок переполнения буфера в исходном коде программ остаётся актуальной задачей на протяжении уже нескольких десятилетий. Одним из традиционных подходов к решению этой задачи является статический (не предполагающий запуск программы) анализ исходного кода, к преимуществам которого можно отнести покрытие всех путей при анализе, отсутствие необходимости генерировать входные данные для анализируемой программы.

```
1  #define SIZE 10
2  int checkIdx(int, int);
3  int findIdx(int val) {
4    int x;
5    for (x=0; x<SIZE; x++)
6      if(checkIdx(x, val))
7        break;
8    return x;
9  }
10 void store(int *b,
11          int i, int val) {
12   b[i]=val;
13 }
14 void foo(int val) {
15   int buffer[SIZE];
16   int idx=findIdx(val);
17   store(buffer, idx, val);
18 }
```

Рис. 1. Пример межпроцедурного срабатывания

Fig. 1. An example of inter-procedural warning

Современные методы анализа позволяют находить всё более сложные типы ошибок, сохраняя при этом приемлемое время анализа и умеренное количество ложных срабатываний. В частности, анализ совместности условий переходов позволяет организовать чувствительный к путям поиск ошибок, т.е. находить такие последовательности из более чем одной точки программы, прохождение пути выполнения по которым обязательно приведет к возникновению ошибки. Построение детектора ошибок переполнения буфера такого типа в рамках одной функции подробно описано в статье [1].

Также в отдельных класс ошибок, обнаружение которых требует привлечения специализированных подходов, можно выделить межпроцедурные ошибки. Под этим термином мы будем понимать ситуацию, когда корректность работы некоторой функции зависит от выполнения некоторого условия над значениями, получаемыми из других функций (значения, возвращаемые или изменяемые вызываемыми функциями, либо передаваемые в качестве параметров из вызывающей функции), и это условие нарушается на некотором исполнении программы, что приводит к ошибке. Такое условие корректности выполнения функции далее будем называть контрактом.

В качестве иллюстрации межпроцедурной ошибки рассмотрим пример дефекта, аналогичный обнаруженному в реальном проекте (см. Рис. 1). В данном случае функция `findIdx` вычисляет некоторое значение, которое в том числе может быть равно `SIZE`. Функция `foo` передаёт результат вызова

`findIdx` и адрес своего локального буфера размера `SIZE` в функцию `store`, в которой к переданному буферу происходит обращение по переданному индексу. Сама по себе функция `store` не содержит ошибки, но её контракт подразумевает, что переданный индекс меньше размера переданного буфера. Как мы видим, в точке вызова в функции `foo` этот контракт может нарушаться, следовательно, в этом месте необходимо выдать предупреждение.

Данный пример иллюстрирует, что выделение буфера, вычисление индекса и инструкция доступа к буферу могут находиться в разных функциях, при этом можно говорить о наличии ошибки в функции, являющейся их ближайшим общим предком в ациклическом графе вызовов. Для обнаружения таких дефектов необходимо вычислять контракты функций, гарантирующие отсутствие ошибок доступа к буферу, анализировать результат и побочные эффекты функции.

2. Постановка задачи

Целью данной работы является расширение разработанного ранее и описанного в статье [1] алгоритма поиска ошибок доступа к буферу для организации поиска межпроцедурных ошибок. Новый алгоритм предполагает те же ограничения и предположения об анализируемой программе, а именно:

- рассматриваются только обращения к буферам, имеющим константный (т.е. известный в момент компиляции) размер и размещённым в статической памяти либо на стеке;
- выполнено *предположение о контрактах*: «Контракт произвольной функции не влияет на выполнимость любого из путей на графе потока управления (ГПУ) этой функции (не существует выполнимого, но запрещенного контрактом пути)»;
- при проведении анализа каждая функция считается точкой входа в программу, что позволяет обнаруживать дефекты, проявляющиеся в потенциальных (возможных, но отсутствующих в доступном анализатору коде) контекстах вызова.

Пусть G – подграф межпроцедурного потока управления программы, содержащий только анализируемую функцию и всех её потомков в графе вызовов вплоть до листьев. Пусть G_k – граф G после развёртки каждого его цикла на k итераций [2]. Анализатор должен выдавать предупреждение об ошибке доступа к буферу, если в графе G_k существует путь, удовлетворяющий следующим условиям:

1. он содержит инструкцию обращения к буферу размера S по индексу i;
2. на любом соответствующем конкретном пути значение переменной i перед этой инструкцией не принадлежит интервалу $[0, \mathrm{S}-1]$;
3. данный путь выполним.

В работе [1] было показано, что если контракт некоторой функция удовлетворяет предположению о контрактах то, с одной стороны, если эта функция удовлетворяет данному условию, то существует подходящий под контракт потенциальный контекст вызова этой функции, в котором её выполнение приведет к ошибке доступа к буферу; и наоборот – если существует такой контекст вызова, при котором выполнение пройдет не более k^n раз по каждому обратному ребру цикла вложенности n и приведет к ошибке доступа к буферу, то такая функция будет удовлетворять условию.

2.1 Определение межпроцедурной ошибки

Рассмотрим некоторый путь P в графе G_k, удовлетворяющий приведённому определению, т.е. содержащий ошибку доступа к буферу. Он представляет собой некоторую конечную последовательность рёбер $P = \{e_i\}$, включающую ребро, ведущее в инструкцию p_{access} доступа к буферу, в которой происходит ошибка. Выберем из $\{e_i\}$ произвольную подпоследовательность $\{e_{i_s}\}$ и на графе вызовов программы пометим все функции, содержащие рёбра из последовательности $\{e_{i_s}\}$. Далее рекурсивно отметим все функции, вызывающие отмеченные, вплоть до единого общего предка. В результате получится некоторый помеченный подграф графа вызовов, также являющийся деревом. Будем считать, что значения, изменяемые и возвращаемые непомеченными вызываемыми функциями, могут быть любыми, но обязаны удовлетворять предположению о контрактах; контекст вызова функции-корня помеченного поддерева может быть любым, удовлетворяющим предположению о контрактах. Если при данных условиях любой путь, проходящий через рёбра $\{e_{i_s}\}$, либо невыполним, либо ошибочен по данному выше определению с ошибкой доступа к буферу в точке p_{access}, то такой набор рёбер $\{e_{i_s}\}$ будем называть *критическим*. Если из некоторого критического набора нельзя выкинуть ни одного ребра с сохранением данного свойства, то такой набор будем называть *минимальным критическим*. Заметим, что для некоторых ошибочных путей можно построить более одного минимального критического набора. Если для некоторого пути, удовлетворяющего определению ошибки, не существует минимального критического набора, целиком состоящего из рёбер единственной функции, то такую ситуацию мы будем называть *межпроцедурной ошибкой*.

Заметим, что пример, изображенный на Рис. 1 является примером межпроцедурной ошибки, т.к. в любой критический набор ошибочного пути обязательно входит ребро, ведущее к инструкции доступа к буферу в функции `store`, но одного этого ребра недостаточно для определения ошибочного пути, т.к. для этой функции можно подобрать безопасный контекст вызова. Таким образом, в критический набор обязательно входят точки из других функций, т.е. ошибка межпроцедурная. В данном случае минимальный критический набор состоит из ребра в функции `store` и ребра,

соответствующего false-ветке условного оператора функции `findIdx` на k-ой итерации цикла.

3. Поиск межпроцедурных срабатываний

3.1 Описание внутрипроцедурного алгоритма

Внутрипроцедурный алгоритм поиска ошибок переполнения буфера реализован в виде модуля-детектора в статическом анализатора Svace и использует предоставляемую им инфраструктуру. Ядро Svace производит нумерацию значений, т.е. вычисляет классы эквивалентности значений переменных, называемые идентификаторами значений (VId) [7]. Детекторы ассоциируют с идентификаторами значений вычисленные свойства программы в виде атрибутов.

Ядро проводит символьное исполнение программы с объединением состояний [5]. При этом вычисляются необходимые условия достижимости каждой точки программы $q \in Instr$ в виде формул алгебры логики $ReachCond(q) = c,\ c \in Cond$, где роль переменных играют идентификаторы значений. Детекторы оповещаются о всех событиях, происходящих внутри функции. Реализация детектора заключается в описании обработчиков для этих событий.

Для организации поиска внутрипроцедурных срабатываний был введён атрибут $ValueSummary$, представляющий собой отображение:

$$VS\colon Instr \times VId \longrightarrow Summary.$$

Это означает, что в произвольной точке программы q для некоторых идентификаторов значений $v \in VId$ определено значение $s \in Summary$, суммирующее необходимую детектором информацию о значениях v по всем путям, заканчивающихся в q (подробно элементы множества $Summary$ и построение атрибута VS в ходе символьного выполнения с объединением состояний рассмотрено в статье [1]). Для каждого $s \in Summary$ определены функции:

$$HB, LB\colon Summary \times VId \ \rightarrow Cond.$$

Для любых $x \in VId, q \in Instr$, если $VS(q,v) = s$, то $HB(s,x)$ является достаточным условием того, что существует путь на ГПУ, заканчивающийся в q, такой что для каждого соответствующего конкретного пути выполнено $v \geq x$ (соответственно $v \leq x$ для формулы $LB(s,x)$). С помощью этих формул для произвольных идентификаторов значения $v, x \in VId$ в произвольной точке программы $q \in Instr$ можно вычислить условия $NotLess$ и $NotGreater$:

$$NotLess, NotGreater\colon Instr \times VId \times VId \rightarrow Cond,$$
$$NotLess(q,v,x) = HB(VS(q,v),x),$$
$$NotGreater(q,v,x) = LB(VS(q,v),x).$$

Формула $NotLess(q, v, x)$ представляет собой достаточное условие того, что, если управление пришло в точку q по некоторому пути графа потока управления, что для него в точке q всегда выполнено $v \geq x$.

Было показано, что для инструкции $ac \in Instr$ доступа к буферу с известным размером $s \in VId$ по индексу $i \in VId$ достаточным условием наличия ошибки в точке ac будет являться выполнимость формулы

$$ReachCond(ac) \wedge \big(NotLess(ac, i, s) \vee NotGreater(ac, i, -1)\big). \qquad (1)$$

С помощью описанных выше построений поиск внутрипроцедурных ошибок доступа к буферу осуществляется в три этапа:

1. В ходе символьного исполнения для идентификаторов значений $v \in VId$ в каждой точке программы $q \in Instr$ строится частичное отображение

$$VS\colon Instr \times VId \longrightarrow Summary.$$

2. При обработке инструкции ac доступа к буферу `b` по индексу `i` на основе значения $VS(ac, i)$ составляется формула (1) и проверяется на выполнимость.
3. В случае, если формула выполнима, т.е. подобраны значения переменных, приводящие к переполнению, из $VS(ac, i)$ путём подстановки конкретных значений переменных извлекается конкретный путь, приводящий к ошибке, и выдается предупреждение, указывающее на этот путь.

Значения атрибута VS принадлежат одному из пяти классов:

$$Summary = Const \cup Assume \cup Arithm \cup Cast \cup Join,$$

и сопоставляются идентификаторам значений при обработке соответствующих событий (см. Рис. 2): объявление константы ($newConst$), сравнение с идентификатором, имеющим непустое значение атрибута ($assume$); арифметические операции ($binaryOp$) инструкции приведения типов ($castZext$, $castTrunc$), слияния значений по путям с условиями ($join$), если идентификаторы значений их аргументов имеют непустые значения атрибута VS.

Каждое из значений множества $Summary$ представляет собой ациклический ориентированный граф, все узлы которого также являются элементами множества $Summary$, а листья являются значениями типа $Const$. Так, для переменной **res** значение $VS(p_7, res) = s_4$ изображено на **Error! Reference source not found.**.

$$\frac{q = newConst(v, n),\ s_v = \langle v, n\rangle \in Const}{VS \uplus \{\langle q, v\rangle \rightarrow\ s_v\}}$$

$$\frac{q = binaryOp(r, a, b, \diamond),\ VS(q, a) = s_a,\ VS(q, b) = s_b,\quad s_r = \langle r, s_a, s_b, \diamond\rangle \in Arithm}{VS \uplus \{\langle q, r\rangle\ \rightarrow\ s_r\}}$$

$$\frac{q = assume(v, cmp, \square),\ VS(q, cmp) = s_{cmp},\ \ s_v = \langle v, s_{cmp}, \square\rangle \in Relation}{VS \uplus \{\langle q, v\rangle\ \rightarrow\ s_v\}}$$

$$\frac{q = castZext(v, op),\ VS(q, op) = s_{op},\ s_v = \langle v, s_{op}\rangle \in ZExt}{VS \uplus \{\langle q, v\rangle \rightarrow s_v}$$

$$\frac{q = castTrunc(v, op, w),\ VS(q, v) = s_{op},\ s_v = \langle v, s_{op}, b\rangle \in Trunc}{VS \uplus \{\langle q, v\rangle \rightarrow\ s_v}$$

$$\frac{q = join(jId, l, c_l, r, c_r),\ VS(q, l) = \emptyset,\ VS(q, r) = s_r,\quad s_{jId} = \langle jId, \langle s_r, c_r\rangle\rangle \in Join}{VS \uplus \left\{\langle q, jId\rangle\ \rightarrow\ s_{jId}\right\}}$$

$$\frac{q = join(jId, l, c_l, r, c_r),\ VS(q, l) = s_l,\ VS(q, r) = \emptyset,\quad s_{jId} = \langle jId, \langle s_l, c_l\rangle\rangle \in Join}{VS \uplus \left\{\langle q, jId\rangle\ \rightarrow\ s_{jId}\right\}}$$

$$\frac{q = join(jId, l, c_l, r, c_r),\ VS(q, l) = s_l,\ VS(q, r) = s_r,\quad s_{jId} = \langle jId, \langle s_l, c_l\rangle, \langle s_r, c_r\rangle\rangle \in Join}{VS \uplus \left\{\langle q, jId\rangle\ \rightarrow\ s_{jId}\right\}}$$

Рис. 2. Правила вывода

Fig. 2. Inference rules

3.2 Поиск межпроцедурных срабатываний с помощью резюме

Межпроцедурный анализ в инструменте Svace реализуется с помощью резюме. Данный подход заключается в том, что все функции анализируются единожды в порядке от листьев к корню в графе вызовов программы, приведенном к ациклическому виду разрывом некоторых рёбер. На основе результата внутрипроцедурного анализа функции формируется и сохраняется её резюме, т.е. краткое описание эффекта от её исполнения, включающее значения некоторых атрибутов для выбранных идентификаторов значений (стратегия формирования резюме для атрибута определяется соответствующим детектором). При обработке инструкции вызова известной

функции происходит применение её резюме, которое обязательно уже сформировано в силу порядка обхода функций. При этом идентификаторам значений вызываемой функции сопоставляются соответствующие идентификаторы значений в контексте вызывающей функции. Значения атрибутов последних вычисляются на основе значений соответствующих идентификаторов в резюме (например, просто копируются из резюме). К преимуществам данного подхода можно отнести однократный анализ каждой функции, естественную контекстную чувствительность.

```
1  int plusOne(int x){
2     if (x1 >= 10){
3         x2 = 10;
4     }
5     x3 = phi(x1, x2);
6     int res = x3 + 1;
7     return res;
8  }
9  int buf11[11];
```

```
10 int innerAccess1(int a){
11    int idx = plusOne(a);
12    return buf11[idx];
13 }
14
15 int buf5[5];
16 int innerAccess2(){
17    int idx = plusOne(4);
18    return buf5[idx];
19 }
```

Рис. 3. Пример ошибки с межпроцедурным вычислением индекса

Fig. 3. An example of defect with inter-procedural index calculation

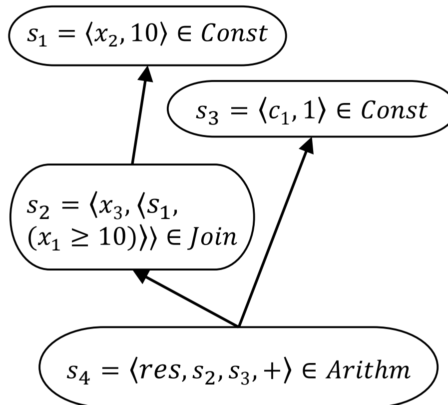


Рис. 4. Значение атрибута Summary для `res`
Fig. 5. Value of Summary attribute for `res`

3.3 Ошибки с межпроцедурным вычислением индекса

Рассмотрим произвольную ошибку доступа к буферу, возникающую в инструкции p_{access}. В данной инструкции используются две переменные – адрес буфера и индекс, значение каждой из которых может определяться в рамках текущей функции, либо вычисляться с помощью значений, вычисляемых в вызываемых или вызывающей функциях. В рамках текущей работы для переменной, содержащей адрес буфера, будем рассматривать два варианта: адрес известен в самой функции (явно используется определенный в данной функции или глобальный массив), адрес передан в качестве параметра-указателя. Для начала рассмотрим первый случай, проиллюстрированный на рис. 3. Функция `innerAccess1` содержит межпроцедурную ошибку доступа к буферу, т.к. можно привести ошибочный путь (11)-**(2)-(3)-(5)-(6)-(7)**-(12). Точку доступа к буферу на строке 12 обозначим p_{12}. Обнаружить такую ошибку можно, используя резюме. Из формулы (1), а также из того что $ReachCond(p_{12}) = true$ следует, что достаточным условием ошибки будет являться формула $NotLess(p_{12}, idx, 11)$.

Индексом, по которому происходит доступ на строке 12 является возвращаемое значение функции `plusOne`. Таким образом, $NotLess(p_{12}, idx, 11) = NotLess(p_7, res, 11)$. В ходе анализа функции `plusOne` было установлено, что $VS(p_7, res) = s_4$. Исходя из этого можно построить достаточное условие $NotLess(p_7, res, 11)$:

$$NotLess(p_7, res, 11) = HB(s_4, 11)$$
$$= (res = x_3 + c_1) \wedge \exists\widetilde{x_3}\exists\widetilde{c_1}\left(HB(s_2, \widetilde{x_3}) \wedge HB(s_3, \widetilde{c_1}) \wedge (\widetilde{x_3} + \widetilde{c_1} \geq 11)\right)$$
$$= (res = x_3 + c_1)$$
$$\wedge \exists\widetilde{x_3}\exists\widetilde{c_1}\,((x_3 = x_2) \wedge (x_1 \geq 10) \wedge HB(s_1, \widetilde{x_3}) \wedge (c_1 = 1)$$
$$\wedge (1 \geq \widetilde{c_1}) \wedge (\widetilde{x_3} + \widetilde{c_1} \geq 11))$$
$$= (res = x_3 + c_1).$$
$$\wedge \exists\widetilde{x_3}\exists\widetilde{c_1}\,((x_3 = x_2) \wedge (x_1 \geq 10) \wedge (x_2 = 10) \wedge (10 \geq \widetilde{x_3}) \wedge (c_1 = 1)$$
$$\wedge (1 \geq \widetilde{c_1}) \;\; \wedge (\widetilde{x_3} + \widetilde{c_1} \geq 11)).$$

Таким образом, для вычисления достаточного условия ошибки в вызывающей функции необходимо поместить в резюме значение атрибут $VS(p_7, res)$. При применении резюме следует сопоставить формальные и фактические аргументы, результат вызова и возвращаемое значение и т.п., например, идентификатору x_1 будет сопоставлен идетификатор a в контексте вызывающей функции. По этому правилу будет вычислено значение $VS(p_{12}, idx)$ путём последовательной миграции узлов дерева $VS(p_7, res) = s_4$ (см. **Error! Reference source not found.**) от листьев к корню. Таким образом, достаточное условие ошибки:

$$NotLess(p_{12}, idx, 11) = HB(VS(p_{12}, idx), 11) = (res = x_3 + c_1)$$
$$\wedge \exists\widetilde{x_3}\exists\widetilde{c_1}\left((x_3 = x_2) \wedge (a \geq 10) \wedge (x_2 = 10) \wedge (10 \geq \widetilde{x_3}) \wedge (c_1 = 1)\right.$$
$$\left.\wedge (1 \geq \widetilde{c_1}) \wedge (\widetilde{x_3} + \widetilde{c_1} \geq 11)\right).$$

Данная формула выполнима при следующих значениях: $c_1 = 1, \widetilde{c_1} = 1, x_2 = 10, x_3 = 10, \widetilde{x_3} = 10, res = 11, a = 20$, следовательно, необходимо выдать предупреждение об ошибке.

К сожалению, данный подход сам по себе не позволит обнаружить ошибку, происходящую в функции `innerAccess2` на . 3. Т.к. анализ функций производится «снизу-вверх», то при анализе функции `plusOne` не было ничего известно о возможных значениях параметра `x`. Поэтому информация о возвращаемом значении, вычисляемом из параметра по пути **(2)-(4)-(5)-(6)-(7)** отсутствует в значении $VS(p_7, res)$. Для того, чтобы отслеживать такие межпроцедурные зависимости между значениями, был введено ещё два класса значений атрибута $FParam \cup AParam \subset Summary$. Значение атрибута типа $FParam = \{v | \; v \in VId\}$ сопоставляется каждому формальному аргументу функции и содержит его идентификатор значения. Далее производится обычный анализ и его результаты сохраняются в резюме. В результате значение $VS(p_7, res)$ будет иметь вид, изображенный на Рис. 5. *Значение* $\boldsymbol{VS(p_7, res)}$.

При применении резюме, если в контексте вызывающей функции идентификатор значения v, передаваемый в качестве фактического аргумента, имел некоторое непустое значение атрибута в точке вызова $VS(p_{call}, v) = s_{actual}$, то для всех мигрирующих из резюме значений атрибутов происходит подстановка на место листа-формального параметра нового значения $s_v = \langle v, s_{actual} | \; v \in VId, \; s_{actual} \in Summary \rangle \in AParam$. Т.к. в точке вызова $VS(p_{17}, \mathrm{c}_4) = s_{11} = \langle c_4, 4 \rangle \in Const$, то вместо $s_5 = \langle x_1 \rangle \in FParam$ будет подставлено значение $s_7 = \langle c_4, s_7 \rangle \in AParam$. В результате значение $VS(p_{18}, idx) = s_{10}$ будет иметь вид, изображенный на Рис. .

Во время анализа процедуры никакой априорной информации о значениях её параметров нет, поэтому:

$$s_v = \langle v \rangle \in FParam \Rightarrow HB(s_v, x) = LB(s_v, x) = false.$$

Для значений, обозначающих фактические параметры выполнено:

$$s_v = \langle v, s_{actual} \rangle \in AParam \Rightarrow \begin{matrix} HB(s_v, x) = HB(s_{actual}, x) \\ LB(s_v, x) = LB(s_{actual}, x) \end{matrix}.$$

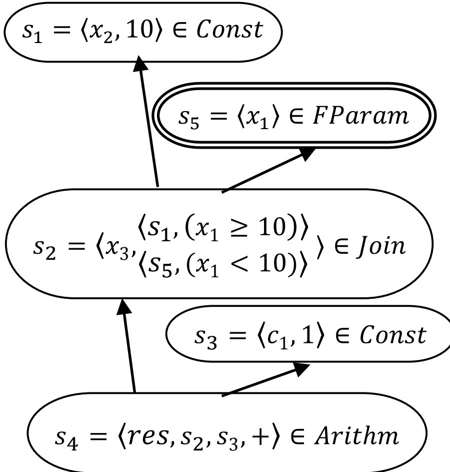


Рис. 5. Значение $\boldsymbol{VS(p_7, res)}$

Fig. 5. Value of $\boldsymbol{VS(p_7, res)}$

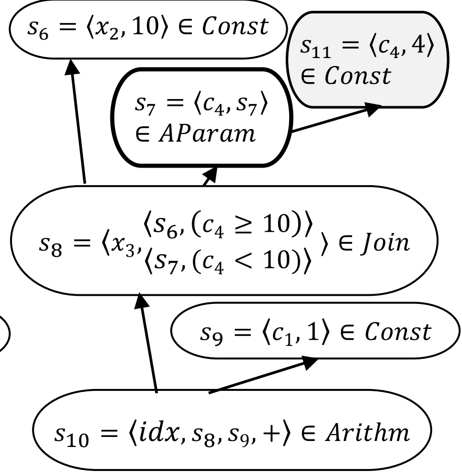


Рис. 6. Значение $\boldsymbol{VS(p_{18}, idx)}$

Fig. 6. Value of $\boldsymbol{VS(p_{18}, idx)}$

3.4. Построение достаточных условий ошибки для функции

Для поиска ошибок доступа к буферу в ситуациях, когда размер буфера или значение индекса определяется в одной из функций-предков (с точки зрения графа вызовов) по отношению к инструкции доступа к буферу, было введено два типа факта доступа к буферу внутри функции, сохраняемых в резюме для последующего для анализа в вызывающей функции:

$$KnowBufferAccess = \{\langle s_{idx}, accessCond, bufferSize\rangle \mid s_{idx} \in Summary, \quad accessCond \in Cond, \quad bufferSize \in \mathbb{N}\}$$

$$UnknowBufferAccess = \{\langle s_{idx}, accessCond, bufferVId\rangle \mid s_{idx} \in Summary, \quad accessCond \in Cond, \quad bufferVId \in VId\}$$

В данном разделе для краткости изложения будем рассматривать только ошибку выхода за правую границу буфера.

Рассмотрим произвольную инструкцию доступа к буферу p_{access}. Если для идентификатора значения индекса в данной точке значение атрибута VS не определено s_v, то информация об его возможных значениях отсутствует как в данной функции, так и во всех вызывающих, поэтому проверить такой доступ невозможно. Предположим, что значение атрибута VS определено для индекса и равно s_{idx}. Пометим в соответствующем ему графе все листовые вершины типа $FParam$, далее рекурсивно пометим все вершины, у которых хотя бы один из потомков помечен (кроме вершин типа $Double \subset Join$ (слияние двух значений с условиями), которые помечаются если оба потомка помечены). Непомеченные вершины соответствуют значениям, полностью определенным

в данной функции. Помеченные вершины соответствуют значениям, которые могут полностью определены только в вызывающей функции.

Если размер буфера известен в точке доступа и равен S, и в s_{idx} есть непомеченные вершины, то необходимо проверить переполнение в данной точке в ходе анализа текущей функции с помощью формулы (1). Если ошибка была найдена, то проверка этой инструкции доступа заканчивается.

В противном случае, если s_{idx} содержит помеченные вершины, то в резюме записывается факт доступа к буферу известного размера:

$$ac = \langle s_{idx}, ReachCond(p_{access}), S\rangle \in KnownBufferAccess.$$

Если размер буфера известен только в вызывающей функции (адрес буфера передан в качестве параметра v_{buf}), то в резюме записывается факт доступа к буферу неизвестного размера

$$ac = \langle s_{idx}, ReachCond(p_{access}), v_{buf}\rangle \in UnknownBufferAccess.$$

Теперь рассмотрим алгоритм применения резюме в точке p_{call}. Предположим, в нём содержится факт доступа к буферу внутри вызываемой функции. Значение s_{idx} при применении резюме трансформируется в значение s_{actual} по обычным правилам, описанным в предыдущем разделе. Рассмотрим случай, когда размер буфера либо был известен в вызываемой функции ($KnownBufferAccess$), либо стал известен при сопоставлении идентификатора значения буфера v_{buf} из факта доступа ($UnknownBufferAccess$), обозначим его за S'. Тогда, если в s_{actual} есть непомеченные вершины, то необходимо проверить наличие ошибки в данной точке, установив выполнимость формулы ($migratedCond$ – условие из факта доступа $accessCond$, транслированное в контекст вызывающей функции):

$$ReachCond(p_{call}) \wedge HB(s_{actual}, S') \wedge migratedCond.$$

Если ошибка была обнаружена, то обработка этой инструкции доступа заканчивается. В противном случае, если s_{actual} содержит помеченные вершины, то в резюме записывается факт доступа к буферу известного размера:

$$\langle s_{actual}, ReachCond(p_{call}) \wedge migratedCond, S'\rangle \in KnownBufferAccess.$$

Если адрес буфера в вызываемой функции был неизвестен, а после сопоставления его идентификатора значения с фактическим аргументом он оказался параметром текущей функции v'_{buf}, то в резюме текущей функции записывается новый факт доступа к буферу неизвестного размера:

$$\langle s_{actual}, ReachCond(p_{call}) \wedge migratedCond, v'_{buf}\rangle \in UnknownBufferAccess.$$

5. Реализация и результаты

Рассмотренный подход был реализован в рамках статического анализатора Svace. В рассмотренный в статье подход был внесён ряд технических изменений. Во-первых, для улучшения производительности были введены ограничения на размер значения атрибута VS как в рамках

внутрипроцедурного анализа, так и (более строгие) для сохранения в резюме. Кроме того, с той же целью был реализован алгоритм упрощения помещаемых в резюме формул (условий в узлах типа $Join$ и в фактах доступа к буферу $KnownBufferAccess$ и $UnknownBufferAccess$). Кроме этого, был выделен ряд типичных ситуаций, в которых нарушается предположение о контрактах, для которых были разработаны подавляющие ложные срабатывания эвристики. Так, например, зачастую сравнения некоторой переменной с параметром функции во многих контекстах всегда имеет одинаковый результат, поэтому нельзя полагаться на то, что обе ветки сравнения достижимы, поэтому такие сравнения игнорируются.

Табл. 1. Результаты работы детекторов на проекте Android 5.0.2

Table 2. Checker results on Android 5.0.2

Тип срабатывания	**Кол-во**	**TP, %**
BUFFER_OVERFLOW.EX	221	62
BUFFER_OVERFLOW.LIB.EX	64	64
OVERFLOW_AFTER_CHECK.EX	66	67

Результаты работы детектора на проекте Android 5.0.2 приведены в Табл. 1. В качестве инструкций доступа к буферу рассматривались обычные инструкции индексации и вызовы библиотечных функций, осуществляющих доступ к переданному в качестве аргумента буферу (например, `memcpy`). Исходя из этого детектор выдает предупреждения двух типов: BUFFER_OVERFLOW.EX и BUFFER_OVERFLOW.LIB.EX.

Кроме того, разработан эвристический алгоритм, который, используя информацию об индуктивных переменных и граничных условиях цикла, строит значения VS для переменных цикла и ищет ошибочные ситуации на основе этих значений. Детектор, разработанный на его основе, выдает предупреждения типа OVERFLOW_AFTER_CHECK.EX

5. Обзор существующих подходов

В работе [8] был проведён подробный сравнительный анализ работ, посвященных поиску ошибок доступа к буферу с помощью статического анализа. Существующие походы можно сравнивать с точки зрения следующих критериев:

- анализ исходного, либо бинарного кода;
- полностью автоматический, либо автоматизированный анализ;
- чувствительность к потоку, путям, контексту;
- внутрипроцедурный, либо межпроцедурный анализ;
- масштабируемость.

С точки зрения данной классификации наиболее близкой к данной работе можно считать инструмент ARCHER [9], реализованный на OCaml. Данный инструмент не требует от пользователя никаких дополнительных данных о программе, способен анализировать большие программы (анализ Linux 2.5.53, представляющий собой 2158 файлов и 1,6 млн. строк кода, занял 4 часа). При этом сохраняется высокий процент истинных срабатываний (65% из 139 срабатываний на Linux 2.5.53). В основе его подхода к анализу лежит традиционное символьное выполнение с ограничением на количество рассмотренных путей и время анализа одной функции (авторы утверждают, что при анализе Linux в среднем в функции было покрыто 96% путей). Поиск межпроцедурных срабатываний организован с помощью метода резюме. К минусам, присущим как инструменту ARCHER, так и рассматриваемому в данной статье, можно отнести отсутствие полной поддержки библиотечных функций работы со строками в языке C.

6. Заключение

В данной работе был метод поиска межпроцедурных ошибок доступа к буферу основанный на символьном исполнении с объединением состояний. Данный алгоритм является чувствительным к путям и учитывает взаимосвязи между переменными, такие как сравнения, арифметические операции и инструкции приведения типа, и кроме этого, взаимосвязь значений переменных между различными функциями, что позволяет обнаруживать межпроцедурные дефекты. Описанный подход был реализован в инструменте статического анализа Svace. На проекте Android 5.0.2 было получено 351 предупреждение об ошибке доступа к буферу, среди которых 64% оказались истинными, при этом существенного замедления анализа не произошло.

Список литературы

[1]. И.А. Дудина, В.К. Кошелев, А.Е. Бородин, Поиск ошибок доступа к буферу в программах на языке C/C++. Труды ИСП РАН, том 28, вып. 4, 2016, стр. 149-168. DOI: 10.15514/ISPRAS-2016-28(4)-9

[2]. В.К. Кошелев, И.А. Дудина, В.И. Игнатьев, А.И. Борзилов, Чувствительный к путям поиск дефектов в программах на языке C# на примере разыменования нулевого указателя, Труды ИСП РАН, том 27, вып. 5, 2015, стр. 59-86. DOI: 10.15514/ISPRAS-2015-27(5)-5

[3]. D. Larochelle, D. Evans. Statically detecting likely buffer overflow vulnerabilities. 10th USENIX Security Symposium, Washington, D.C., August 2001.

[4]. В.П. Иванников, А.А. Белеванцев, А.Е. Бородин, В.Н. Игнатьев, Д.М. Журихин, А.И. Аветисян, М.И. Леонов. Статический анализатор Svace для поиска дефектов в исходном коде программ. Труды ИСП РАН, том 26, 2014 г., стр. 231–250. DOI: 10.15514/ISPRAS-2014-26(1)-7.

[5]. V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. 2012. Efficient state merging in symbolic execution. SIGPLAN Not. 47, 6 (June 2012), 193-204. DOI: 10.1145/2345156.2254088

[6]. А.Е. Бородин, А.А. Белеванцев. Статический анализатор Svace как коллекция анализаторов разных уровней сложности. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 111-134. DOI: 10.15514/ISPRAS-2015-27(6)-8.
[7]. А.Е. Бородин, Межпроцедурный контекстно-чувствительный статический анализ для поиска ошибок в исходном коде программ на языках Си и Си++: дис. канд. ф.-м. наук. Москва, 2016 г.
[8]. Shahriar, H., and Zulkernine, M. Classification of static analysis-based buffer overflow detectors. SSIRI-C 2010 - 4th IEEE International Conference on Secure Software Integration and Reliability Improvement Companion, 2010, pp. 94-101.
[9]. Y. Xie, A. Chou, and D. Engler, "ARCHER: Using Symbolic, Path-sensitive Analysis to Detect Memory Access Errors," Proceedings of the 9th European Software Engineering Conference, Helsinki, Finland, 2003, pp. 327-336.

Inter-procedural buffer overflows detection in C/C++ source code via static analysis

I. Dudina <eupharina@ispras.ru>
ISP RAS,
25 Alexander Solzhenitsyn Str., Moscow, 109004, Russian Federation
CMC MSU, CMC faculty, 2 educational building,
MSU, Leninskie gory str., Moscow 119991, Russian Federation

Abstract. We propose inter-procedural static analysis tool for buffer overflow detection. It is based on previously developed intra-procedural algorithm which uses symbolic execution with state merging. This algorithm is path-sensitive and supports tracking several kinds of value relations such as arithmetic operations, cast instructions, binary relations from constraints. In this paper we provide a formal definition for inter-procedural buffer overflow errors and discuss different kinds of such errors. We use function summaries for inter-procedural analysis, so it provides natural path-sensitivity in some degree. This approach allowed us to improve intra-procedural algorithm by tracking inter-procedural value dependencies. Furthermore, we introduce a technique to extract the sufficient condition of buffer overflow for a function, which is supposed to be stored in the summary of this function and checked at every call site. This approach was implemented for Svace static analyzer as the new buffer overflow detector, and it has shown 64% true-positive ratio on Android 5.0.2.

Keywords: static analysis, software error detection, buffer overflow, path-sensitivity, symbolic execution, context-sensitivity, inter-procedural analysis.

DOI: 10.15514/ISPRAS-2016-28(5)-7

For citation: I. Dudina. Inter-procedural buffer overflows detection in C/C++ source code via static analysis. Trudy ISP RAN/Proc. ISP RAS, 2016, vol. 28, issue 5, 2016, pp. 119-134 (in Russian). DOI: 10.15514/ISPRAS-2016-28(5)-7

References

[1]. I. Dudina, V. Koshelev, A. Borodin. [Statically detecting buffer overflows in C/C++]. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 4, 2016, pp. 149-168 (in Russian). DOI: 10.15514/ISPRAS-2016-28(4)-9

[2]. V. Koshelev, I. Dudina, V. Ignatyev, A. Borzilov. [Path-Sensitive Bug Detection Analysis of C# Program Illustrated by Null Pointer Dereference], Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 5, 2015, pp. 59-86 (in Russian). DOI: 10.15514/ISPRAS-2015-27(5)-5

[3]. D. Larochelle, D. Evans. Statically detecting likely buffer overflow vulnerabilities. 10th USENIX Security Symposium, Washington, D.C., August 2001.

[4]. V.P. Ivannikov, A.A. Belevantsev, A.E. Borodin, V.N. Ignatiev, D.M. Zhurikhin, A.I. Avetisyan, M.I. Leonov. [Static analyzer Svace for finding of defects in program source code]. Trudy ISP RAN/Proc. ISP RAS, vol. 26, issue 1, 2014, pp. 231-250 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-7

[5]. V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. 2012. Efficient state merging in symbolic execution. SIGPLAN Not. 47, 6 (June 2012), 193-204. DOI: 10.1145/2345156.2254088

[6]. A. Borodin, A. Belevancev. [A Static Analysis Tool Svace as a Collection of Analyzers with Various Complexity Levels]. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, pp. 111-134 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-8.

[7]. A. Borodin. PhD thesis. Interprocedural contex-sensitive static analysis for error detection in C/C++ source code. ISP RAN, Moscow, 2016

[8]. Shahriar, H., and Zulkernine, M. Classification of static analysis-based buffer overflow detectors. SSIRI-C 2010 - 4th IEEE International Conference on Secure Software Integration and Reliability Improvement Companion, 2010, pp. 94-101.

[9]. Y. Xie, A. Chou, and D. Engler, “ARCHER: Using Symbolic, Path-sensitive Analysis to Detect Memory Access Errors,” Proceedings of the 9th European Software Engineering Conference, Helsinki, Finland, 2003, pp. 327-336.

Поиск ошибок выхода за границы буфера в бинарном коде программ[*]

В.В. Каушан <korpse@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. В статье рассматривается метод поиска ошибок выхода за границы буфера в рамках метода комбинированного (статико-динамического) анализа бинарного кода. Для поиска ошибок используется символьная интерпретация последовательности машинных инструкций, выполненных за время работы программы. Рассматриваются способы увеличения точности анализа за счёт анализа циклов работы со строками, а также предварительного расширения покрытия кода. С помощью инструмента, разработанного на основе предложенных методов, были найдены как известные ошибки, так и ошибки, информация о которых ранее не была опубликована.

Ключевые слова: поиск ошибок; бинарный код; динамический анализ; символьное выполнение.

DOI: 10.15514/ISPRAS-2016-28(5)-8

Для цитирования: В.В. Каушан. Поиск ошибок выхода за границы буфера в бинарном коде программ. Труды ИСП РАН, том 28, вып. 5, 2016, стр. 135-144. DOI: 10.15514/ISPRAS-2016-28(5)-8

1. Введение

В настоящее время особенно остро стоит задача обеспечения безопасности информационных систем. Наиболее частой причиной нарушения безопасности в таких системах являются уязвимости в программном обеспечении этих систем, позволяющие нарушить конфиденциальность, доступность или целостность обрабатываемой информации. В связи с этим актуальной является задача поиска ошибок и уязвимостей в программном обеспечении.

Одним из наиболее распространённых типов уязвимостей является уязвимость переполнения буфера, уступающая по распространённости лишь XSS и SQL-инъекциям, которые присущи веб-приложениям [1]. Эксплуатация этого типа уязвимости во многих случаях позволяет выполнить произвольный код в

[*] Работа поддержана грантом РФФИ № 16-29-09632

рамках уязвимого приложения и, таким образом, скомпрометировать систему. Кроме того, уязвимость "Heartbleed" [2] в OpenSSL продемонстрировала, что большую опасность может представлять не только переполнение буфера (при записи данных), но и выход за границы буфера (при чтении данных). Таким образом, поиск ошибок работы с буферами в памяти является актуальной задачей.

2. Поиск ошибок с помощью символьной интерпретации

Предложенный метод поиска ошибок основан на анализе результатов наблюдения за выполнением программы. Такой анализ может быть проведен с помощью отладчиков, систем post-mortem анализа трасс выполнения [3-5], а также систем динамического двоичного инструментирования (DBI) таких как Pin [6] и Valgrind [7]. Анализируется последовательность машинных инструкций, которые выполнялись в рамках одного запуска исследуемой программы. Анализ на уровне машинных инструкций позволяет находить ошибки в программах в тех случаях, когда исходный код программы или отладочная информация недоступны.

В качестве основного механизма поиска ошибок используется символьная интерпретация последовательности машинных инструкций. Каждая инструкция транслируется в уравнения для SMT решателя [8] относительно входных данных программы. Полученный набор уравнений для некоторого пути в программе, называемый предикатом пути, отражает прохождение программы по этому пути. К этим уравнениям добавляются уравнения (называемые предикатом безопасности), описывающие ошибочную ситуацию в программе в некоторый момент её выполнения. Если полученная система уравнений оказалась совместной - её решением будет набор входных данных, приводящих к проявлению ошибки.

Для поиска ошибок выхода за границы буфера дополнительно вводится понятие символьной длины буферов - дополнительные переменные, связанные с длинами соответствующих буферов. С помощью оставления уравнений над этими переменными возможен поиск ситуаций выхода за границы буфера как при единичном чтении, так и при последовательном доступе к памяти (например, при копировании или вычислении длины строк). Такой подход позволяет абстрагироваться от реальных размеров данных, обрабатываемых программой, и таким образом находить ошибки работы с памятью, когда размер обрабатываемых данных потенциально может превышать размер буфера, выделенного под эти данные. Кроме того, символьная интерпретация отдельных инструкций дополняется интерпретацией укрупнённых блоков программы, таких как вызовы библиотечных функций и экземпляры циклов программы. Правила интерпретации подробно описаны в работе [9].

Данная работа является продолжением работы [9] и предлагает методы увеличения точности анализа с помощью анализа циклов работы со строками,

а также с помощью предварительного расширения покрытия кода анализируемой программы.

3. Анализ циклов

Одной из важных задач, возникающих во время анализа программ, является анализ циклов. Ошибка в программе может проявиться в результате выполнения большого числа итераций цикла, для обнаружения такой ситуации необходимо иметь возможность вычисления числа итераций цикла, при котором достигается ошибочная ситуация. Кроме того, во многих случаях, число итераций цикла зависит непосредственно от обрабатываемых данных и такие циклы в случае символьной интерпретации должны анализироваться, как если бы число итераций цикла было бы произвольным.

В рамках задачи динамического анализа кода проблема анализа циклов имеет свою специфику: последовательность выполненных инструкций уже содержит выполнение некоторого количества итераций цикла, а символьная интерпретация инструкций этих итераций накладывает ограничения на входные данные, неявно фиксирующие число итераций цикла. Для того чтобы абстрагироваться от числа итераций цикла, необходимо рассматривать этот цикл как единое целое по аналогии с библиотечными функциями с известной семантикой.

В рамках предлагаемого метода выполняется анализ циклов работы со строками, таких как циклы копирования строк и циклы вычисления длины строки. Во время анализа происходит обнаружение таких циклов, извлекаются параметры соответствующих им операций (адреса и длины строк), после чего такие циклы интерпретируются как единое целое по аналогии с вызовами функций strcpy и strlen. Циклы, которые не были обнаружены в процессе анализа, специальным образом не обрабатываются.

Обнаружение циклов работы со строками происходит в несколько этапов. На первом этапе для каждого цикла определяется множество индуктивных переменных и характер их изменения. Под переменной понимается регистр или ячейка памяти с постоянным адресом, связанная с конкретной инструкцией в коде программы. Для таких переменных анализируются значения переменной на последовательных итерациях цикла. На основе анализа нескольких итераций цикла делается вывод о характере изменения значения переменной. Если значение переменной изменяется линейно с одним и тем же шагом на всех итерациях цикла, переменная включается в множество индуктивных переменных цикла. Далее для каждого цикла определяется множество буферов в памяти, соответствующих следующим критериям:

- переменная, используемая в качестве адреса ячейки памяти является индуктивной;
- на каждой итерации цикла происходит обращение к последовательным адресам памяти;

- размеры обрабатываемых ячеек соответствуют размеру символа для одного из типов строк (1 или 2 байта) и равны шагу переменной, используемой в качестве адреса;
- данные буфера представляют собой нуль-терминированную строку.

На этом этапе фиксируется характер доступа к каждому буферу (чтение и/или запись), его размер, а также размер адресуемой ячейки.

На основе полученных данных производится классификация циклов на принадлежность к одному из видов.

Цикл считается циклом копирования строк, если выполнены следующие критерии:

- существует два буфера, к которым осуществляется доступ на каждой итерации цикла;
- к первому буферу обращаются только на чтение;
- ко второму буферу обращаются только на запись;
- на каждой итерации цикла значения, прочитанные из первого буфера, и значения, записанные во второй, совпадают.

Цикл считается циклом вычисления длины строки, если выполнены следующие критерии:

- существует один буфер, к которому осуществляется доступ на каждой итерации цикла;
- к буферу обращаются только на чтение;
- значение одной из индуктивных переменных после последней итерации цикла равно фактической длине строки, находящейся в буфере.

Все остальные циклы классифицируются как циклы с неизвестной семантикой. Если в цикле происходит одновременно копирование строки и вычисление её длины, то циклу присваиваются оба класса. В дальнейшем, различные семантики такого цикла обрабатываются независимо друг от друга.

Циклы копирования и вычисления длины строк обрабатываются аналогично вызовам функций strcpy и strlen. Для каждого такого цикла определяются его границы и описываются значения фактических параметров, как если бы цикл являлся вызовом соответствующей функции. Такой подход позволяет единообразно обрабатывать как вызовы строковых функций, так и эквивалентные им циклы.

4. Расширение покрытия кода

Недостатком анализа последовательности инструкций для одного запуска программы является отсутствие возможности итеративного анализа путей выполнения. Этот недостаток можно компенсировать возможностью анализа нескольких запусков исследуемой программы. Входные данные для таких запусков подбираются так, чтобы максимизировать суммарное покрытие кода

исследуемой программы, что приводит к увеличению вероятности обнаружения ошибки.

Предлагаемый метод получения такого набора входных данных, максимизирующих суммарное покрытие кода, основан на динамическом онлайн символьном выполнении. В процессе символьного выполнения назначаются символьные значения заданным переменным программы и эти символьные значения распространяются по мере выполнения программы, при этом все преобразования в программе, в которых участвуют символьные значения, транслируются в соответствующие уравнения. Для хранения символьных значений переменных программы поддерживается *состояние*, описывающее отображение множества переменных программы на множество соответствующих им символьных значений, а также выполняемую на данный момент операцию программы. Если в процессе выполнения встречается ветвление, зависящее от символьных данных, порождается два состояния, соответствующие двум веткам ветвления, и выполнение продолжается дальше для каждого из состояний.

Обычно символьные значения назначают ячейкам памяти, содержащим входные данные программы. В этом случае, различные состояния соответствуют различным путям выполнения в программе при обработке входных данных. С помощью SMT-решателя для каждого состояния и соответствующего ему пути можно получить подтверждающий набор входных данных. Кроме того, если производится выполнение бинарного кода, часто можно получить покрытие кода в терминах базовых блоках. Анализ покрытия кода для каждого из состояний позволяет получить набор входных данных, для которого, в совокупности, достигается существенный прирост покрытия кода по сравнению с единичным запуском программы.

Пути выполнения, соответствующие порождаемым состояниям, представляют собой древовидную структуру. Это приводит к тому, что покрытие кода, соответствующее двум соседним состояниям отличается незначительно. Кроме того, с каждым ветвлением количество состояний в программе растёт по экспоненциальному закону, что приводит к огромному количеству анализируемых состояний и неэффективности анализа соответствующих запусков. Для решения этой проблемы можно выделить такое подмножество состояний, суммарное покрытие кода для которого будет таким же, как и для всего множества состояний. Это возможно сделать с помощью классической задачи о покрытии множества. Задача о покрытии относится к классу NP-полных, но может быть эффективно решена приближённым алгоритмом, который даёт приемлемый результат. Таким способом на практике удаётся уменьшить количество рассматриваемых состояний на несколько порядков.

Для оценки покрытия используется метрика покрытия кода по базовым блокам, так как использование этой метрики оказывается достаточным для поиска ошибок, привязанных к конкретному месту программы. В большинстве случаев выход за границы буфера происходит из-за отсутствия

проверок на размер буфера непосредственно перед операцией копирования. Для поиска таких ошибок требуется покрыть как можно большее количество мест в программе, в которых происходит копирование данных, что, в свою очередь, можно свести к задаче получения хорошего покрытия в терминах базовых блоков.

Поскольку, в общем случае, процесс перебора путей выполнения может выполняться неопределённо долго, требуется критерий завершения процесса перебора путей. Таким критерием может быть оценка прироста покрытия за определённый период времени. Если за заданный интервал времени прирост покрытия в терминах базовых блоков оказался меньше, чем некоторое пороговое значение, перебор путей завершается.

Для реализации описанного метода использовался инструмент S2E [10], предоставляющий возможности динамического онлайн символьного выполнения в рамках полносистемного эмулятора QEMU. В качестве входных данных для этого инструмента выступает исследуемая программа и начальный набор входных данных. В процессе работы S2E выполняется перебор путей при помощи инвертирования условных переходов, зависящих от символьных данных. В результате перебора путей в программе для каждого завершённого пути формируется набор входных данных, а также достигаемое покрытие кода на этом пути. Далее количество таких путей минимизируется с целью получения минимального набора входных данных, на котором достигалось бы такое же покрытие, как и на полном наборе входных данных. Полученный набор входных данных далее используется непосредственно для анализа каждого из запусков.

5. Апробация

Предлагаемый метод поиска ошибок был реализован и апробирован на программах, работающих под ОС Linux и Windows, а также на программном обеспечении для маршрутизатора, выпускаемого одним из крупных производителей сетевого оборудования. Были найдены ошибки, связанные как с записью, так и с чтением данных за пределами буфера. С помощью метода расширения покрытия удавалось получить прирост покрытия до 25% от такового для начального набора входных данных. Кроме того, с помощью расширения покрытия для приложения mkfs.jfs был автоматически восстановлен список поддерживаемых аргументов командной строки. В таблице 1 приведены наиболее показательные результаты применения предложенного метода поиска ошибок. Время анализа для большинства примеров не превышало нескольких минут. Следует отметить, что анализу предшествует стадия расширения покрытия кода (0.5-4 часа). В столбце "исходный размер данных" указан размер данных, который подавался на вход программе во время анализа и не приводил к ошибке, а в столбце "конечный размер данных" – размер данных, приводящий к ошибке работы с памятью. Для каждого приложения было обработано только первое срабатывание

инструмента, в общем случае возможно получение всех срабатываний ценой незначительного увеличения времени анализа. Также следует отметить, что случаи ложноположительных и ложноотрицательных срабатываний инструмента не встречались. Запуск инструментов проводился на машине с конфигурацией Intel Xeon E5-2650 v2, 32Gb RAM, 500Gb HDD, на всех этапах было задействовано только одно ядро процессора.

Табл. 1. Результаты применения метода.

Table 1. Method approbation results.

ОС	Программа	Размер данных		Размещение буфера	CVE / OSVDB
		исходный	конечный		
Linux	openssl	18	25	куча	CVE: 2014-0160
Linux	alsa_out	14	95	стек	-
Linux	mkfs.jfs	31	436	стек	-
Linux	prepmx	11	813	стек	-
WinXP SP2	httpdx	329	330	куча	OSVDB-ID: 84454
прошивка маршрути-затора	pptp	16	257	стек	-

6. Заключение

В статье представлен метод поиска ошибок выхода за границы буфера в бинарном коде программ. В частности, рассматривается метод анализа циклов, а также метод расширения покрытия кода, позволяющие улучшить точность при поиске ошибок и потенциально уменьшить количество ложноотрицательных срабатываний инструмента. Методы были реализованы в виде программных инструментов, работающих в рамках инструмента динамического анализа бинарного кода, а также среды динамического символьного выполнения S2E.

Список литературы

[1]. Younan Y. 25 Years of Vulnerabilities: 1988-2012. Sourcefire Vulnerability Research Team. – 2013.

[2]. CVE-2014-0160: https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160

[3]. В.А. Падарян, А.И. Гетьман, М.А. Соловьев, М.Г. Бакулин, А.И. Борзилов, В.В. Каушан, И.Н. Ледовских, Ю.В. Маркин, С.С. Панасенко. Методы и программные средства, поддерживающие комбинированный анализ бинарного кода. Труды ИСП РАН, том 26, вып. 1, 2014 г., стр. 251-276. DOI: 10.15514/ISPRAS-2014-26(1)-8

[4]. П.М. Довгалюк, Н.И. Фурсова, Д.С. Дмитриев. Перспективы применения детерминированного воспроизведения работы виртуальной машины при решении

задач компьютерной безопасности. Материалы конференции РусКрипто'2013. Москва, 27 – 30 марта 2013.

[5]. Довгалюк П.М., Макаров В.А., Падарян В.А., Романеев М.С., Фурсова Н.И. Применение программных эмуляторов в задачах анализа бинарного кода. Труды ИСП РАН, том 26, вып. 1, стр. 277-296. DOI: 10.15514/ISPRAS-2014-26(1)-9

[6]. Alex Skaletsky, Tevi Devor, Nadav Chachmon, Robert Cohn, Kim Hazelwood, Vladimir Vladimirov, Moshe Bach. Dynamic Program Analysis of Microsoft Windows Applications. International Symposium on Performance Analysis of Software and Systems (ISPASS). White Plains, NY. April 2010.

[7]. Nicholas Nethercote. Dynamic Binary Analysis and Instrumentation or Building Tools is Easy. A dissertation submitted for the degree of Doctor of Philosophy at the University of Cambridge, 2004.

[8]. Silvio Ranise and Cesare Tinelli. The SMT-LIB Format: An Initial Proposal. Proceedings of PDPAR'03, July 2003

[9]. В.В. Каушан, А.Ю. Мамонтов, В.А. Падарян, А.Н. Федотов. Метод выявления некоторых типов ошибок работы с памятью в бинарном коде программ. Труды ИСП РАН, том 27, вып. 2, 2015, стр. 105-126. DOI: 10.15514/ISPRAS-2015-27(2)-7

[10]. Chipounov V., Kuznetsov V., Candea G. S2E: A platform for in-vivo multi-path analysis of software systems. In Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems, 2011, pp. 265–278.

Buffer overrun detection method in binary code★

V.V. Kaushan <korpse@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. Buffer overflows are one of the most common and dangerous software errors. Exploitation of such errors can lead to an arbitrary code execution and system disclosure. This paper considers a method for detecting memory violations. The method is based on combined (static-dynamic) analysis of binary code. Analysis is based on symbolic interpretation of machine instructions executed during a single program run. Proposed method also provides abstraction from buffer sizes and can reveal sizes that cause buffer overflow errors. Analysis can be applied to program binaries and doesn't require a source code. Two techniques are proposed to improve method precision: cycle analysis and code coverage increase. Cycle analysis is one of the cumbersome problems in dynamic analysis. Separate cycle instruction analysis leads to an excess of constraints over input data that causes potential false negatives. The proposed technique is able to analyze cycles entirely and abstract from number of cycle iterations. One of the drawbacks of a single run analysis is an insufficient code coverage which prevents some errors from discovery. The technique proposed to increase code coverage is based on a dynamic symbolic execution. Some minimal path set from discovered code paths is selected and used to achieve better code coverage than from a single run. Inputs corresponding to each path from selected set are used to analyze several program runs. Proposed techniques were implemented and used to discover both known and non-disclosed bugs.

Keywords: bug finding; binary code; dynamic analysis; symbolic execution.

DOI: 10.15514/ISPRAS-2016-28(5)-8

For citation: V.V. Kaushan. Buffer overrun detection method in binary code. Trudy ISP RAN/Proc. ISP RAS, 2016, vol. 28, issue 5, 2016, pp. 135-144 (in Russian). DOI: 10.15514/ISPRAS-2016-28(5)-8

References

[1]. Younan Y. 25 Years of Vulnerabilities: 1988-2012 Sourcefire Vulnerability Research Team. – 2013.
[2]. CVE-2014-0160 https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160
[3]. Padaryan V. A., Getman A. I., Solovyev M. A., Bakulin M. G., Borzilov A. I., Kaushan V. V., Ledovskikh I. N., Markin Yu. V., Panasenko S. S. Methods and software

★ The paper is supported by RFBR grant 16-29-09632

tools to support combined binary code analysis. Programming and Computer Software. September 2014, Volume 40, Issue 5, pp 276-287.

[4]. Dovgalyuk P.M., Fursova N.I., Dmitriev D.S. Prospects of using virtual machine deterministic replay insolving computer security problems. The Proceedings RusCrypto'2013, 2014 (In Russian).

[5]. Dovgalyuk P.M., Makarov V.A., Romaneev M.S., Fursova N.I. [Applying program emulators for binary code analysis]. Trudy ISP RAN/Proc. ISP RAS, vol. 26, issue 1, 2014, pp. 277-296. DOI: 10.15514/ISPRAS-2014-26(1)-9.

[6]. Alex Skaletsky, Tevi Devor, Nadav Chachmon, Robert Cohn, Kim Hazelwood, Vladimir Vladimirov, Moshe Bach. Dynamic Program Analysis of Microsoft Windows Applications. International Symposium on Performance Analysis of Software and Systems (ISPASS). White Plains, NY. April 2010.

[7]. Nicholas Nethercote. Dynamic Binary Analysis and Instrumentation or Building Tools is Easy. A dissertation submitted for the degree of Doctor of Philosophy at the University of Cambridge, 2004.

[8]. Silvio Ranise and Cesare Tinelli. The SMT-LIB Format: An Initial Proposal. Proceedings of PDPAR'03, July 2003

[9]. Kaushan V.V., Mamontov A.Yu., Padaryan V.A., Fedotov A.N. [Memory violation detection method in binary code]. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 2, 2015, pp. 105-126 (in Russian). DOI: 10.15514/ISPRAS-2015-27(2)-7

[10]. Chipounov V., Kuznetsov V., Candea G. S2E: A platform for in-vivo multi-path analysis of software systems. In Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems, 2011, pp. 265–278.

Использование анализа недостижимого кода в статическом анализаторе для поиска ошибок в исходном коде программ

Р.Р. Мулюков <eygz@ispras.ru>
А.Е. Бородин <alexey.borodin@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. В статье описывается поиск недостижимого кода в исходном коде программ, написанных на языках Си и Си++. Приведена классификация видов недостижимого кода. Описаны цели, достигаемые поиском недостижимого кода: выдача предупреждений о возможных ошибках в анализируемой программе и улучшение точности других анализов. Формально поставлены три задачи анализа потока данных: анализ интервалов значений, анализ выколотой точки, предикатный анализ. Решения этих задач применены для поиска инвариантных условий ветвления программы. Показаны особенности поиска недостижимого кода в статических анализаторах, предназначенных для поиска ошибок. Отмечены общие ситуации, в которых нет необходимости сообщать пользователю о найденном недостижимом коде. Описанные алгоритмы реализованы в статическом инструменте Svace, разрабатываемом в ИСП РАН. Оценка результатов детекторов произведена для исходного кода операционных систем Android-5.02 и Tizen-2.3 в виде количественного сравнения предупреждений, выданных каждым из анализов, и их пересечения между собой.

Ключевые слова: статический анализ; недостижимый код; анализ потока данных; Svace; поиск ошибок.

DOI: 10.15514/ISPRAS-2016-28(5)-9

Для цитирования: Р.Р. Мулюков, А.Е. Бородин. Использование анализа недостижимого кода в статическом анализаторе для поиска ошибок в исходном коде программ. Труды ИСП РАН, том 28, вып. 5, 2016, стр. 145-158. DOI: 10.15514/ISPRAS-2016-28(5)-9

1. Введение

В статье описывается реализация анализов недостижимого кода в инструменте статического анализа Svace, предназначенного для поиска ошибок в исходном коде программ, написанных на языках Си и Си++ [1-3].

Общая схема реализации анализа недостижимого кода в Svace и находимые ситуации приведены в [4], в данной статье формально описываются используемые алгоритмы и произведена оценка результатов анализа.

1.1 Классификация недостижимого кода

Недостижимым кодом называются инструкции программы, которые ни при каком её выполнении не могут быть достигнуты. Можно выделить несколько разновидностей недостижимого кода:

(1) “По управлению” – это ситуации, когда инструкции в графе потока управления недостижимы от входной вершины.

```
void foo() {
    ...
    return;
    a = 1; // недостижимый код
}
```

```
void bar() {
    ...
    goto label;
    a = 1; // недостижимый код
label:
    a = 2;
}
```

(2) Инструкции недостижимы из-за вызова функций, завершающих выполнение программы.

```
void foo() {
    ...
    fatal_error(1);
    a = 1; // недостижимый код
}
```

(3) “Из-за сравнения” – инвариантность сравнения при ветвлении программы.

```
void foo() {
    int a = 0;
    int b = 1;
    if (a > b) {
        ... // недостижимый код
    }
}
```

```
void bar(char *p) {
    if (!p)
        return;

    ... // p не меняется

    if (p) {
        ...
    } else {
        ... // недостижимый код
    }
}
```

1.2 Цели поиска недостижимого кода

В статическом анализаторе для поиска ошибок в исходном коде программ анализ недостижимого кода используется как для выдачи предупреждений об ошибках в программе, так и для улучшения точности других анализов.

Поскольку программисты, за исключением некоторых ситуаций, нарочно не пишут код, который не может быть исполнен, то наличие такого кода вполне наверняка свидетельствует

- об ошибке в реализации задуманного алгоритма,
- об устаревшем коде, который забыли удалить,
- о непонимании программистом программы, которую он изменяет.

Поэтому существует необходимость в поиске недостижимого кода и выдачи пользователю предупреждения.

```
void foo() {
    ...
    for (i = 0; i < n; ++i) { // "++i" недостижима
        a[i] = i;
        if (m--);  // ошибочная точка с запятой
            break;
    }
}
```

Рис. 1. Недостижимый код как следствие допущенной ошибки

Fig. 1. Unreachable code as a result of a defect

Также обнаружение недостижимого кода полезно и самому статическому анализатору в качестве предварительного этапа анализа, для того чтобы на последующих этапах исключать из рассмотрения найденные недостижимые инструкции программы, что должно благоприятно сказываться на точности его работы.

1.3 Используемые анализы для поиска недостижимого кода

Из-за временных ограничений компиляторы находят недостижимый код для относительно простых случаев. В компиляторах обычно реализован поиск недостижимого кода “по управлению” при построении графа потока управления. Также находятся некоторые инвариантные сравнения при помощи анализа распространения констант.

К статическим анализам применяются менее жёсткие ограничения по времени работы, благодаря чему можно реализовывать более сложные виды анализов. Были реализованы следующие виды анализов: анализ интервалов значений, анализ выколотой точки, предикатный анализ. Для повышения точности анализов применялась нумерация значений.

Отметим, что при анализе не используется анализ алиасов, поэтому описываемые далее анализы потока данных оперируют только локальными переменными, чей адрес не передавался в функции и другим переменным.

Для поиска недостижимого кода "из-за вызова функций", завершающих выполнение программы, был реализован обратный межпроцедурный анализ, который для каждой функции выясняет, что её вызов приведёт к завершению программы.

2 Анализ недостижимого кода

В этом разделе формально описаны реализованные анализы потока данных, которые применялись для поиска инвариантных сравнений в программе.

2.1 Анализ интервалов значений

Анализ интервалов значений вычисляет в каждой точке программы интервалы, покрывающие все возможные значения целочисленных переменных.

Интервалы значений позволяют обнаруживать инвариантные сравнения в программе. Так как интервалы покрывают все возможные значения переменных, то если на основе возможных значений переменных будет сделан вывод о недостижимости некоторой инструкции, то и при выполнении программы, эти инструкции не будут достигнуты.

```
if (…)
    x = 1;
else
    x = 3;

// x ∈ [1, 3]
if (x > 5) {
    ... // недостижимый код
}
```

Полурешётка анализа интервалов значений описывается следующим образом:

- Элементами полурешётки для одной целочисленной переменной являются интервалы [a, b]. Полурешётка для точки программы – это декартово произведение полурешёток для переменных.
- Частичный порядок определяется отношением включения $\subseteq$.
- Наименьшая верхняя граница для нескольких интервалов вычисляется при помощи объединения $\cup$.

Передаточные функции для арифметических операций основываются на интервальной арифметике [5], например (табл. 1):

Табл. 1. Передаточные функции для переменных `x` *и* `y` *с интервалами значений* `[a, b]` *и* `[c, d]` *и константы* `k`*.*

Table 1. Transfer functions for variables x and y with value intervals [a, b] and [c, d] and for a constant k

Выражение	Вычисление интервала значений для результата выражения
x ← k	`x: [k, k]`
`x + y`	`[a, b] + [c, d] = [a + c, b + d]`
`x - y`	`[a, b] - [c, d] = [a - c, b - d]`
`x * y`	`[a, b] * [c, d] = [min(ac, ad, bc, bd),` `max(ac, ad, bc, bd)]`

Сравнение	Вычисление интервалов значений для `x` и `y` в результате сравнения
`assume x = y`	`x: [a, b] ∩ [c, d]` `y: [a, b] ∩ [c, d]`
`assume x ≤ y`	`x: [a, b] ∩ [-inf, d]` `y: [a, +inf] ∩ [c, d]`
`assume x > y`	`x: [a, b] ∩ [c + 1, +inf]` `y: [-inf, b - 1] ∩ [c, d]`

`assume x ≠ k`	`x: [a + 1, b], если k = a`
	`x: [a, b + 1], если k = b`
	`x: [a, b],     если k ∈ [a + 1, b - 1]`

Отметим, что полурешётка интервалов значений имеет бесконечную высоту. Это означает, что в обычном виде анализ потока данных не будет сходиться. Чтобы анализ сошёлся, мы применяем технику *widening* [6]: внутри компонент сильной связности графа потока управления на очередной итерации анализа при перевычислении интервала в точке программы:

- при расширении границы интервала, она сразу заменяется на бесконечность,
- левая граница интервала не смещается вправо, а правая - не смещается влево.

Подсчитанные значения уточняются в режиме *narrowing*: передаточные функции работают, как было описано выше.

На инструкциях сравнения может быть вычислен пустой интервал значений. Пустой интервал свидетельствует о том, что сравнение инвариантно: результат сравнения всегда ложный.

```
// x ∈ [0, 1]
// y ∈ [3, 7]
if (x == y) { // assume x = y
    ... // [0, 1] ∩ [3, 7] = Ø
    ... // недостижимый код
}
```

Недостижимому коду соответствует значение ⊥ из полурешётки для всех переменных.

2.2 Анализ выколотой точки

Анализ интервалов значений не учитывает выколотые точки интервалов, что приводит к недостаточной точности: анализ не обнаруживает многие инвариантные сравнения. Например:

```
if (x != 0) { // assume x ≠ 0
    // ???
    if (x == 0) {
        // x ∈ [0, 1]
    }
}
```

Для таких ситуаций можно описать анализ, который бы вычислял в каждой точке программы для каждой переменной множество констант, которым она в этой точке не равна.

Мы реализовали простой анализ, вычисляющий в каждой точке программы, какие переменные не равны нулю. Константа ноль является наиболее популярной, и работа только с ней тем не менее позволяет обнаруживать много ситуаций недостижимого кода (см. гл. 3).

Элементом полурешётки для одной переменной является утверждение о том, что переменная не равна нулю.

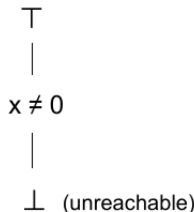


Рис. 2. Частичный порядок полурешётки анализа выколотой точки

Fig. 2. Partial order for semilattice of missing value analysis

Передаточные функции описаны следующим образом (табл. 2):

Табл 2. Передаточные функции для переменных x и y и константы k

Table 2. Transfer functions for variables x and y and for a constant k

Инструкция	Значение потока данных
x = 0	⊤
x = k	x ≠ 0 , если k ≠ 0
y = *x	x ≠ 0
assume x ≠ 0	x ≠ 0
assume x = 0	⊥ , если x ≠ 0

Недостижимый код по приведённой схеме будет обнаружен в примере, который приводился выше:

```
if (x != 0) { // assume x ≠ 0
    // x != 0
    if (x == 0) {
        // недостижимый код
    }
}
```

2.3 Предикатный анализ

В описанных выше анализах не учитываются сравнения тех переменных, которым сопоставлены значения ⊤, например:

```
void foo(int a, int b) {
    if (a > b) {
        if (a <= b) {
            // недостижимый код
            ...
        }
    }
}
```

Учесть подобные ситуации можно при помощи вычисления необходимых условий достижения точек программы.

Для этого был реализован предикатный анализ. Этот анализ производится над программой, переведённой в SSA-форму. Вычисляемые необходимые условия достижения точек программы – это конъюнкции, состоящие из предикатов инструкций сравнения.

```
void foo(int a1, int b1) {
    if (a1 > b1) {
        // a1 > b1
        if (b1 != 3) {
            // a1 > b1 && b1 != 3
        }
        // a1 > b1
    } else {
        // a1 <= b1
    }
}
```

Рис. 3. Пример вычисляемых необходимых условий достижения

Fig. 3. Example of computed necessary conditions

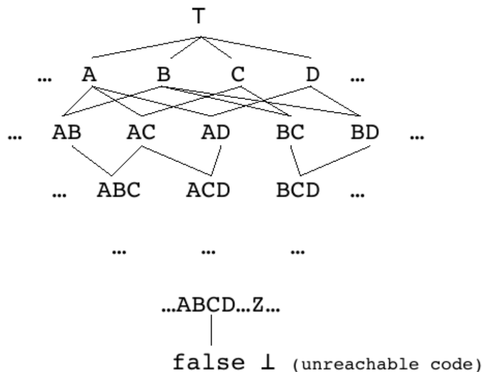


Рис 4. Частичный порядок полурешётки конъюнкций

Fig. 4. Partial order for semilattice of conjunctions

Полурешётка предикатного анализа описывается следующим образом:

- Элементами полурешётки для точки программы являются конъюнкции предикатов, содержащихся в программе (предикаты обозн. `A`, `B`, `C`, …).
- Предикат – атомарное сравнение двух операндов (операнды – SSA-определения и константы).
- Каждая конъюнкция рассматривается как множество предикатов.
- Частичный порядок конъюнкций определяется отношением вложенности этих множеств ⊆.
- Наименьшая верхняя граница для набора конъюнкций вычисляется при помощи их пересечения ∩.

Передаточные функции анализа предикатов описываются только инструкций сравнения:

Для конъюнкции `IN[instr]` на входе в инструкцию `instr` вычисляется конъюнкция `OUT[instr]` на выходе:

Инструкция	Значение потока данных
`instr: assume A`	`OUT[instr] ← A && IN[instr]`

При этом `OUT[instr]` сокращается в `false`, если `IN[instr]` включает в себя противоположный предикат `!A` (рис. 5).

```
void foo(int a1, int b1) {
    if (a1 > b1) {      // assume A
        ...
        if (a1 <= b1) { // assume !A
            // недостижимый код
        }
    }
}
```

Рис. 5. Пример обнаружения недостижимого кода

Fig. 5. Example of unreachable code detection

2.4 Нумерация значений

Нумерация значений – это анализ, при помощи которого переменные разбиваются на классы эквивалентности в каждой точке программы. Номерами значений обозначаются классы эквивалентности.

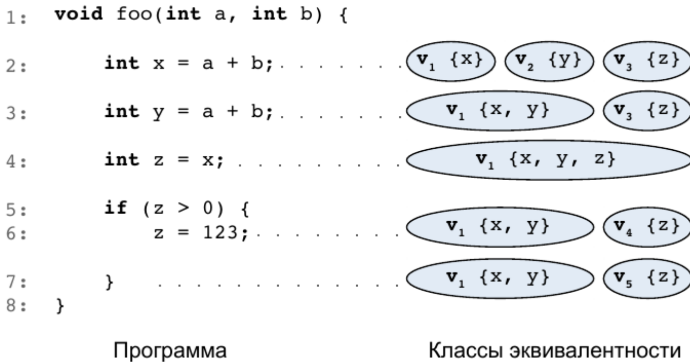


Рис. 6. Разбиение переменных на классы эквивалентности

Fig. 6. Variables partitions into equivalence classes

Это разбиение на классы эквивалентности применяется для того, чтобы анализ потока данных проводился не над переменными, а над их номерами значений. Таким образом, анализ будет устанавливать утверждения сразу для всего класса эквивалентности.

```
x = y;
if (x > 10) {
    ... // x > 10, y > 10
}
```

Рис. 7. Утверждение для класса эквивалентности

Fig. 7. Data-flow value for equivalence class

Вычисление номеров значений производится следующим образом.

- Программа переводится в SSA-форму. Задача сводится к тому, чтобы определить, какие SSA-определения равны между собой.
- Вначале всем SSA-определениям сопоставляются уникальные номера значений:

```
for each x = op(a, b) in Definitions
    vn[x] ← uniqueValue();
eval ← ∅
```

- Итеративно вычисляя хэш от операции и от номеров значений её операндов, мы можем объединять переменные в классы эквивалентности:

```
While (changes)
    For each x = op(a, b) in Definitions
        h ← hash(op, vn[a], vn[b]);
        If (eval[h] == null)
            eval[h] ← vn[x];
        Else
            vn[x] ← eval[h]
```

В приведённом алгоритме предполагается, что хэш-функция обрабатывает коллизии таким образом, что гарантируется, что она никогда не вернёт одно и то же значение для двух разных её аргументов.

3 Сравнение результатов анализов

Описанные анализы были запущены на проектах с открытым исходным кодом: Android-5.02 и Tizen-2.3. Каждый анализ выдал предупреждения о найденных инвариантных сравнениях в коде программ. На рис. 8 приведены графики, демонстрирующие количество предупреждений, полученных от каждого из анализов и их пересечение между собой.

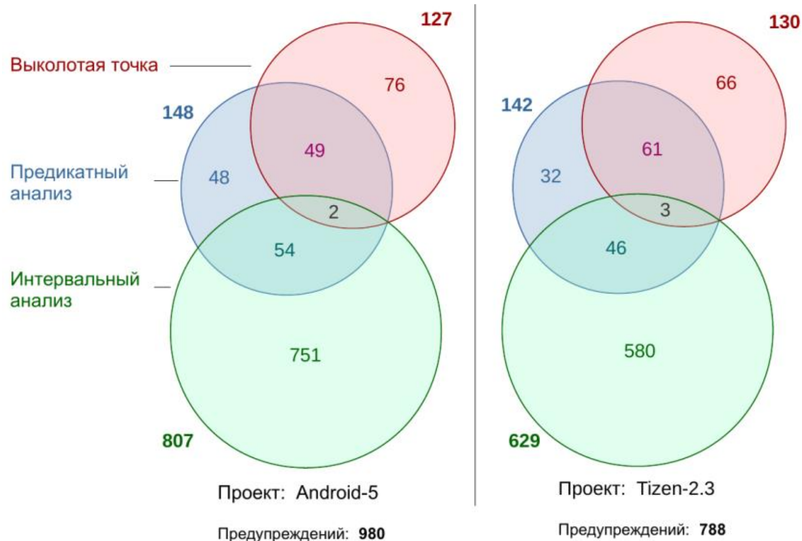


Рис 8. Пересечение предупреждений об инвариантных сравнениях

Fig. 8. Intersection of warnings about invariant comparisons

В графики не включены предупреждения, которые были автоматически отсеяны анализатором. Отсеивание производилось для многих случаев, когда предупреждение не означает ошибку в программе, и исходный код не будет исправлен программистом. Тем не менее, на текущий момент выдаваемых бесполезных предупреждений остаётся 45% от всех предупреждений, и их автоматическое отсеивание является частью нашей дальнейшей работы. Можно выделить типичные виды недостижимого кода, не являющиеся ошибками:

- Недостижимый код внутри раскрытого макроса
- Недостижимый код, возникший из-за подставленного параметра в шаблоне Си++.
- Недостижимый код, возникший из-за сравнения с константой, которую разработчик может при желании менять.
- Недостижимая метка default в инструкции switch.
- Недостижимый код из-за проверки “на всякий случай”, например:

```
free(p);
p = NULL;
goto failure;
...
return;
failure:
    if (p != NULL) {
        // недостижимый код "Won't fix"
        free(p);
        p = NULL;
    }
```

Приведённая в примере ситуация зачастую возникает в методах с большим количеством строк. Такие методы могут модифицироваться в будущем, и потому эта проверка все равно является полезной.

Как видно из графиков, большая часть предупреждений находится с помощью интервального анализа. Анализ предикатов позволяет найти больше ошибок, чем анализ выколотой точки. Но результаты при этом в большей степени пересекаются с результатами интервального анализа, что не удивительно, т. к. интервальный анализ не учитывает выколотые точки. Пересечение результатов интервального анализа и анализа выколотой точки происходит в случае, когда выколотая точка находится на границе интервала.

Заключение

В статье описано использование анализов недостижимого кода для задачи поиска ошибок в исходном коде программ. Описанные анализы позволили найти сотни ошибок для операционных систем Android 5.02 и Tizen 2.3.

Список литературы

[1]. Аветисян А. И., Бородин А. Е. Механизмы расширения системы статического анализа Svace детекторами новых видов уязвимостей и критических ошибок. Труды ИСП РАН, том 21, 2011, стр. 39-54.

[2]. Аветисян А. И., Белеванцев А. А., Бородин А. Е., Несов В. С. Использование статического анализа для поиска уязвимостей и критических ошибок в исходном коде программ. Труды ИСП РАН, том 21, 2011, стр. 23-38.

[3]. Ivannikov V. P., Belevantsev A. A., Borodin A. E. et al. Static analyzer Svace for finding defects in a source program code. Programming and Computer Software, vol. 40, no. 5, 2014, pp. 265–275. DOI: 10.1134/S0361768814050041

[4]. Бородин А. Е., Белеванцев А. А. Статический анализатор Svace как коллекция анализаторов разных уровней сложности. Труды ИСП РАН, том 27, вып. 6, 2015, стр. 111-134. DOI: 10.15514/ISPRAS-2015-27(6)-8.

[5]. Шокин Ю.И. Интервальный анализ. Новосибирск: Наука, 1981.

[6]. P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In Proc. Int. Workshop on Programming Language Implementation and Logic Programming, volume 631 of LNCS, pages 269–295. Springer-Verlag, 1992.

Using unreachable code analysis in static analysis tool for finding defects in source code

R.R. Mulyukov <eygz@ispras.ru>
A.E. Borodin <alexey.borodin@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. The goal of finding unreachable code is to report warnings about possible bugs in the source code and an increase of other analyses accuracy. The paper describes unreachable code classification and approaches for finding unreachable code in C/C++ programs. We described three data-flow analysis problems: value interval analysis, excluded value analysis, predicate analysis. Solutions for these problems are used to detect redundant expressions in conditional statements. We described common occurrences of useless warnings. The algorithms are implemented in the Svace tool that is developed by ISP RAS. The results are evaluated for open source projects Android-5.02 and Tizen-2.3. They represent the number of found warnings and its intersection.

Keywords: static analysis; unreachable code; data-flow analysis; Svace; defects in source code.

DOI: 10.15514/ISPRAS-2016-28(5)-9

For citation: R.R. Mulyukov, A.E. Borodin. Using unreachable code analysis in static analysis tool for finding defects in source code. Trudy ISP RAN/Proc. ISP RAS, 2016, vol. 28, issue 5, 2016, pp. 145-158 (in Russian). DOI: 10.15514/ISPRAS-2016-28(5)-9

References

[1]. A.I. Avetisjan, A.E. Borodin. [Mechanisms for extending the system of static analysis Svace by new types of detectors of vulnerabilities and critical errors]. Trudy ISP RAN/Proc. ISP RAS volume 21, 2011, pp. 39–54 (in Russian).

[2]. A.I. Avetisjan, A.A. Belevantsev, A.E. Borodin, V.S. Nesov. [Using static analysis for searching vulnerabilities and critical errors in the source code of programs]. Trudy ISP RAN/Proc. ISP RAS, volume 21, 2011, pp. 23–38 (in Russian).

[3]. Ivannikov V. P., Belevantsev A. A., Borodin A. E. et al. Static analyzer Svace for finding defects in a source program code. Programming and Computer Software. 2014. Vol. 40, no. 5. P. 265–275. 5. DOI: 10.1134/S0361768814050041

[4]. Borodin A., Belevancev A. [A Static Analysis Tool Svace as a Collection of Analyzers with Various Complexity Levels]. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015, pp. 111-134 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-8

[5]. Y.I. Shokin. Interval analysis. Novisibirsk – Science, 1981.

[6]. P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In Proc. Int. Workshop on Programming Language

Implementation and Logic Programming, volume 631 of LNCS, pages 269–295. Springer-Verlag, 1992.

Вычисление входных данных для достижения определенной функции в программе методом итеративного динамического анализа

[1] *А.Ю. Герасимов <agerasimov@ispras.ru>*
[2] *Л.В. Круглов <kruglov@ispras.ru>*
[1] *Институт системного программирования РАН,*
109004, Россия, г. Москва, ул. А. Солженицына, д. 25
[2] *Московский государственный университет имени М.В. Ломоносова,*
119991, Россия, Москва, Ленинские горы, д. 1

Аннотация. Динамическое символьное исполнение – это хорошо известная техника, применяемая для решения различных задач анализа программ: генерация входных данных для увеличения тестового покрытия программы, для эксплуатации уязвимостей и т.д. В то же время значительное падение производительности при динамическом символьном исполнении также является хорошо известной, в общем случае NP-сложной проблемой в связи с экспоненциальным взрывом количества анализируемых путей и решением задачи SAT/SMT при разрешении предиката пути. Применение метода грубой силы при попытке анализа всех достижимых путей в программе, как правило, не имеет смысла в условиях жесткого ограничения времени решения поставленных задач. Поэтому применяются различные методы и эвристики для увеличения производительности анализа и сокращения анализируемого пространства. Мы представляем подход совмещения статического анализа исполняемого кода, основанного на использовании библиотеки binutils, и метода динамического символьного исполнения, основанного на инструменте итеративного динамического анализа Avalanche, для направленной генерации входных данных программы с целью достижения заранее определенной функции в программе. На первом шаге предлагаемого подхода строится усеченный граф вызовов программы, который содержит только те функции, вызов которых в конечном счете приводит к вызову заранее определенной функции. Далее мы дополняем граф вызовов графом потока управления внутри функций, включенных в усеченный граф вызовов. С использованием усеченного графа потока управления программы, который содержит только вызовы и условные переходы, приводящие в конечном итоге к вызову заранее определенной функции, вычисляется метрика наиболее перспективного пути для проведения дальнейшего анализа. Предложенный подход позволил значительно (до двенадцати раз для некоторых реальных программ) сократить время достижения заранее определенной функции по сравнению с методом грубой силы.

Ключевые слова: статический анализ; динамический анализ; генерация входных данных.

DOI: 10.15514/ISPRAS-2016-28(5)-10

Для цитирования: А.Ю. Герасимов, Л.В. Круглов. Вычисление входных данных для достижения определенной функции в программе методом итеративного динамического анализа. Труды ИСП РАН, том 28, вып. 5, 2016, стр. 159-174. DOI: 10.15514/ISPRAS-2016-28(5)-10

1. Введение

В связи с активным развитием области разработки программного обеспечения, применяемого в таких критических областях жизни и деятельности человека, как автоматическое управление транспортными средствами и управление опасными производствами, медицине и военной технике, то есть в областях, где вмешательство оператора вычислительного устройства для исправления ошибочной ситуации минимально или сведено к нулю, — на первый план выходит задача обеспечения высокого качества программ с точки зрения отсутствия критических ошибок времени исполнения и уязвимостей, позволяющих злоумышленнику получить контроль над исполнением программы. Для обеспечения качества программного обеспечения могут применяться различные подходы, такие как тестирование [1] и верификация моделей [2, 3] — подходы, которые направлены на выяснение соответствия разработанной программы спецификации требований или модели, составленной на стадии проектирования. С другой стороны, данные подходы не всегда способны обеспечить проверку отсутствия ошибок или нежелательного поведения программы, которое не может быть заложено в функциональные требования к программе или в модель. Для решения данной задачи применяются методы анализа программ, направленные на автоматическое выявление нежелательного поведения или критических ошибок. При этом сами методы анализа программ либо недостаточно точны и выдают большое количество ложных предупреждений об ошибках (статический анализ), либо обладают высокой вычислительной сложностью, что не позволяет проводить полный анализ программы за приемлемое время (динамический анализ).

Нередко в процессе анализа программ может быть использовано несколько инструментов, проводящих разные виды анализа программ, а также может быть вовлечен аналитик, который использует инструменты анализа программ для обнаружения ошибок и уязвимостей в программе. Например, аналитику может быть интересно проанализировать программу на наличие возможности эксплуатации уязвимости, связанной с использованием форматной строки [4] или разработчику требуется подтвердить наличие дефекта в программе, найденного инструментами статического анализа исходного кода программы. В первом случае аналитик указывает функцию, достижимость которой

необходимо проверить и предоставить информацию о том, как входные данные программы преобразуются в аргументы функции, во втором случае информация о функции, в которой потенциально реализуется дефект или уязвимость, передается инструментом статического анализа программы инструменту динамического анализа программы в виде трассы с предусловиями реализации ошибочной ситуации. В обоих случаях возникает задача вычисления входных данных, при получении которых в программе происходит вызов определенной функции, а также выявления зависимости между входными данными программы и аргументами интересующей функции. В данной статье описывается подход, направленный на совмещение методов статического и динамического анализа программ с целью повышения производительности динамического анализа программ при решении задачи вычисления входных данных для достижения указанной функции программы.

Далее статья построена следующим образом. Во втором разделе производится обзор методов анализа программ и ограничений, которые им присущи, а также даётся краткий обзор применения результатов статического анализа для упрощения проведения динамического анализа программ. В третьем разделе подробно рассматривается подход к применению результатов статического анализа для повышения производительности динамического анализа программ, в четвертом разделе приводятся результаты экспериментов, подтверждающих практическую применимость предложенного подхода, в заключении рассматриваются возможные приложения для применения описанного подхода, а также направления дальнейших исследований.

2. Методы анализа программ и их ограничения

Методы анализа программ условно можно разбить на две группы: статические, то есть такие, при которых программа анализируется без запуска на выполнение, и динамические, при которых программа анализируется в процессе (online-анализ) или по результатам (offline-анализ) выполнения.

Статический анализ программ может выполняться как по исходному коду, так и по исполняемому коду программы. Методы статического анализа программ строят модель программы, являющейся абстракцией исходного или бинарного кода программы, и производят анализ с использованием данной модели. Как правило, методы статического анализа обладают возможностью масштабирования путём независимого анализа отдельных функциональных блоков в программе и составлению автоматических аннотаций для каждого блока по набору критериев, описывающих его поведение, но при этом характеризуются возможностью выдачи ложно-положительных предупреждений об ошибочной ситуации в связи с тем, что модель может недостаточно полно описывать поведение программы, а проверка ошибочной ситуации может проводиться не на полном пути выполнения от точки входа в программу до реализации ошибочной ситуации, а только на некотором

подпути в программе от точки инициализации ошибочной ситуации до точки реализации ошибки.

В свою очередь, методы динамического анализа программ обладают высокой точностью обнаружения ошибочных ситуаций и, как правило, возможностью их воспроизведения в связи с тем, что ошибка регистрируется в момент, когда программа уже выполнила недопустимую операцию, сохранена трасса выполнения от точки входа в программу и имеются точные входные данные, которые позволяют воспроизвести ошибочное поведение. С другой стороны, методы динамического анализа крайне плохо масштабируются. Это связанно с тем, что для проведения анализа программы по альтернативному пути выполнения необходимо тем или иным способом вычислить входные данные, например методами фаззинга [5] или методами символьного выполнения [6, 7, 8] с решением формулы ограничений пути, и произвести полный или частичный запуск программы на выполнение с использованием вычисленных входных данных.

Как отмечалось в работах [9, 6], при применении методов итеративного динамического анализа, построенных на принципе символьного выполнения программы, в процессе анализа программы достаточно быстро возникает проблема экспоненциального роста количества путей, которые необходимо проанализировать. Для каждого нового пути в программе необходимо построить и вычислить одну или несколько формул ограничений пути при помощи решателя, например [10, 11], что в свою очередь согласно [6, 12] в зависимости от реализации анализируемой программы может составить до 99% времени анализа, поскольку для решения этой задачи не известно полиномиальных алгоритмов [13, 14], то на каждом запуске программы необходимо решить в общем случае NP-сложную задачу [15] для решения формулы ограничений пути или определения несовместности ограничений пути. Ранее рассматривались подходы к увеличению производительности итеративного динамического анализа путем применения параллельных и распределенных вычислений [16] и кэширования результатов вычисления булевых формул [17], но в общем случае кардинально увеличить производительность итеративного динамического анализа программ при сохранении полноты анализа на данный момент не представляется возможным в связи с экспоненциальным ростом количества путей, которые необходимо проанализировать в программе.

В связи с выше сказанным, проведение полного анализа всех путей в программе за приемлемое время не представляется возможным и крайне важным становится решение задачи выбора для анализа наиболее перспективных путей в программе. Как правило, выбор пути для дальнейшего анализа программы происходит на основе некоторой эвристической оценки перспективности пути, в связи с этим возникает задача построения качественной метрики выбора следующего пути для анализа в зависимости от целей анализа. В настоящей статье предлагается подход к определению

лучшего пути до указанной тем или иным способом функции программы на очередном шаге итеративного динамического анализа на основе результатов предварительного статического анализа исполняемого кода программы.

2.1 Различные подходы к выбору пути для проведения динамического анализа

Предлагаемый нами подход реализован в рамках инструмента Avalanche [6] путем разработки дополнительного модуля вычисления метрики наилучшего пути. В работе, описывающей инструмент Avalanche, рассматривается подход к реализации итеративного динамического анализа с применением метрики перспективности пути по критерию максимально быстрого прироста покрытия базовых блоков программы. Если множество проанализированных базовых блоков программы на i-той итерации анализа — $\boldsymbol{BBSetAnalyzed_i}$, количество базовых блоков программы, которое будет проанализировано на пути $\boldsymbol{\pi_n}$ даст покрытие $\boldsymbol{BBSet_{\pi n}}$, а количество базовых блоков, которые будут проанализированы на пути $\boldsymbol{\pi_m}$, даст покрытие $\boldsymbol{BBSet_{\pi m}}$, то в первую очередь будет проанализирован путь, для которого мощность разности множества проанализированных базовых блоков на предыдущих итерациях анализа и множества базовых блоков пути будет больше или равна:

$$\pi_{i+1} = \begin{cases} \pi_n, |BBSetAnalyzed_i \setminus BBSet_{\pi n}| \geqslant |BBSetAnalyzed_i \setminus BBSet_{\pi m}| \\ \pi_m, |BBSetAnalyzed_i \setminus BBSet_{\pi n}| < |BBSetAnalyzed_i \setminus BBSet_{\pi m}| \end{cases}$$

В другой работе [18], посвященной инструменту автоматической генерации кода эксплуатации уязвимостей в программе, рассматриваются две эвристики выбора пути для поиска программных ошибок с целью их последующей эксплуатации. Эвристика **Уязвимый-путь-сначала (Buggy-Path-First)** предполагает поиск на пути выполнения операции с ошибкой на единицу при вычисления диапазона в процессе работы с буферами в памяти (off-by-one error), которая сама по себе может и не быть эксплуатируема, однако позволяет предположить, что программист недостаточно внимательно следит за границами выделенных буферов, что может привести к нахождению ошибки переполнения буфера далее на пути, которую можно будет проэксплуатировать. Эвристика **Разрежения циклов (Loop Exhaustion)** применяется в случае, если индуктивная переменная цикла зависит от входных данных. Тогда инструмент не пытается выполнить цикл на всём пространстве значений индуктивной переменной, а вычисляет максимально возможное значение переменной и анализирует выполнение цикла сначала на максимально возможном количестве итераций с учётом ограничений пути на значение индуктивной переменной в надежде, что вычисления, требующие большего количества итераций, приведут к ошибкам переполнения буфера с большей вероятностью.

3. Описание предлагаемого подхода

Итеративный динамический анализ осуществляет последовательный обход дерева путей программы, зависящих от условных переходов, и осуществляет поиск ошибок на каждом исполненном пути. Для этого в процессе исполнения программы производится отслеживание операций над входными данными и данными, являющимися результатами этих операций. Такие данные называются помеченными (tainted). Необходимо отслеживать все операции чтения из внешних источников и записи прочитанных значений в память и регистры, которые далее также необходимо считать источниками помеченных данных до тех пор, пока они не будут перезаписаны непомеченными значениями. По результатам запуска программы производится сбор трассы выполненных операций и по собранной трассе выполненных операций строится система ограничений, описывающая ограничения на значения переменных и регистров, содержащих помеченные данные. Так как каждый путь определяется набором направлений условных переходов, то в соответствующую путям систему уравнений также необходимо включать условия переходов, зависящие от помеченных данных.

Для получения на основе исходной системы уравнений новых систем, возможно соответствующих новым путям исполнения, достаточно инвертировать условия каждого условного перехода, зависящего от входных данных. При наличии в системе N условий переходов будет получено N новых систем: в i-ой системе будет инвертировано условие i-го условного перехода, условия до i-го будут оставлены без изменений, а условия после i-го отброшены. Решив полученные N систем с помощью решателя формул, можно построить входные данные в случае успешной разрешимости формулы, которые соответствуют новым путям в исследуемой программе.

На каждом шаге итеративного динамического анализа исследуется один путь исполнения, строится описывающая его система уравнений и вычисляются новые наборы входных данных, соответствующие новым путям исполнения. Выбор единственного пути для анализа на следующей итерации осуществляется на основе вычисления метрик (эвристических оценок) и происходит при запуске программы на новых входных данных, вычисленных с учётом метрики наилучшего пути. Пути, отброшенные на текущем шаге анализа, сохраняются для анализа на последующих шагах, когда метрика перспективности отброшенного пути превысит метрику перспективности оставшихся путей на очередном шаге анализа. Это позволяет проанализировать программу на всех достижимых путях с помощью итеративного динамического анализа, но просматривать в первую очередь наиболее перспективные пути, получая различные наборы входных данных, соответствующие различным путям, на которых происходит вызов заданной функции.

В настоящей работе мы предлагаем проводить предварительный статический анализ исполняемого кода программы до запуска итеративного динамического

анализа. В результате проведения статического анализа программы строится граф вызовов и граф условных переходов, которые позволяют вычислить множество путей достижения указанной функции, наиболее интересной с точки зрения целей анализа. Итеративный динамический анализ в процессе построения формулы ограничений пути использует информацию, извлеченную из графа вызовов и графа переходов, полученных в результате статического анализа, и формирует набор утверждений для вычисления входных данных программы таким образом, чтобы на каждой следующей итерации анализа максимально быстро приближаться к указанной функции вплоть до достижения точки её вызова на очередном шаге анализа.

3.1 Построение путей до функции

Под графом вызовов будем понимать ориентированный циклический граф, узлы которого представляют собой различные функции исследуемой программы, а ребра — вызовы функций. Минимальное расстояние между узлами в графе вызовов определим как минимальное количество ребер на пути между данными узлами.

Для вычисления предлагаемой метрики достаточно иметь неполный граф вызовов — граф, содержащий только те узлы, из которых достижима заданная функция. Такой граф может быть эффективно построен обратным проходом, начиная с указанной функции. Сначала необходимо найти все точки вызова выбранной функции, соответствующие некоторым вызывающим функциям. Затем для каждой вызывающей функции необходимо найти все точки её вызова и соответствующие им вызывающие функции. И так до тех пор, пока не будет достигнута некоторая конечная функция или набор функций, которые явно не вызываются ни из каких других функций программы. Это может быть как точка входа в программу, так и функция, вызываемая через указатель (виртуальный вызов), или недостижимая функция. Далее под графом путей до выбранной функции будем понимать построенный таким образом неполный граф вызовов программы.

Расширим граф вызовов до графа переходов: дополним каждый узел в графе вызовов (функцию) информацией о путях исполнения внутри функции. Для этого требуется построить неполный граф потока управления каждой функции, состоящий из базовых блоков, лежащих на путях от входа в функцию до точек вызова других функций из неполного графа вызовов, ведущих к указанной функции. Определим расстояние в графе переходов аналогично — как количество ребер между узлами в расширенном графе переходов.

Вычисление расстояния от некоторой вершины графа переходов до выбранной функции удобно проводить для графа, в котором все ребра заменены на обратные и имеют единичный вес, с помощью алгоритма поиска кратчайшего расстояния от указанной вершины до всех остальных, например с помощью алгоритма Дейкстры.

Описанные графы вызовов и переходов строятся по исполняемому коду программы до начала итеративного динамического анализа, так как в этом случае будут учтены оптимизации компилятора, и он будет точно соответствовать путям исполнения исследуемой программы. При реализации предложенного подхода построение графов производится на основе анализа ассемблерного кода, полученного с помощью утилиты objdump пакета инструментов для работы с кодом объектных файлов binutils [19].

3.2 Метрика выбора наиболее перспективного пути

В качестве метрики для эффективного достижения указанной функции в анализируемой программе предлагается использовать расстояния в графе вызовов и условных переходов от точки на пути исполнения до точки входа в интересующую функцию. Минимальное расстояние будет соответствовать лучшему пути для анализа на следующей итерации.

Проиллюстрируем работу алгоритма выбора следующего пути для анализа на примере (Рис. 1). Допустим, что в процессе выполнения очередной итерации анализа (путь выполнения отмечен белыми узлами графа переходов) обнаружено несколько возможных путей для дальнейшего анализа, которые можно получить путём инвертирования условий в условных переходах 1, 2, 3 и 4.

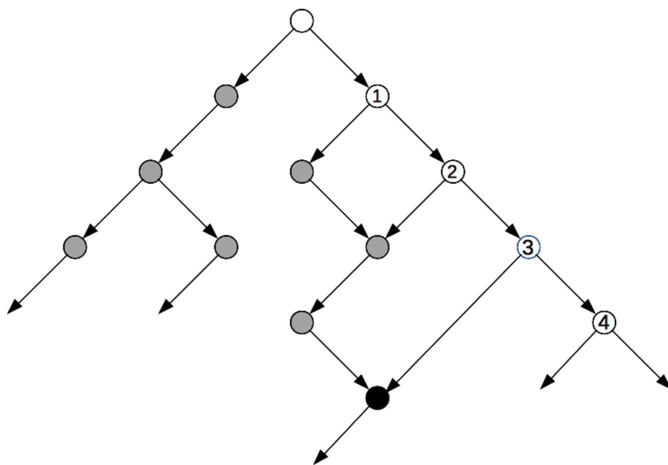


Рис. 1. Вычисление метрики выбора наилучшего пути для анализа

Fig. 1. Metric calculation for best path selection during analysis

Для каждого из возможных путей выполнения проводится вычисление входных данных и запускается легковесный проход, который отслеживает только посещения базовых блоков, присутствующих на пути выполнения. Для каждого базового блока по графу потока управления полученному на этапе

статического анализа исполняемого файла вычисляется расстояние до указанной функции. Тот путь, на котором вычисленное расстояние будет наименьшим, выбирается как наиболее перспективный для анализа на следующей итерации. Успешное достижение указанной функции фиксируется, когда значение метрики расстояния становится равным нулю. В представленном примере:

- при инвертировании условия в узле 1 до указанной функции необходимо будет пройти 3 условных перехода или вызова (путь 1). Значение метрики равно 3;
- при инвертировании условия в узле 2 до указанной функции необходимо будет пройти 2 условных перехода или вызова (путь 2). Значение метрики равно 2;
- при инвертировании условия в узле 3 до указанной функции останется 0 условных переходов или вызовов (путь 3). Значение метрики равно 0;
- при инвертировании условия в узле 4 мы никогда не достигнем указанной функции. Анализ для данного пути проводится в последнюю очередь, так как в процессе статического анализа путь, ведущий к указанной функции, не найден, но может существовать путь, зависящий от состояния времени выполнения программы, например при наличии перехода по вычисляемому адресу.

Соответственно, для последующего анализа будет выбран путь 3, как путь с наименьшей метрикой приближения к указанной функции.

4. Результаты экспериментов

Полученное решение было протестировано на ряде проектов с открытым исходным кодом, в которые специально был привнесен дефект разыменования нулевого указателя:

- `qtdump`. `libquicktime` – библиотека чтения и записи файлов в форматах QuickTime/AVI/MP4; утилита `qtdump` отображает разобранное содержимое файла;
- `cjpeg`. `libjpeg7` – библиотека для обработки файлов в формате JPEG; утилита `cjpeg` конвертирует файлы в формат JPEG;
- `swfdump`. `swftools` – пакет программ для работы с SWF-файлами; утилита `swfdump` отображает подробную информацию о файле и дизассемблированный код;
- `gif2rgb`. `giflib` – библиотека для чтения и записи файлов в формате GIF; утилита `gif2rgb` конвертирует GIF-изображения в 24-битные RGB изображения;

- `tiffdump`. `libtiff` – библиотека для обработки файлов в формате TIFF; утилита `tiffdump` выводит подробную информацию о TIFF-файле;
- `mpeg3cat`. `Libmpeg3` — библиотека для редактирования MPEG-файлов. `mpeg3cat` — утилита для конкатенации и разделения MPEG-потоков;
- `xmllint`. `libxml2` — библиотека для работы с файлами в формате XML. Утилита `xmllint` позволяет производить разбор XML-файлов.

В таблице 1 представлены результаты измерения времени достижения одного и того же дефекта и количестве потребовавшихся для этого итераций. Сравниваются исходная метрика покрытия базовых блоков и реализованная метрика минимального расстояния до указанной функции, использующая граф вызовов и использующая граф переходов с упрощенным графом потока управления для каждого узла.

Табл. 1. Время и количество итераций анализа до обнаружения дефекта

Table 1. Time and iterations count before reaching defect

Проект	Avalanche		с графом вызовов		с графом путей	
	Итераций	Время	Итераций	Время	Итераций	Время
qtdump	73	1:56:07	60	17:15	60	17:26
	38	19:06	33	2:47	33	2:26
cjpeg	116	6:12	98	4:48	93	4:43
	30	6:08	14	0:33	5	0:27
swfdump	5	0:10	3	0:10	3	0:10
	12	0:21	6	0:18	4	0:13
gif2rgb	10	1:13	432	7:34:13	5	0:24
tiffdump	16	0:22	3	0:12	3	0:14
mpeg3cat	24	4:32:24	421	53:01	15	8:28
xmllint	Не найдено за 10 часов		Не найдено за 10 часов		325	46:04

По результатам экспериментов можно сделать следующие выводы:

- количество итераций, потребовавшихся для достижения дефекта, в общем случае незначительно меньше для новой метрики;
- анализ проекта `gif2rgb` показал, что новая метрика, использующая только граф вызовов, в некоторых случаях значительно замедляет достижение дефекта — более чем в 420 раз для данного эксперимента. Это связано с тем, что, с одной стороны, содержащая дефект функция расположена неглубоко в стеке вызовов и, с другой стороны, путь от точки входа в функцию до дефекта содержит большое количество условных переходов. В этом случае при использовании графа вызовов большое количество путей внутри функции будет иметь одинаковую эвристическую оценку вне зависимости от близости дефекта, что не позволяет выбирать для анализа пути, приближающиеся к дефекту. Однако учёт условных переходов при использовании метрики с графом переходов позволяет получить выигрыш по времени в 3 раза для того же дефекта по сравнению с исходной метрикой пути, рассчитанной по приросту покрытия базовых блоков.

Реализованная метрика позволяет получить значительный выигрыш по времени нахождения дефекта по сравнению с исходным инструментом Avalanche в случае, если известна функция, в которой наиболее вероятно обнаружение дефекта. Наилучшие результаты были получены для проектов `qtdump` (ускорение в 6 раз), `cjpeg` (ускорение в 12 раз).

5. Заключение

Применение статического анализа исполняемого или исходного кода программ, как предварительного шага перед проведением динамического анализа программ не является новым подходом. В работах [20, 21] рассматривается применение предварительного статического анализа графа вызовов программы для обнаружения точек входа в программах для ОС Android и построения путей для последующего проведения динамического символьного выполнения программы с целью обнаружения ошибок времени исполнения. Особенностью программ для ОС Android является наличие нескольких точек входа в потоке управления программы, связанное с моделью обработки сообщений операционной системы как от элементов управления пользовательского интерфейса, так и от других системных событий, по этому для проведения динамического анализа крайне важно определить возможные точки входа в поток управления программы.

Представленный в настоящей статье подход к решению задачи повышения производительности итеративного динамического анализа может быть применен для увеличения производительности решения целого класса задач в области обеспечения качества программного обеспечения. Дополнительно, представленный подход является частью решения задачи подтверждения

ошибок и уязвимостей (дефектов) в программном обеспечении, обнаруженных методами статического анализа исходного кода.

Дельнейшие исследования в области совмещения статического и динамического анализа программ могут быть направлены на решение задачи подтверждения дефектов, найденных методами статического анализа программ, генерации кода эксплуатации подтвержденных дефектов и улучшение производительности алгоритмов генерации данных для фаззинга.

Список литературы

[1]. Myers G. J., Badgett T., Sandler C. The Art of Software Testing. Third Edition. John Wiley & Sons, Inc., Hoboken, New Jersey, 2012, 240 p.

[2]. Ю.Г. Карпов. MODEL CHECKING. Верификация параллельных и распределенных систем. СПб:БХВ-Петербург. 2010

[3]. Кларк Э.М., Грамберг О., Пелед Д. Верификация моделей программ: Model Checking, М.:МЦНМО, 2002

[4]. Kyung-Suk Lhee, S.J. Chapin. Buffer Overflow and Format String Overflow Vulnerabilities. Software-Practice & Experience — Special Issue: Security Software, Volume 33 Issue 5, 25 April 2003, pp. 423-460

[5]. Ari Takanen, Jared D. Demott, Charles Miller. Fuzzing for Software Security Testing and Quality Assurance. Artech House, 2008

[6]. И.К. Исаев, Д.В. Сидоров. Применение динамического анализа для генерации входных данных, демонстрирующих критические ошибки и уязвимости в программах. Программирование №4, 2010 г.

[7]. Cadar C., Dunbar D., Engler. D. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008), December 8-10, 2008, San Diego, CA, USA

[8]. Chipounov V., Kuznetsov V., Candea G. The S2E Platform: Design, Implementation, and Applications. ACM Transactions on Computer Systems (TOCS) Special issue: Best papers of ASPLOS, February 2012.

[9]. В.В. Каушан, Ю.В. Маркин, В.А. Падарян, А.Ю. Тихонов. Методы поиска ошибок в бинарном коде. Препринты Института системного программирования РАН, препринт 24, 2013 г.

[10]. L. de Moura, N. Bjørner. Z3: an Efficient SMT Solver. Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008

[11]. Ganesh V.. Decision Procedures for Bit-Vectors, Arrays and Integers. (PhD. Thesis) Computer Science Department, Stanford University, Stanford, CA, U.S., Sept 2007

[12]. Исаев И.К., Сидоров Д.В., Герасимов А.Ю., Ермаков М.К. Avalanche: Применение динамического анализа для автоматического обнаружения ошибок в программах использующих сетевые сокеты. Труды ИСП РАН, том 21, 2011 г., стр. 55-70

[13]. I. Johnson. Formal Verification with SMT Solvers: Why and How. ACL2 Theorem Proving Seminar at the University of Texas, Autin, 2009

[14]. Новикова Н.М. Основы оптимизации. Москва. 1998. 17–22 с.

[15]. S.A. Cook. The complexity of theorem-proving procedures. Proceedings of the third annual ACM symposium on Theory of computing, New York, USA, NY,1971, pp 151-158

[16]. М.К. Ермаков, А.Ю. Герасимов. Avalanche: применение параллельного и распределенного динамического анализа программ для ускорения поиска дефектов и уязвимостей. Труды ИСП РАН, том 25, 2013 г., стр. 29-38. DOI: 10.15514/ISPRAS-2013-25-2
[17]. С.П. Вартанов, Д.В. Сидоров. Оптимизация задачи проверки выполнимости булевских ограничений при помощи кэширования промежуточных результатов. Труды ИСП РАН, том 22, 2012 г., стр. 281-292. DOI: 10.15514/ISPRAS-2012-22-16
[18]. Thanassis Agerinos, Sang Kil Cha, Brent Lim Tze Hao, David Broomley. AEG: Automatic Exploit Generation. Proceedings of the Network and Distributed Security Symposyum, Carnegie Mellon University, 2011
[19]. GNU Binutils [HTML] (http://www.gnu.org/software/binutils/)
[20]. Schütte J., Fedler R., Titze D. ConDroid: Targeted Dynamic Analysis of Android Applications. Advanced Information Networking and Applications (AINA), IEEE, Gwangui, 2105, DOI:10.1109/AINA.2015.238
[21]. Wong M., Lie D. IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware. In Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS), Feb 2016.

Input data generation for reaching specific function in program by iterative dynamic analysis

[1] *A.Y. Gerasimov <agerasimov@ispras.ru>*
[2] *L.V. Kruglov <kruglov@ispras.ru>*
[1] *Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.*
[2] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia.*

Abstract. Dynamic symbolic execution is a well-known technique used for different tasks of program analysis: input generation for increasing test coverage for program, inputs of death generation, exploit generation and etc. But huge time costs of program analysis during dynamic symbolic execution for any real-life program is a well-known problem caused by path explosion and necessity of path constraint solving for every path with different SAT/SMT techniques which is a NP-complete task in general case. Brute force analysis of every path in program has limited practical sense for time limited analysis; instead different techniques and heuristics are used to improve analysis performance and reduce space of analysis for specific needs of analyst or while solving specific problem under analysis. We present our approach which combines static analysis of program binary code based on binutils library with dynamic symbolic execution tool based on Avalanche – an iterative dynamic analysis tool to perform targeted input data generation for reaching specific function in the program. As the first step of our algorithm we extract reduced program call graph which contains only calls to functions which ends with the function of interest, then we amplify this call graph with control flow graph inside of functions included into reduced call graph. Using the reduced control-flow graph of program which contain only calls and conditional jumps directions which lead to the function of interest we built the metric of best next analysis direction. This approach allows us to significantly (up to twelve times for some real world programs) reduce the time of reaching function of interest comparatively to brute force program paths analysis with inversion of every conditional jump at the execution path dependent on tainted data.

Keywords: dynamic program analysis; static program analysis; directed analysis; input data generation.

DOI: 10.15514/ISPRAS-2016-28(5)-10

For citation: A.Y. Gerasimov, L.V. Kruglov. Input data generation for reaching specific function in program by iterative dynamic analysis method. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 5, 2016, pp. 159-174 (in Russian). DOI: 10.15514/ISPRAS-2016-28(5)-10

References

[1]. Myers G. J., Badgett T., Sandler C. The Art of Software Testing. Third Edition. John Wiley & Sons, Inc., Hoboken, New Jersey, 2012, 240 p.

[2]. Ju.G. Karpov. MODEL CHECKING. Verification of parralel and distributed systems. SPb:BHV-Peterburg. 2010 (in Russian)

[3]. Klark Je.M., Gramberg O., Peled D. Verification of program models: Model Checking, M.:MCNMO, 2002 (in Russian)

[4]. Kyung-Suk Lhee, S.J. Chapin. Buffer Overflow and Format String Overflow Vulnerabilities. Software-Practice & Experience — Special Issue: Security Software, Volume 33 Issue 5, 25 April 2003, pp. 423-460

[5]. Ari Takanen, Jared D. Demott, Charles Miller. Fuzzing for Software Security Testing and Quality Assurance. Artech House, 2008

[6]. I.K. Isaev, D.V. Sidorov. Application of dynamic analysis for generating input data exposing critical errors and vulnerabilities in programs. Programmirovanie №4, 2010 g. (in Russian)

[7]. Cadar C., Dunbar D., Engler. D. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008), December 8-10, 2008, San Diego, CA, USA

[8]. Chipounov V., Kuznetsov V., Candea G. The S2E Platform: Design, Implementation, and Applications. ACM Transactions on Computer Systems (TOCS) Special issue: Best papers of ASPLOS, February 2012.

[9]. V.V. Kaushan, Ju.V. Markin, V.A. Padarjan, A.Ju. Tihonov. Methods of finding errors in binary code. ISP RAS preprints, Preprint 24, 2013. (in Russian)

[10]. L. de Moura, N. Bjørner. Z3: an Efficient SMT Solver. Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008

[11]. Ganesh V. Decision Procedures for Bit-Vectors, Arrays and Integers. (PhD. Thesis) Computer Science Department, Stanford University, Stanford, CA, U.S., Sept 2007

[12]. Isaev I.K., Sidorov D.V., Gerasimov A.Ju., Ermakov M.K. Avalanche: application of dynamic analysis for automatic error detection in programs using network sockets. Trudy ISP RAN/Proc. ISP RAS, vol 21, 2011. (in Russian)

[13]. I. Johnson. Formal Verification with SMT Solvers: Why and How. ACL2 Theorem Proving Seminar at the University of Texas, Autin, 2009

[14]. Novikova N.M. Optimization basics. Moskva. 1998. pp. 17–22. (in Russian)

[15]. S.A. Cook. The complexity of theorem-proving procedures. Proceedings of the third annual ACM symposium on Theory of computing, New York, USA, NY,1971, pp 151-158

[16]. M.K. Ermakov, A.Y. Gerasimov. [Avalanche: adaptation of parallel and distributed computing for dynamic analysis to improve performance of defect detection]. Trudy ISP RAN/Proc. ISP RAS, vol. 25, 2013, pp. 29-38 (in Russian).

[17]. S.P. Vartanov, D.V. Sidorov. [Optimization of Boolean satisfiability solver by caching intermediate results]. Trudy ISP RAN/Proc. ISP RAS, vol. 22, 2012, pp. 281-292 (in Russian).

[18]. Thanassis Agerinos, Sang Kil Cha, Brent Lim Tze Hao, David Broomley. AEG: Automatic Exploit Generation. Proceedings of the Network and Distributed Security Symposyum, Carnegie Mellon University, 2011

[19]. GNU Binutils [HTML] (http://www.gnu.org/software/binutils/)

[20]. Schütte J., Fedler R., Titze D. ConDroid: Targeted Dynamic Analysis of Android Applications. Advanced Information Networking and Applications (AINA), IEEE, Gwangui, 2105, DOI:10.1109/AINA.2015.238

[21]. Wong M., Lie D. IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware. In Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS), Feb 2016.

Ускорение оптимизации программ во время связывания

К.Ю. Долгорукова <unerkannt@ispras.ru>
С.В. Аришин <arishin@phystech.edu>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. В данной статье речь идет о двух методах ускорения процесса сборки программы: распараллеливании межпроцедурной оптимизации и о легковесном методе проведения оптимизаций. Ускорение в первом случае достигается за счет горизонтального масштабирования системы оптимизации времени связывания. Во втором случае метод представляет собой работу исключительно на аннотациях, что позволяет работать с минимально необходимой информацией вместо кода всей программы целиком. Проблема горизонтальной масштабируемости системы всегда связана с необходимостью разделения большой задачи на несколько подзадач, которые могут быть выполнены независимо и параллельно. Масштабирование оптимизаций времени связывания – непростая задача, так как оптимизирующие преобразования работают последовательно на всем промежуточном представлении программы, и результат их работы зависит от предыдущих преобразований. Для оптимизации на этапе связывания необходимо разделить промежуточное представление компилируемой программы на участки, минимизируя зависимости между ними, и оптимизировать эти участки отдельно. Для анализа программы используется граф вызовов. Таким образом, задача сводится к тому, чтобы разделить граф вызовов на несколько слабо связанных друг между другом компонент. Данная задача относится к одной из сложных комбинаторных проблем, и нахождение оптимального решения – NP-полная задача. Тем не менее, качество работы алгоритма зависит от свойств графа, поэтому целесообразно исследовать свойства графа вызовов в контексте оптимизаций времени связывания и подобрать к нему приемлемый алгоритм, проверив работу на реальных программах. Основная задача данного исследования – найти легковесный и эффективный метод разбиения структуры программы таким образом, чтобы как можно меньше ухудшить производительность собираемых программ при независимой оптимизации участков кода. В статье представлен новый метод разбиения графа вызовов программ, проведено его сравнение с некоторыми другими существующими методами для графов вызовов тестовых программ. Также описана реализация предложенного метода в системе LLVM, представлены результаты сравнения производительности программ, собранных в один поток и в несколько потоков. Запуск на 4-х потоках показал ускорение процесса сборки в среднем на 31%, тогда как производительность по сравнению с собранными в один поток программами упала в среднем на 3%. Для легковесного метода оптимизаций описана реализация

преобразования удаления мертвого кода. Также приведены результаты тестирования в совокупности с ленивой загрузкой кода.

Ключевые слова: компиляторы; оптимизация времени связывания; масштабирование; разбиение графов.

DOI: 10.15514/ISPRAS-2016-28(5)-11

Для цитирования: К.Ю. Долгорукова, С.В. Аришин. Ускорение оптимизации программ во время связывания. Труды ИСП РАН, том 28, вып. 5, 2016, стр. 175-198. DOI: 10.15514/ISPRAS-2016-28(5)-11

1. Введение

Оптимизации времени связывания зарекомендовали себя как эффективный инструмент оптимизации программ [1]. Проводимые на этапе компоновки объектного кода с внедренным промежуточным представлением программ, они способны работать над всем кодом целиком, эффективно оптимизируя межпроцедурные зависимости. Также полное представление о коде дает возможность проводить более агрессивные локальные оптимизации и оптимизации с использованием профиля.

Сложность подхода оптимизаций времени связывания заключается в том же, что и его сила: весь код программы находится в памяти. Этот факт накладывает дополнительные требования к ресурсам, в особенности, для крупных программ. К примеру, сборка состоящего из 36,5 тысяч исходных C/C++ файлов браузера Firefox на компиляторах GCC и LLVM с оптимизацией времени связывания требует от 6 до 34 Гб ОЗУ в зависимости от настроек, и работает от 11 до 26 минут на x86-64 [2]. Для офисного текстового редактора LibreOffice, состоящего почти из 20 тысяч C/C++ файлов, эти числа будут иметь значения 8-14 Гб и 61-68 минут соответственно [3].

Данная статья является частью исследования, посвященной разработке масштабируемой системы оптимизации времени связывания. Полученные ранее результаты, а также подробное описание системы можно найти в [4] и [5]. Масштабируемость системы подразумевает как возможность работать на устройствах с ограниченными ресурсами, так и на высокопроизводительных машинах. И, если в ранних работах мы вели речь о сборке программ на устройствах с низкой производительностью, то в данной работе упор будет сделан на масштабировании под устройства с несколькими ядрами. Как было показано в [4], традиционный путь масштабирования подразумевает наличие трёх стадий при сборке программы из исходного кода: генерацию промежуточного представления, межпроцедурный анализ и генерацию машинного кода. Первая и последняя стадии теоретически могут проводиться параллельно. На практике же возможность распараллеленной оптимизации во время генерации машинного кода спорна, и применяется далеко не во всех

оптимизирующих компоновщиках. В частности, в системе LLVM ранее такой возможности не было.

Статья построена следующим образом: в разделе 2 речь пойдет о разбиении графа вызовов для проведения межпроцедурной оптимизации в несколько потоков. В подразделе 2.1 приведен краткий обзор существующих методов разбиения графа. В 2.2 приводятся оценки разбиения, а также обсуждается применимость алгоритмов к графам вызовов программ. В 2.3 описывается предлагаемый для разбиения алгоритм, а в 2.4 приводится его сравнение с некоторыми другими алгоритмами по предложенным в 2.2 метрикам. В разделе 3 описываются легковесные межпроцедурные оптимизирующие преобразования, поясняется их место в разрабатываемой системе оптимизации времени связывания. В 3.1 описывается общий метод работы таких преобразований. В 3.2 проводится краткий обзор существующих оптимизаций времени связывания и обсуждается применимость к ним предложенного метода. Раздел 4 посвящен реализации предлагаемых методов в системе LLVM. Раздел 5 содержит результаты тестирования. Заключение завершает статью выводами и планами дальнейших работ.

2. Горизонтальное масштабирование системы оптимизации времени связывания

Как было упомянуто в [4], этап анализа не может быть распараллелен, но, как правило, распараллеливается этап генерации машинного кода. Для этого на этапе анализа производится такое разбиение промежуточного представления программы на части, чтобы можно было проводить независимую оптимизацию этих частей без отрицательного влияния на результирующую производительность программы.

Большинство межпроцедурных оптимизаций работает с инструкциями вызовов: например, встраивание вставляет тело функции в место его вызова, если оценщик встраивания признает эту процедуру целесообразной, межпроцедурное распространение констант вставляет константы в параметры вызова и распространяет в тело вызываемой функции, если это возможно; распространение аргументов производит похожие манипуляции с переменными, и так далее.

Несложно догадаться, что ключевой зависимостью для межпроцедурных оптимизаций является зависимость по вызовам функций программ. При этом, локальные данные функций и инструкции, работающие с ними и не выводящие поток управления программы за пределы тела функции, в целом не повлияют на работу межпроцедурных оптимизаций, если они будут проводиться независимо на функциях. Таким образом, функцию можно взять за единицу анализа, а вызовы между функциями – за отношения между функциями.

По причине того, что зависимости между функциями играют ключевую роль в работе оптимизаций времени связывания, произвольное разбиение кода на части для независимой оптимизации может сильно испортить результаты межпроцедурных оптимизаций, снизив производительность собираемой программы. Таким образом, необходимо распределить функции на группы так, чтобы связей между группами было как можно меньше. Это могут быть связи вызовов, частота вызовов, а также дополнительная информация о контексте вызовов, необходимая, например, для оптимизации встраивания.

С этой целью в данной работе производится обзор алгоритмов разбиения графов, а также исследуются кластерные свойства графов вызовов для подбора подходящего алгоритма разбиения.

2.1 Определения

Для графа G = (V, E), где V - множество вершин, E – множество ребер, положим число вершин n=|V| и число ребер m=|E|. Граф называется ориентированным, если для ребра {u, v} важен порядок следования вершин, – и неориентированным, если не важен, тогда {u, v} ≡ {v, u}.

Все нижеописанные определения даны для неориентированных графов, но тривиальным образом переопределяются для ориентированных.

Определим плотность графа:

$$\delta(G) = \frac{m}{\binom{n}{2}}$$

Для *n={0,1}* положим (G)=0. Граф, плотность которого равна 1, называется полным. Если *{u, v}∈E*, то u и v называются соседями.

Матрица смежности A_G графа *G = (V, E)* порядка *n* – это матрица n×n $A_G=(a^G_{v,u})$, где

$$a\frac{G}{v,u} = \begin{cases} 1, \ \{v,\ u\} \in E, \\ 0, \ \{v, u\} \notin E \end{cases}$$

Разбиением графа назовем совокупность подмножеств $C_1, C_2, \ldots, C_k$ графа *G=(V,E)*, CiV, i=1,k, что $\bigcup_{i=1}^{k} C_i = V$. Разбиения графа могут быть пересекающимися и непересекающимися. В нашей статье речь пойдет, в основном, о непересекающихся разбиениях, таких, что $C_i \cap C_j = 0, \forall i \neq j$. Далее будем называть такие подмножества *кластерами*.

Пусть S – подмножество вершин графа G=(V,E), S⊆*V*. Индуцированным S графом называется подграф E(S): *{{u,v}|u∈V, v∈V, {u,v}∈E}*. Назовём *локальной плотностью* подграфа E(S) величину δ(G(S))=|E(S)||S|.

Степенью вершины назовём число инцидентных ей ребер, то есть deg(v) = |{{u,v}| u∈ V, u≠v}|. Внутренней степенью кластера назовём величину deg_{int} (C) = |{{v, u} ∈ E | v, u ∈ C}|, а внешней – deg_{ext} (C) = |{{v, u} ∈ E | v ∈ C, u ∈ V\C}|.

Также нам понадобится определение компоненты сильной связности ориентированного графа. Сильно связным графом G=(V,E) называется такой граф, что для любых двух его вершин u∈V, v∈V существует ориентированный путь из u в v. Компонентами сильной связности, или сильно связными компонентами, называются максимальные по включению его сильно связные подграфы.

2.2 Краткий обзор методов разбиения графов

В связи с бурным развитием области анализа больших данных, – будь то социальные сети, биологические сети, сети коммутации, – соответственно развивалась область анализа графов.

В данной работе мы не будем углубляться в обзор методов разбиения графов, а лишь приведем классификацию и коротко упомянем некоторые из них. Более подробный обзор можно прочитать в статье Shaeffer [6].

По доступной области данных методы разбиения графов делятся на глобальные и локальные. Глобальные имеют в каждый момент времени доступ ко всему графу, тогда как локальные работают с несколькими так называемыми *зерновыми* вершинами и информацией об их соседях второго уровня. Зерновые вершины могут выбираться какими-то эвристическими методами, либо случайно. В случае больших данных глобальные методы оказываются требовательны к ресурсам, требования же локальных методов не зависят от размера исходного графа. Результаты глобальных методов в целом превосходят результаты локальных, но продвинутые модификации локальных методов также хорошо работают на некоторых графах [7].

Глобальные методы, в свою очередь, делятся на итеративные и иерархические. Итеративные методы – это такие широко известные методы, как k-средних и его модификации; иерархические же методы основаны на построении множества вложенных кластеров. Проблемы итеративных методов, прежде всего, состоят в том, что необходимо точно определить метрику схожести элементов. Для графов за нее обычно берут расстояние. Также итеративные методы чувствительны к порядку обхода и начальному разбиению. Иерархические методы не чувствительны к порядку обхода и начальным данным, но основная их проблема заключается в спорном моменте, как определить конечное разбиение при полученном в результате работы алгоритма дереве кластеров.

Иерархические методы, в свою очередь, делятся на дивизивные и агломеративные по порядку обхода дерева иерархии: сверху вниз или снизу вверх. При дивизивном разбиении в начале алгоритма задан один кластер C_0=G, который итеративно разбивается на подкластеры. При агломеративном разбиении наблюдается обратный процесс: начальное разбиение представляет собой n кластеров, содержащих по одной вершине, и на каждой итерации происходит объединение существующих кластеров в более крупные.

Дивизивные методы классифицируются соответственно метрикам качества разбиения. Методы, основанные на разрезах, ищут минимальный разрез; методы, основанные на мере промежуточности, стараются максимизировать ее; спектральные методы ищут собственные вектора на матрице Лапласа. Более экзотические методы типа методов на основе цепей Маркова, электрических цепей, сводятся к описанным выше.

Самые точные результаты дают спектральные методы, но они требуют огромных математических вычислений, а значит, ресурсов. Остальные глобальные методы обладают сложностью начиная от O(n log n) и заканчивая $O(n^4)$. Некоторые вероятностностые методы имеют сложность O(n), но дают хороший результат с некоторой вероятностью. К тому же, у методов разная чувствительность к шумам в виде добавления или удаления случайных ребер.

В целом же выбор того или иного метода прежде всего зависит от свойств анализируемого графа.

2.3 Методы оценки разбиения

Оценка качества разбиения – тема, на которую по сей день ведутся споры, так как не все метрики одинаково отражают свойства полученных разбиений [9]. Поэтому для оценки качества разбиения применялись две метрики: модулярность и средняя относительная внутрикластерная плотность.

Для заданной группы кластеров $(C_1, \ldots C_k)$ вычисляется матрица относительных степеней кластеров E = $\{\mathbf{e}_{ij}\}$, где для кластеров C_i и C_j $\boldsymbol{e}_{ij} = |(u,v)|/m$, $u \in C_i$, $v \in C_j$, а $\boldsymbol{e}_{ii} = deg(C_i)/m$.

Мера модулярности кластера $Q = \boldsymbol{\Sigma}_i(\boldsymbol{e}_{ii} - \boldsymbol{a}_i^2)$, где $\boldsymbol{a}_i = \boldsymbol{\Sigma}_j\ \boldsymbol{e}_{ij}$ была введена Ньюманом[7] для оценки разбиения крупных биологических и социальных сетей. Отметим, что Q≤1, и только для “идеальных” кластеров, представляющих собой несвязные клики, Q=1. Между тем, “хорошими” можно считать кластеры, модулярность которых порядка 10^{-1}, и разбиение можно назвать “плохим”, если Q<0.

Вторая мера – относительная внутрикластерная плотность, – это отношение плотности кластера к плотности графа целиком. Относительной плотностью подграфа назовем отношение $\frac{\delta(G(S))}{\delta(G)}$. Это более очевидная метрика, показывающая, насколько полученный кластер “плотнее” всего графа целиком.

2.4 Исследования свойств графов вызовов программ

О графе вызовов априори известно то, что это разреженные ориентированные графы. Так как иногда вызовов одной и той же функции в теле другой функции может быть несколько, граф также является взвешенным: каждому ребру мы сопоставим количество вызовов одной функции из тела другой функции. Также может оказаться полезным присваивать веса и самим узлам,

указывая, например, размер функции в инструкциях: это может помочь равномерно загружать вычислительные модули при оптимизации.

Если собираемый код представляет собой исполняемую программу, то у нее будет одна точка входа – функция main. В этом случае программа имеет древоподобный вид с выраженным корнем. Если же компонуется библиотека, то точек входа может быть несколько. В случае применения callback-функций в совокупности с системными вызовами, точек входа может быть несколько даже в исполняемой программе; тем не менее, как правило, в графе все равно прослеживается древовидность. Более всего нарушают древоподобную структуру сильно связные компоненты. В случае с графами вызовов компоненты сильной связности можно интерпретировать как рекурсивные вызовы. При разбиении графа такие подмножества целесообразно помещать в один кластер в виду высокой связности. Чтобы проверить вышеописанные предположения, мы провели статистическое исследование графов вызовов реальных программ, посчитав в них количество компонент сильной связности, а также разбив эти графы достаточно точным итеративным алгоритмом Girvan & Newman [7][8], максимизирующим меру промежуточности [10].

Диаграммы 1 и 2 показывают влияние различных форм для одних и тех же графов на качество разбиения.

Первый (синий) столбец показывает, насколько хорошо локализованы связи в исходных файлах: то есть, мы принимаем файлы за кластер и измеряем для них модулярность и относительную плотность.

Второй (красный) показывает характеристики кластеров, если в качестве исходного графа использовать граф файлов G_f, где каждому узлу v соответствует файл с исходным кодом F, а каждому ребру (u,v) – вызовы функции g внутри функций f, где g – функция из файла F'≠F. При этом, из-за небольшого количества файлов в некоторых тестах, граф для них разбить на 4 компоненты не удалось.

Третий (желтый) показывает, насколько удачно получается разбить графы вызовов посредством алгоритма Girvan & Newman.

Четвертый (зелёный) соответствует кластеризации графа компонент сильной связности.

Следующие три – это работа алгоритма с учетом весов ребер.

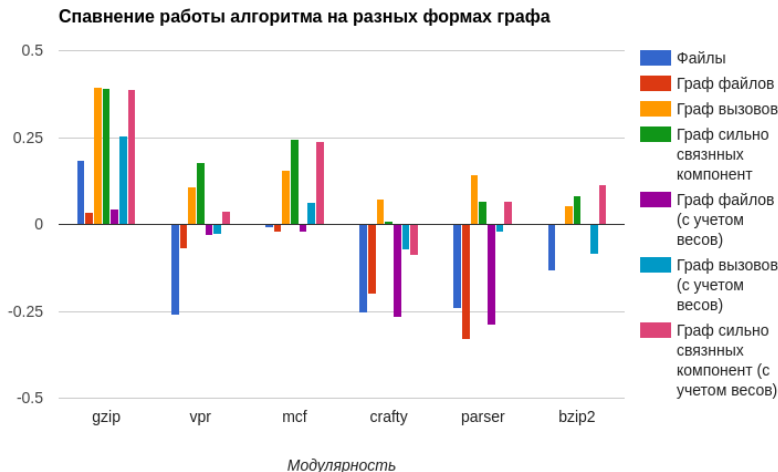


Диаграмма 1. Сравнение модулярности графов

Diagram 1. Graph modularity comparison

Результаты исследования мы трактовали следующим образом. Графы вызовов – разреженные графы, в основном связные, с небольшой плотностью и плохо выраженными естественными кластерами. При этом функции далеко не всегда грамотно разбиты на файлы, что и обуславливает необходимость оптимизации времени связывания.

Проведение разбиения на графах компонент сильной связности почти так же эффективно, как на самих графах вызовов, при этом учет весов незначительно ухудшает результат.

Проведение разбиения графа файлов возможно, но эффективность полученного разбиения зависит от аккуратности разработчиков программы.

Заметим, что для алгоритма разбиения графа на компоненты совсем не обязательно уметь выявлять естественные кластеры, ведь их размеры сильно варьируются, а поиск – очень дорогостоящая задача, на тему которой в данный ведутся многочисленные исследования [11] и даже проводятся конкурсы [12]. Для использования в задаче разбиения графа вызовов подойдет алгоритм, который будет работать за линейное или псевдолинейное время, и на выходе которого будут получены одинаковые по размеру разбиения с достаточно высокой плотностью.

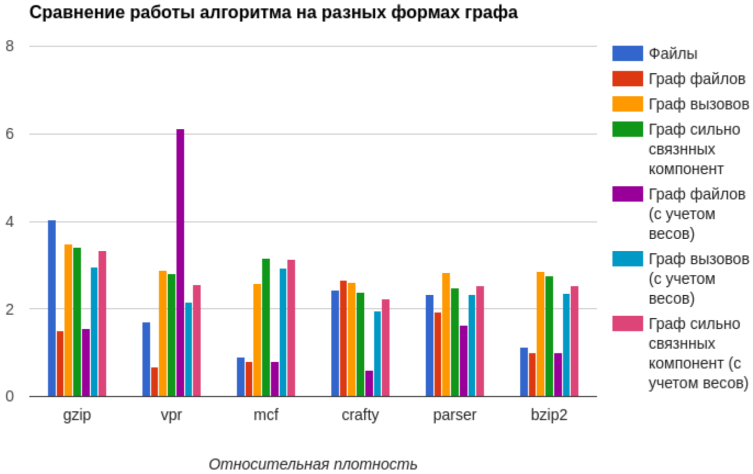


Диаграмма 2. Сравнение относительной плотности графов

Diagram 2. Graph relative density comparison

2.5 Описание предлагаемого алгоритма

Алгоритм работает в два этапа: на первом этапе формируется граф $G_c=(V_c, E_c)$ компонент сильной связности и ориентированный граф приводится к неориентированному виду, на втором этапе полученный граф разбивается на k частей.

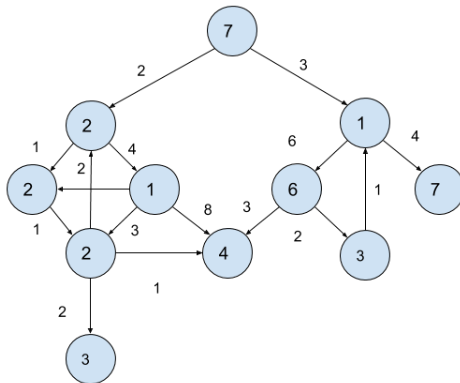


Рис. 1. Пример взвешенного ориентированного графа.

Fig. 1. Weighted oriented graph example

На начальном этапе в нашем распоряжении взвешенный ориентированный граф G = (V,E). Компоненты сильной связности могут быть найдены посредством алгоритма Косарайю или Тарьяна.

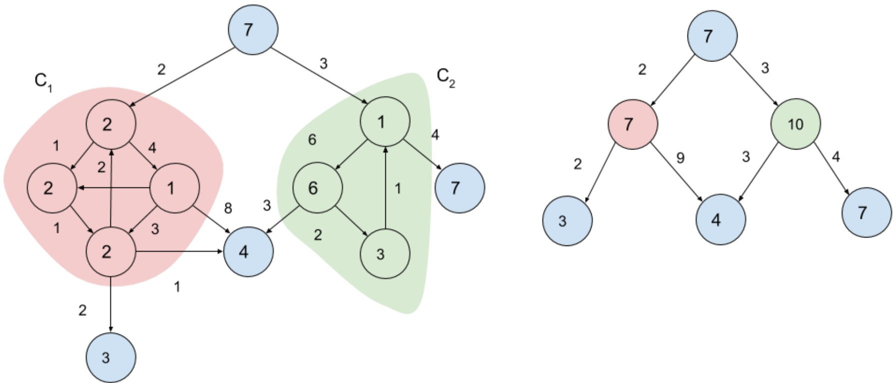


Рис. 2. Преобразование графа в граф сильно связных компонент.

Fig. 2. Graph transformation to strong-connected component graph

Затем для найденных компонент связности формируется множество V_c вершин G_c ***sc****:* $v \longrightarrow v_c$. При этом, если вершина $v \in V$ не входит ни в одну найденную компоненту из V_c, то сама вершина v помещается в V_c. Ребра получаются тривиальным образом: $\forall v \in V$: $v \longrightarrow v_c$, $u \in V$: $u \longrightarrow u_c$, $u_c \neq v_c$, если $\{v,u\} \in E$, то $\{v_c,u_c\} \in E_c$. Веса ребер также: $w(vc,uc)=\sum_{v \to v_c,\ u \to u_c} w(v,u)$. Если имеют место взвешенные вершины, то вес тоже получается за счет суммирования весов вершин, входящих в компоненту.

Затем граф приводится к неориентированному виду тривиальным образом. Так, матрица смежности графа становится симметричной относительно главной диагонали. В результате этих манипуляций получаем граф $G'=(V', E')$.

Перейдем к самому разбиению. Положим $n=|V'|$, $m=|E'|$, а k – число кластеров, на которые нужно разбить граф, $k < n$. Первая наша задача – выбрать потенциальные центры кластеров, обладающие большой “гравитацией ”. Для этого выбираем $2k$ вершин с максимальным удельным весом. Удельный вес каждой вершины $v \in V'$ считается по формуле $\omega(v)=w(v)\cdot\sum_{(v,u)\in E'} w(v,u)$, где $w(v)$ – вес вершины или 1, если вес не задан, а $w(v,u)$ – вес ребра (v,u). Далее для каждого потенциального центра алгоритмом Дейкстры ищем расстояния от данной вершины до других центров. В определении расстояния могут возникнуть разночтения из-за того, что мы имеем дело со взвешенным графом. То, как мы трактуем веса ребер,

зависит исключительно от контекста. Так как алгоритм рассчитан на работу с графами вызовов программ, и веса ребер в нашем случае – результат профилирования или количество статических вызовов, то для расстояния между соседними вершинами логично использовать величину, обратную весу ребер. Подробно о том, как вычислялись расстояния в нашем графе, описано в 4.1.

Сортируем эти вершины по удаленности от остальных в порядке убывания. В качестве меры удаленности от других вершин можно брать среднее или среднее квадратичное расстояние. Для простоты в наших примерах возьмём среднее. Получаем вектор $\boldsymbol{c}=\{c_1,c_2,...,c_k\}$, $c_i \in V'$ $\forall$ $i=1..k$, и выполняется неравенство $\sum_{i\neq j,i\leq k,j\leq k} d(c_i,c_j) > \sum_{j\neq p,p\leq k,p\leq k} d(c_j,c_p)$, $\forall i\in[1..k], j\in[1..k]$, $i<j$. Далее для каждого центра C_i жадным алгоритмом с помощью обхода в ширину набираем в кластер C_i $\lfloor N/k \rfloor$ вершин. Если у какого-то центра закончились соседи, то назначаем центром следующий по списку центр, пока не наберется требуемое количество вершин или не кончатся центры.

Если случилось так, что центры все равно находились слишком близко, и быстро исчерпали всех соседей, то добавляем в кластеры оставшиеся вершины случайным образом.

Центров кластеров $2k$ лишь потому, что может возникнуть ситуация взаимного поглощения кластеров, когда 2 центра оказываются слишком близко, и один из них поглощает всех соседей второго.

Сложность полученного алгоритма зависит от свойств графа, от выбранного k, а также от используемых вспомогательных алгоритмов для промежуточных целей. В случае разреженных графов алгоритм показывает асимптотику роста $O(k\, n \log n + k\, m \log n)$.

k=2 d(v1,v2)=8, d(v1,v3)=7, d(v1,v6)=11,
c_0={v1,v2,v3,v6} d(v2,v3)=8, d(v2,v6)=14, d(v3,v6)=6

D(v1)=9, D(v2)=10, D(v3)=7, D(v6)=10
c={v2,v6,v1,v3}

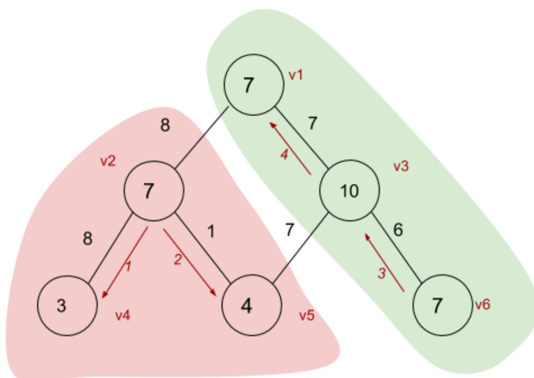


Рис. 3. Пример работы алгоритма на графе. Нумерованными стрелками показан порядок обхода вершин.

Fig. 3. Applying the algorithm to SCC-graph example. Order of vertex handling is denoted by numbered arrows

3. Легковесные оптимизации

Необходимость облегчать оптимизации обусловлена не только необходимостью сокращения времени сборки приложений; в большинстве масштабируемых систем оптимизации времени связывания это неотъемлемая часть фазы анализа программы, когда в памяти находится не весь код программы, а лишь информация, необходимая для анализа связей в программе [13][14].

Суть метода заключается в разбиении преобразования на 2 стадии: анализирующую и оптимизирующую. Первая стадия проводится на этапе компиляции и генерации промежуточного представления для файлов исходного кода. Когда промежуточное представление оказывается сформировано, анализатор обходит его и формирует дополнительную минимально необходимую информацию, достаточную для проведения оптимизации конкретного типа, – *аннотации*. На второй стадии аннотации считываются и обрабатываются оптимизирующим пребразованием, непосредственно меняющим код. Вышеописанный метод приводит к увеличению размера промежуточного представления, но ускоряет процесс

анализа кода программы, а самое главное – позволяет компоновщику не загружать часть кода программы в память. Таким образом, метод позволяет проводить оптимизации при использовании метода ленивой загрузки кода еще до загрузки тел функций.

К сожалению, не каждое существующее межпроцедурное оптимизирующее преобразование можно с легкостью привести к аналогичному, работающему только на аннотациях. В рамках исследования уже было реализовано построение графа вызовов с помощью аннотаций [5], который может быть использован межпроцедурными преобразованиями, не требующими сигнатуру вызова. Для преобразований, требующих доступ к локальным переменным и инструкциям не всегда оптимально использовать аннотации, поскольку загрузка тел функций оказывается дешевле, чем накладные расходы на поддержание аннотаций.

Таким образом, необходимо определить, какие преобразования возможно и необходимо адаптировать в первую очередь. Так как работа ведется на основе компиляторной системы LLVM, были исследованы уже реализованные в ней оптимизирующие проходы.

Оптимизации времени связывания в LLVM представлены спектром зарекомендовавших себя методов, в их числе: встраивание кода, межпроцедурное удаление мёртвого кода, межпроцедурное распространение констант и т.д. В сочетании с локальными методами оптимизации и использованием профиля разработчик может получить увеличение производительности программы до 30%.

Мы проанализировали существующие на данный момент оптимизации и сравнили влияние на производительность существующих методов без использования профиля. Оценить влияние оптимизирующих преобразований оптимизации времени связывания LLVM на улучшение производительности довольно сложно из-за их большого количества и влияния друг на друга.

Поэтому для оценки влияния была предложена следующая схема:

1. Для каждого преобразования запускалась серия тестов, в которой оно исключалось из списка оптимизирующих преобразований.
2. Запускалась серия тестов, в которой участвовали все преобразования.
3. Для каждого теста из серии строился массив влияния преобразования на производительность. Элемент массива представляет собой разность результатов тестов с пунктов 1 и 2 для каждого преобразования. Массив сортировался, лучшие 4 преобразования получали баллы от 4 до 1, худшие 4 от -4 до -1, остальные – 0.
4. Баллы для каждого преобразования суммировались по всей серии тестов. На основе данной схемы оценки была построена следующая диаграмма для оптимизирующих преобразований LTO LLVM.

Мы также проанализировали, какую информацию используют оптимизирующие преобразования, чтобы выяснить, какие из них могут проводиться в облегченном варианте, а какие – нет. Большинству оптимизирующих преобразований необходим доступ к инструкциям, базовым блокам и инструкциям вызовов. Это значит, что им необходим доступ к телам функций для выполнения анализа, что нежелательно в случае ленивой загрузки кода. В числе таких трудноадаптируемых межпроцедурных методов оказались: продвижение аргументов, распространение констант и даже встраивание. Тем не менее, самое эффективное преобразование с точки зрения тестирования – межпроцедурное удаление мертвого кода, – практически не затрагивает код, а лишь просматривает использование глобальных переменных в коде.

Таким образом, для реализации метода легковесной оптимизации был выбран метод межпроцедурного удаления мёртвого кода (GlobalDCE).

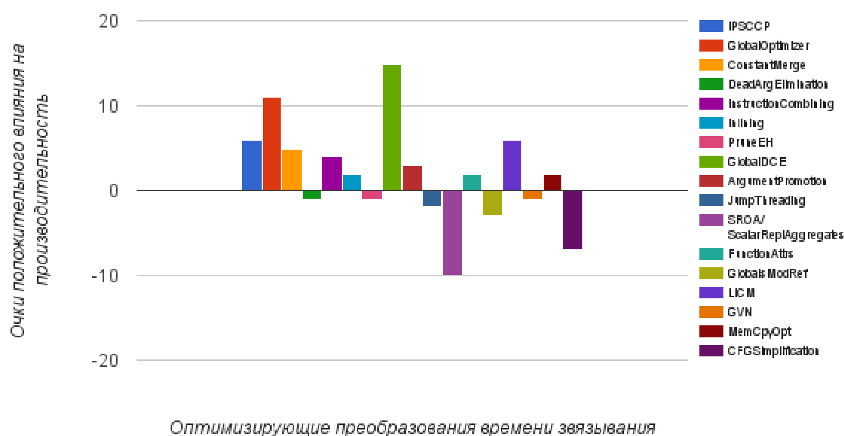


Диаграмма 3. Влияние оптимизирующих проходов на производительность тестов

Diagram 3. Performance impact of optimization passes

4. Реализация методов

Оба метода были реализованы в разрабатываемой в Институте системного программирования масштабируемой системе оптимизаций времени связывания на основе LLVM [4][5].

4.1 Реализация метода разбиения

За счет наличия в LLVM эффективной алгоритмической базы, некоторые вспомогательные алгоритмы были реализованы с их помощью. Например, в системе LLVM уже присутствует алгоритм поиска компонент сильной

связности в графе, поэтому он был использован для построения графа компонент сильной связности. Для сортировки массивов использовались функции стандартной библиотеки STL. Тем не менее, дополнительно была реализована двоичная куча для подсчета расстояний между вершинами графа.

Для поиска расстояний в графе использовалась следующая формула преобразования весов ребер: $w'(v)=\max_{u\in E'}(w(u)) + 1 - w(v)$, которая использовалась для нахождения наиболее удаленных вершин в массиве.

Для исследования были взяты некоторые тесты из тестового набора SPEC CPU2000, для которых удалось построить все структуры и на которых удалось запустить оба алгоритма поиска разбиения.

Табл. 1. Сравнение модулярности при разбиении предложенным алгоритмом.

Table 1. Modularity measurements of graph division performed by proposed algorithm

Модулярность	**Граф файлов**	**Граф вызовов**	**Граф сильно связных компонент**	**Граф файлов (с учетом весов)**	**Граф вызовов (с учетом весов)**	**Граф сильно связных компонент (с учетом весов)**
gzip	0.02	0.02	-0.15	-0.04	-0.06	-0.03
vpr	-0.28	-0.28	-0.34	-0.33	-0.05	-0.06
mcf	-0.01	-0.06	-0.25	-0.01	0.19	0.16
crafty	-0.28	-0.43	-0.42	-0.36	-0.43	-0.44
parser	-0.36	-0.28	-0.42	-0.21	-0.32	-0.41
bzip2	0	-0.54	-0.55	0	-0.45	-0.45

Табл. 2. Сравнение относительной плотности при разбиении предложенным алгоритмом.

Table 2. Relative density measurements of graph division performed by proposed algorithm

Относительная плотность	**Граф файлов**	**Граф вызовов**	**Граф сильно связнных компонент**	**Граф файлов (с учетом весов)**	**Граф вызовов (с учетом весов)**	**Граф сильно связнных компонент (с учетом весов)**
gzip	2.37	2.75	2.14	3.03	2.81	2.78
vpr	1.17	1.7	1.48	1.05	2.57	2.55
mcf	2.44	3.17	1.62	1.7	3.89	4.21
crafty	1.68	1.62	1.33	1	1.58	2.2
parser	1.81	1.76	1.6	1.09	2.61	2.38
bzip2	0.25	1.11	1.81	0.25	2.42	2.47

В сравнении с алгоритмом Джирвана и Ньюмана, а также с простым жадным алгоритм показывает результаты, представленные на диаграммах 4 и 5.

Алгоритм не показал высокой эффективности в поиске “естественных” кластеров, и его результаты немногим лучше случайного жадного алгоритма, тогда как при учете весов он проявил себя в поиске наиболее плотных кластеров, что с учетом дополнительного ограничения на размер кластеров и небольшой сложности алгоритма – весомый аргумент в его пользу.

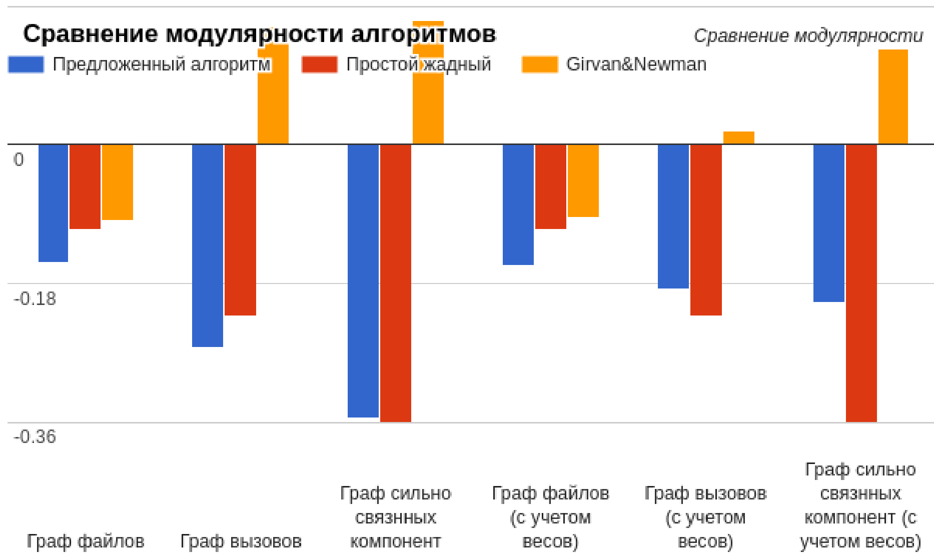


Диаграмма 4. Сравнение работы различных алгоритмов относительно модулярности

Diagram 4. Comparison of modularity of graph division performed by different algorithms

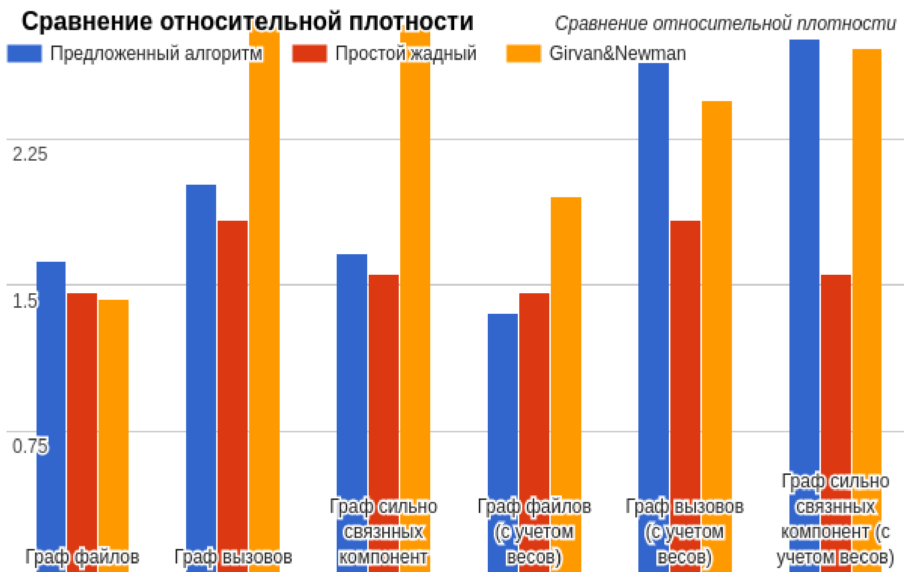


Диаграмма 5.Сравнение алгоритмов относительно относительной плотности

Diagram 5. Comparison of relative density of graph division performed by different algorithms

4.2 Реализация оптимизации удаления мертвых глобальных переменных

GlobalDCE (Global Dead Code Elimination) – оптимизация, выполняющая поиск и удаление из программы мертвого кода. Мертвым кодом называют код, результат выполнения которого не влияет на результат работы программы. Эта оптимизация полезна тем, что позволяет уменьшить количество выполняемых операций и размер промежуточного представления.

В LTO LLVM GlobalDCE реализован как агрессивный алгоритм, который ищет глобальные переменные и функции, которые «живы», хранит их в списке. В дальнейшем все, что не попало в этот список, удаляется. В частности, GlobalDCE также позволяет удалить части недостижимого кода. В процессе работы плагина Gold GlobalDCE запускается несколько раз.

Предлагается заменить это оптимизирующее преобразование двумя. Одно, как уже говорилось выше, будет запускаться на этапе работы Clang и, аналогично принципам поиска «живых» переменных в GlobalDCE, искать глобальные переменные, которые используются в модуле. Второе, основываясь на результате первого преобразования, в процессе работы плагина Gold проанализирует полученную информацию и удалит неиспользуемые переменные.

Одним из способов прикрепления дополнительной информации к инструкциям является использование метаданных. Метаданные позволяют передать дополнительную информацию о коде оптимизирующим преобразованиям или генераторам кода. Примером может служить отладочная информация. В LLVM есть два примитива метаданных – строки и узлы. Синтаксически метаданные помечаются специальным символом '!'. Метаданные можно прикреплять к модулям, функциям, инструкциям. Также метаданные не имеют типа и это не значения. Рассмотрим каждый примитив подробнее.

Строка метаданных – строка, заключенная в двойные кавычки. Причем в строку может быть включен любой символ, в том числе даже непечатаемые символы вида '\xx', где xx – двузначный шестнадцатеричный код символа.

Узлы метаданных похожи по устройству на структуры – список разнообразных элементов, разделенных запятой и заключенных в кавычки. Элементом узла могут быть не только строки, но также операнды, константы, функции и т.д.

Отдельно стоит отметить, что узлам метаданных можно давать имена, по которым легко найти узел в любой момент компиляции либо просто определить, есть ли такие данные.

Для оптимизирующего преобразования оптимальным является использование именованного узла метаданных, прикрепленного к каждой глобальной переменной.

5. Результаты

Тестирование проводилось на четырехъядерном Intel(R) Core(TM) i5-2500 на тестовом наборе SPEC CPU 2000 (int) [15].

Диаграмма 6 показывает сравнение времени работы жадного алгоритма, работающего за время O(m), с предложенным алгоритмом разбиения. Как видно из диаграммы, отличия присутствуют лишь на тесте gcc ввиду большого размера графа вызовов программы и составляют 2%.

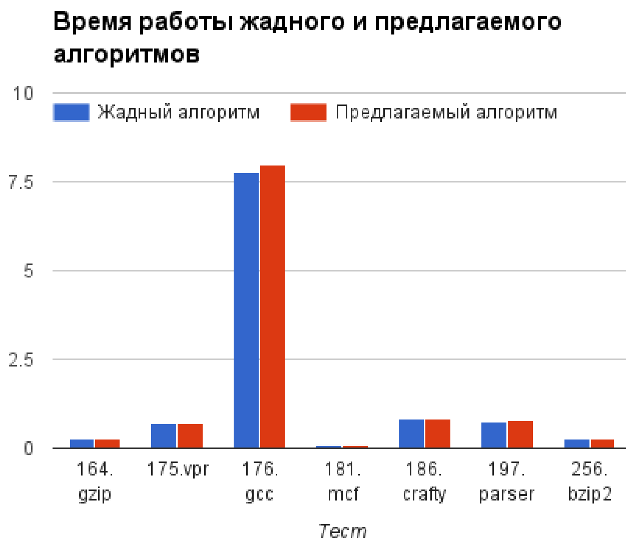


Диаграмма 6. Сравнение времени работы разработанного алгоритма с жадным

Diagram 6. Run time comparison of proposed algorithm vs greedy

Табл. 3. Измерение времени работы сборки с разбиением посредством предлагаемого алгоритма разбиения

Table 3. Run time of building with proposed algorithm graph division and parallel optimization

Тест	**Сборка без распараллеливания, с**	**Сборка с распараллеливанием в 4 потока, с**	**Ускорение, %**
164.gzip	0.28	0.26	7.28
175.vpr	1.06	0.71	32.9
176.gcc	13.88	7.97	42.60
181.mcf	0.13	0.09	31.44
186.crafty	1.5	0.85	43.14
197.parser	1.07	0.78	27.66
256.bzip2	0.4	0.24	38.67
Суммарное время	**18.31**	**10.89**	**31.9**

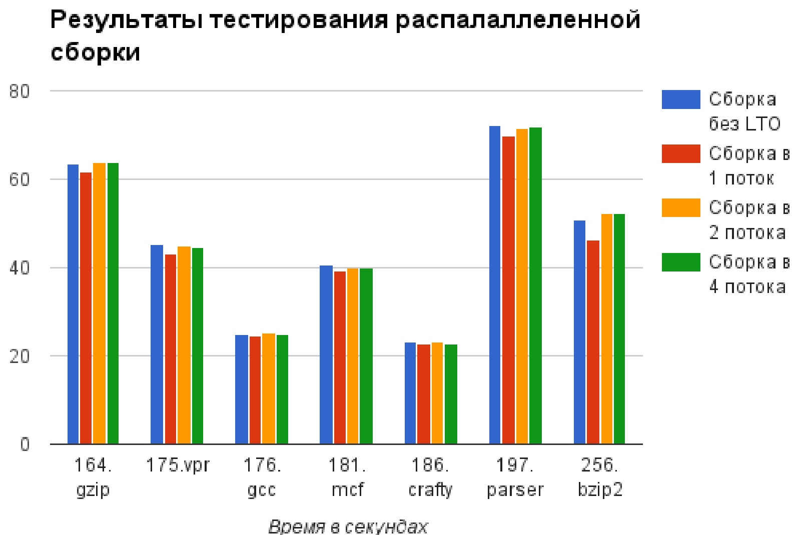


Диаграмма 7. Сравнение производительности программ, собранных с разными уровнями оптимизации времени связывания

Diagram 7. Performance measurements of benchmark building with different LTO parallelization levels

На *таблице 3* видно ускорение сборки программ в среднем на 31%, при этом алгоритм разбиения замедляет работу программы чуть более, чем работающий за O(m) жадный алгоритм.

Так как распараллеливание затрагивает лишь оптимизационную часть сборки и не может быть распространено на этап чтения и компоновки файлов с промежуточным представлением, а также требует дополнительных накладных расходов по времени, теоретический максимум ускорения сборки на 4х потоках составляет всего 50-60% в зависимости от программы.

На *диаграмме 7* приведены сравнения времен работы программ, собранных без оптимизации времени связывания, с оптимизацией в 1 поток, 2 и 4 соответственно.

При тестировании оптимизация проводилась исключительно на разных потоках, и ни предварительного анализа, ни оптимизации перед разбиением проведено не было. Также отсутствовал сильно влияющий на производительность фактор – данные профилирования. Это влечет за собой необходимость более тщательного тестирования, а также работы над предварительным анализом программ. Поэтому довольно сложно сказать, что точно в некоторых случаях отрицательно сказалось на производительности: алгоритм разбиения, отсутствие анализа всей программы или специфика программы. Также ввиду того, что выбранный тестовый набор представляет собой относительно небольшие тесты с небольшим временем сборки, сами

результаты межпроцедурной оптимизации весьма невелики, а «разрыв» некоторых межпроцедурных связей ведет к деградации производительности. Для полной картины необходимо тестирование на целевых для нашей задачи крупных программах: Mozilla Firefox, LibreOffice, Android OS.

Результаты тестирования, указанные в *таблице 4*, являются усредненными значениями серии из 3х тестов. В среднем прирост производительности составляет 0,51%. Стоит отметить, что на отдельных тестах прирост производительности выше — gzip, gcc. Эти тесты состоят из большого числа модулей, на такие программы рассчитана оптимизация. Этим же объясняется ухудшение производительности на тесте bzip2.

Тест состоит всего из одного модуля, поэтому оптимизация ведет к небольшому ухудшению. Из результатов видно, что в большинстве случаев использование аннотаций приводит к улучшению производительности. Стоит также отметить, что тестирование производилось при ленивой загрузке модулей программы на этапе оптимизации времени связывания.

Табл. 4. Измерение влияния легковесного оптимизирующего прохода на производительность.

Table 4. Performance measurements of lightweight GlobalDCE pass

Название теста	**Время работы без оптимизации, с**	**Время работы с оптимизацией, с**	**Прирост производительности, %**
164.gzip	78.8	77.9	1.14
175.vpr	58.4	58.3	0.17
176.gcc	33.4	32.8	1.79
181.mcf	86.9	86.6	0.34
197.parser	95.4	94.2	0.21
256.bzip2	69.3	69.7	-0.57
	421.2	**419.5**	**0.51**

Заключение

Данная статья является продолжением исследования на тему “Масштабирование системы оптимизации времени связывания” и описывает работы по проведению горизонтальной масштабируемости, а также разработке легковесных оптимизирующих преобразований. В статье проведено исследование разбиения графов вызовов программ на компоненты для обеспечения параллельной оптимизации, а также предложен алгоритм разбиения, обеспечивающий высокую внутреннюю плотность компонент при равном числе узлов. Также приведено описание реализации оптимизирующего прохода межпроцедурного удаления мёртвого кода для работы только с

аннотациями. В статье приведены результаты тестирования разработанных методов на тестовом наборе SPEC CPU2000 (INT).

В дальнейшем планируется расширить возможности системы для работы с профилем программы, а также провести тестирование на больших современных программах с открытым исходным кодом, таких как LibreOffice и браузер Firefox с использованием более мощных многоядерных архитектур с возможностью выполнять значительно большее число потоков одновременно.

Литература

[1]. Andrew Ayers, Richard Schooler, and Robert Gottlieb. Aggressive inlining. SIGPLAN Not., 32(5):134–145, 1997. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/258916.258928.

[2]. Honza Hubicka. Linktime optimization in GCC, part 2 – Firefox. http://hubicka.blogspot.ru/2014/04/linktime-optimization-in-gcc-2-firefox.html.

[3]. Honza Hubicka. Linktime optimization in GCC, part 3 – LibreOffice. http://hubicka.blogspot.ru/2014/09/linktime-optimization-in-gcc-part-3.html.

[4]. К. Ю. Долгорукова. Обзор масштабируемых систем межмодульных оптимизаций. Труды ИСП РАН, том 26, вып. 3, 2014 г., стр. 69-90. DOI: 10.15514/ISPRAS-2014-26(3)-3.

[5]. К. Ю. Долгорукова. Разработка и реализация метода масштабирования по памяти для систем межмодульных оптимизаций и статического анализа на основе LLVM. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 97-110. DOI: 10.15514/ISPRAS-2015-27(6)-7.

[6]. Satu Elisa Schaeffer. Graph clustering. Survey. Computer Science Review. Volume 1, Issue 1, August 2007, Pages 27–64. DOI:10.1016/j.cosrev.2007.05.001.

[7]. M. Girvan, M. E. J. Newman. Community structure in social biological networks. PNAS, June 11, 2002, vol.99, no.12. DOI: 10.1073/pnas.122653799.

[8]. M. E. J. Newman. Scientific collaboration networks. II. Shortest paths, weighted networks, and centrality. Physical review E, vol. 64, 016132. DOI: 10.1103/PhysRevE.64.016132.

[9]. L. da F. Costa, F. A. Rodrigues, G. Travieso & P. R. Villas Boas (2007). Characterization of complex networks: A survey of measurements, Advances in Physics, 56:1, 167-242. DOI:10.1080/00018730601170527

[10]. M. E. J. Newman, M. Girvan. Finding and evaluating community structure in networks. Phys. Rev. E 69, 026113 (2004). arXiv:cond-mat/0308217 DOI:10.1103/PhysRevE.69.026113.

[11]. L. Danon, A. Díaz Guilera, J. Duch, A. Arenas, Comparing community structure identification, Journal of Statistical Mechanics Theory and Experiment (2005) P09008.

[12]. D. A. Bader, H. Meyerhenke, P. Sanders, D. Wagner. Contemporary mathematics: graph partitioning and graph clustering. 10th DIMACS Implementation Challenge Workshop, February 13–14, 2012, Georgia Institute of Technology, Atlanta, GA.

[13]. Preston Briggs, Doug Evans, Brian Grant, Robert Hundt, William Maddox, Diego Novillo, Seongbae Park, David Sehr, Ian Taylor, Ollie Wild. WHOPR - Fast and Scalable Whole Program Optimizations in GCC. Initial Draft, 12 Dec 2007.

[14]. Gautam Chakrabarti, Fred Chow. Structure Layout Optimizations in the Open64 Compiler: Design, Implementation and Measurements. Open64 Workshop at CGO 2008, April 6, 2008. Boston, Massachusetts.

[15]. SPEC CPU benchmark. https://www.spec.org/cpu2000.

Link-time optimization speedup

K. Dolgorukova <unerkannt@ispras.ru>
S. Arishin <arishin@phystech.edu>
Institute for System Programming of the Russian Academy of Sciences, 109004, Moscow, Alexander Solzhenitsyn st., 25.

Abstract. This paper proposes the two different approaches to speed-up program build: making link-time optimization work in parallel and lightweight optimization approach. The former technique is achieved by scaling LTO system. The latter makes link to work faster because of using summaries to manage some of interprocedural optimization passes instead of full IR code in memory.

The problem of horizontal LTO system scaling leads to the problem of partition of the large task to several subtasks that can be performed concurrently. The problem is complicated by the compiler pipeline model: interprocedural optimization passes works consequentially and depends on previous performed ones. That means we can divide just data on which passes works, not passes themselves. We need to divide IR code to sub-independent parts and run LTO on them in parallel. We use program call graph analysis to divide a program to parts. Therefore, our goal is to divide call graph that is one of NP-compete problems. Nevertheless, the choice of the dividing algorithm strongly depends on properties of divided graph.

The main goal of our investigation is to find lightweight graph partition algorithm that works efficiently on call graphs of real programs and that does not spoil LTO performance achievements after optimizing code pieces separately.

This paper proposes new graph partition algorithm for program call graphs, results of comparing this one with two other methods on SPEC CPU2000 benchmark and implementation of the algorithm in scalable LLVM-based LTO system. The implementation of this approach in LTO system shows 31% link speedup and 3% of performance degradation for 4 threads.
The lightweight optimization shows 0,5% speedup for single run in lazy code loading mode.

Keywords: compilers; link-time optimization; scalability; graph division; graph clustering.

DOI: 10.15514/ISPRAS-2016-28(5)-11

For citation: K. Dolgorukova, S. Arishin. Link-time optimization speedup. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 5, 2016, pp. 175-198 (in Russian). DOI: 10.15514/ISPRAS-2016-28(5)-11

References

[1]. Andrew Ayers, Richard Schooler, and Robert Gottlieb. Aggressive inlining. SIGPLAN Not., 32(5):134–145, 1997. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/258916.258928.

[2]. Honza Hubicka. Linktime optimization in GCC, part 2 – Firefox. http://hubicka.blogspot.ru/2014/04/linktime-optimization-in-gcc-2-firefox.html.

[3]. Honza Hubicka. Linktime optimization in GCC, part 3 – LibreOffice. http://hubicka.blogspot.ru/2014/09/linktime-optimization-in-gcc-part-3.html.

[4]. K. Dolgorukova. [Overview of Scalable Frameworks of Cross-Module Optimization]. Trudy ISP RAN/Proc. ISP RAS, vol. 26, issue 3, 2014, pp. 69-90 (in Russian). DOI: 10.15514/ISPRAS-2014-26(3)-3.

[5]. K. Dolgorukova. [Implementation of Memory Scalability Approach for LLVM-Based Link-Time Optimization and Static Analyzing Systems]. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015, pp. 173-192 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-7.

[6]. Satu Elisa Schaeffer. Graph clustering. Survey. Computer Science Review. Volume 1, Issue 1, August 2007, Pages 27–64. DOI:10.1016/j.cosrev.2007.05.001.

[7]. M. Girvan, M. E. J. Newman. Community structure in social biological networks. PNAS, June 11, 2002, vol.99, no.12. DOI: 10.1073/pnas.122653799.

[8]. M. E. J. Newman. Scientific collaboration networks. II. Shortest paths, weighted networks, and centrality. Physical review E, vol. 64, 016132. DOI: 10.1103/PhysRevE.64.016132.

[9]. L. da F. Costa, F. A. Rodrigues, G. Travieso & P. R. Villas Boas (2007). Characterization of complex networks: A survey of measurements, Advances in Physics, 56:1, 167-242. DOI:10.1080/00018730601170527

[10]. M. E. J. Newman, M. Girvan. Finding and evaluating community structure in networks. Phys. Rev. E 69, 026113 (2004). arXiv:cond-mat/0308217 DOI:10.1103/PhysRevE.69.026113.

[11]. L. Danon, A. Díaz Guilera, J. Duch, A. Arenas, Comparing community structure identification, Journal of Statistical Mechanics Theory and Experiment (2005) P09008.

[12]. D. A. Bader, H. Meyerhenke, P. Sanders, D. Wagner. Contemporary mathematics: graph partitioning and graph clustering. 10th DIMACS Implementation Challenge Workshop, February 13–14, 2012, Georgia Institute of Technology, Atlanta, GA.

[13]. Preston Briggs, Doug Evans, Brian Grant, Robert Hundt, William Maddox, Diego Novillo, Seongbae Park, David Sehr, Ian Taylor, Ollie Wild. WHOPR - Fast and Scalable Whole Program Optimizations in GCC. Initial Draft, 12 Dec 2007.

[14]. Gautam Chakrabarti, Fred Chow. Structure Layout Optimizations in the Open64 Compiler: Design, Implementation and Measurements. Open64 Workshop at CGO 2008, April 6, 2008. Boston, Massachusetts.

[15]. SPEC CPU benchmark. https://www.spec.org/cpu2000.

Задача глобального распределения регистров во время динамической двоичной трансляции

К.А. Батузов <batuzovk@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. Распределение регистров оказывает существенное влияние на производительность генерируемого кода. В данной статье исследуется задача распределения регистров во время динамической двоичной трансляции. Так как существующие алгоритмы распределения регистров рассчитаны на использование в компиляторах, они плохо подходят для использования во время динамической двоичной трансляции. Был разработан новый алгоритм, который определяет, какие переменные должны располагаться на каких регистрах в начале и в конце базового блока (назовем эти ограничения пред- и постусловиями данного базового блока), а затем решает задачу локального распределения регистров в данных ограничениях. Для обеспечения корректности ограничений алгоритм должен работать так, чтобы бля любого базового блока b', предшествующего блоку b, постусловия блока b' совпадали с предусловиями блока b. Этого можно достичь, если выделить в графе потока управления группы дуг, на всех концах которых ограничения должны быть неизменны. Такие дуги называются точками синхронизации. Точки синхронизации являются связными компонентами в неориентированном графе, вершинами которого являются дуги графа потока управления, а ребрами соединены те дуги-вершины, которые либо входят, либо выходят из одного базового блока. Данное утверждение позволяет эффективно находить точки синхронизации. Для определения того, сколько переменных должно находиться на регистрах в точке синхронизации, количество доступных регистров оценивается с помощью регистрового давления. Затем выбираются конкретные переменные на их частоты использования в данном фрагменте кода. Алгоритм локального распределения регистров был модифицирован, чтобы использовать предусловия и обеспечивать постусловия. В статье приводится эффективный алгоритм для приведения существующего распределения в конце базового блока к требуемым постусловиям и доказывается оптимальность этого алгоритма. Применение описанного алгоритма распределения регистров привело к сокращению времени работы синтетического примера на 29.6%.

Ключевые слова: глобальное распределение регистров; динамическая двоичная трансляция; эмуляторы; QEMU.

DOI: 10.15514/ISPRAS-2016-28(5)-12

Для цитирования: К.А. Батузов. Задача глобального распределения регистров во время динамической двоичной трансляции. Труды ИСП РАН, том 28, вып. 5, 2016, стр. 199-214. DOI: 10.15514/ISPRAS-2016-28(5)-12

1. Введение

При разработке виртуальных машин возникает задача выполнения кода, написанного для одной процессорной архитектуры, на другой архитектуре. Для ее решения используется динамическая двоичная трансляция. Трансляция кода ведется небольшими фрагментами, которые называются блоками трансляции. Типичная реализация динамической двоичной трансляции предполагает сначала дизассемблирование блока трансляции в некоторое внутреннее представление, а затем генерацию машинного кода для целевой архитектуры из этого внутреннего представления. На втором этапе возникает необходимость распределения регистров. Данная статья посвящена вопросу решения задачи глобального распределения регистров во время динамической трансляции.

Распределение регистров оказывает существенное влияние на производительность полученного кода. При этом необходимо учитывать, что в случае динамической двоичной трансляции распределение регистров происходит во время работы программы, и время работы самого алгоритма должно быть как можно меньшим. Локальное распределение регистров может быть реализовано очень быстрым жадным алгоритмом [1], однако глобальное распределение регистров может давать более эффективный результирующий код. Существующие алгоритмы глобального распределения регистров (алгоритмы раскраски графов зависимостей [2], алгоритмы линейного сканирования [3]) не вполне учитывают ограничения на накладные расходы для выполнения самого алгоритма. Поэтому был разработан новый алгоритм, который выполняет глобальное распределение регистров с очень низкими накладными расходами. Данный алгоритм учитывает особенности кода, возникающего во время динамической двоичной трансляции. В своей работе он использует локальный алгоритм для распределения регистров внутри базового блока.

Полученный алгоритм был реализован в эмуляторе QEMU [4]. Этот эмулятор был выбран, так как он обладает открытым исходным кодом, а также широко применяется на практике.

В качестве внутреннего представления в QEMU используется последовательность инструкций, оперирующих с переменными трех видов и константами. Инструкции представляют собой язык ассемблера упрощенной абстрактной машины и содержат арифметические и логические операции, операции установки меток, условного и безусловного перехода на метку, вызова функций, загрузки из памяти и сохранения в память. Переменные внутреннего представления делятся на три вида: глобальные, которые

существуют все время работы эмулятора, локальные, которые существуют в рамках одного блока трансляции, и временные, которые существуют в рамках одного базового блока. Вид переменной явно указывается при ее создании.

В QEMU распределение регистров объединено с генерацией кода. Вариант промежуточного представления, в котором регистры были бы частично распределены, отсутствует. Кроме того, при разработке алгоритма время его работы является очень критичным, так как он выполняется во время выполнения программы. Таким образом, при введении любого нового внутреннего представления требуется учитывать необходимые накладные расходы на создание данного представления.

Алгоритмы глобального распределения регистров, основанные на раскраске графа зависимостей, плохо решают поставленную задачу, так как они имеют непредсказуемое время работы, что недопустимо в случае динамической двоичной трансляции. Алгоритмы линейного сканирования требуют введение еще одного полноценного внутреннего представления с распределенными регистрами, так как им требуется второй проход для устранения противоречий на границах базовых блоков.

Построим новый алгоритм, который будет минимизировать расходы на дополнительное внутреннее представление, но перед этим отметим одну особенность промежуточного представления, получаемого при динамической двоичной трансляции.

При трансляции гостевого кода во внутреннее представление каждая инструкция переводится в последовательность команд внутреннего представления. Все входные аргументы эта последовательность команд получает на глобальных переменных (соответствующих регистрам гостевой архитектуры) либо в памяти. Выходные данные располагаются там же. Все же остальные переменные живы только внутри таких последовательностей и используются для хранения результатов промежуточных вычислений. Таким образом, количество переменных, интервалы жизни которых пересекают границы базовых блоков, существенно меньше общего числа переменных.

Алгоритм, описанный в статье [1], может быстро и эффективно распределять регистры в рамках одного базового блока. Не составляет труда немного модифицировать его так, чтобы он принимал во внимание условия на границах базового блока: какие глобальные переменные должны находиться на каких регистрах. Значит, можно построить алгоритм, являющийся комбинацией глобального и локального распределения регистров. Часть, отвечающая за глобальное распределение регистров, будет работать только с глобальными переменными и устанавливать условия на границах базовых блоков. Часть, отвечающая за локальное распределение регистров, будет осуществлять распределение регистров внутри базовых блоков с учетом условий на их границах. Условия на границах блока трансляции будут иметь вид множества пар *(v, r)*, означающих что переменная *v* должна в данной

точке находиться на регистре *r*. Если переменная не фигурирует ни в одной из пар, то считается что переменная должна располагаться в памяти.

2. Схема комбинированного алгоритма

Назовем условия на распределение регистров в начале базового блока начальными, а в конце — конечными, а совокупность начальных и конечных условий — граничными. Введем следующие обозначения.

- Если есть базовый блок b, то начальные условия этого базового блока обозначим $\mathrm{b}^{\mathrm{pre}}$, а конечные — как $\mathrm{b}^{\mathrm{post}}$.
- Все множество начальных условий для всех базовых блоков блока трансляции TB с графом потока управления $\mathrm{G} = \langle \mathrm{B}, \mathrm{E} \rangle$ обозначим как $\mathrm{C}^{\mathrm{pre}}$, конечных — $\mathrm{C}^{\mathrm{post}}$, а всех граничных условий — C.

Множество граничных условий C блока трансляции TB с графом потока управления $\mathrm{G} = \langle \mathrm{B}, \mathrm{E} \rangle$ назовем корректным, если $\forall(\mathrm{b}_1, \mathrm{b}_2) \in \mathrm{E}: \mathrm{b}_1^{\mathrm{post}} = \mathrm{b}_2^{\mathrm{pre}}$.

Дуги e_1и e_2 графа потока управления назовем родственными, если они выходят из одного и того же блока, либо входят в один и тот же блок. Обозначим $\mathrm{e}_1 \sim \mathrm{e}_2$.

Точкой синхронизации назовем не пустое множество J дуг графа потока управления, такое что

- для любых двух родственных дуг e_1 и e_2 выполнено соотношение $\mathrm{e}_1 \in \mathrm{J} \Leftrightarrow \mathrm{e}_2 \in \mathrm{J}$,
- для любых двух дуг u и v графа потока управления входящих в одну точку синхронизации существует последовательность дуг $\mathrm{e}_{1,}\mathrm{e}_{2,}\dots,\mathrm{e}_\mathrm{k}$ таких, что $\mathrm{e}_1 = \mathrm{u}$, $\mathrm{e}_\mathrm{k} = \mathrm{v}$, $\forall \mathrm{i} \in [1, \mathrm{k}-1] \mathrm{e}_\mathrm{i} \sim \mathrm{e}_{\mathrm{i}+1}$.

Утверждение Пусть дуги $\mathrm{e}_1 = (\mathrm{u}_{1,}\mathrm{v}_1)$ и $\mathrm{e}_2 = (\mathrm{u}_{2,}\mathrm{v}_2)$ графа потока управления принадлежат одной и той же точке синхронизации J. Тогда для любого корректного множества граничных условий C выполнены равенства $\mathrm{u}_1^{\mathrm{post}} = \mathrm{u}_2^{\mathrm{post}}$и $\mathrm{v}_1^{\mathrm{pre}} = \mathrm{v}_2^{\mathrm{pre}}$.

Это утверждение может быть легко доказано от противного на основе определения точки синхронизации.

Данное утверждение, являющееся необходимым условием корректности множества граничных условий, послужит основой для комбинированного алгоритма: необходимо для каждой точки синхронизации выбрать граничные условия, которые будут записаны на обоих концах дуг, входящих в эту точку синхронизации.

Пусть дан блок трансляции TB с графом потока управления $\mathrm{G} = \langle \mathrm{B}, \mathrm{E} \rangle$. Построим неориентированный граф $\mathrm{G}_\mathrm{E} = \langle \mathrm{E}, \mathrm{F} \rangle$, в котором $(\mathrm{e}_{1,}\mathrm{e}_2) \in \mathrm{F}$ тогда и только тогда, когда $\mathrm{e}_1 \sim \mathrm{e}_2$.

Утверждение Множество дуг $\{e_i\}$ графа потока управления $G = \langle B, E \rangle$ является точкой синхронизации тогда и только тогда, когда они образуют компоненту связности в графе G_E.

Доказательство этого утверждения следует из определений точки синхронизации и компоненты связности графа.

У данного утверждения есть ряд важных следствий.

1. Любые две точки синхронизации либо совпадают, либо не пересекаются.
2. Каждая дуга принадлежит ровно одной точке синхронизации.
3. Эффективно находить и обходить точки синхронизации можно с помощью поиска в ширину, либо поиска в глубину в графе G_E.

Далее приведем псевдокод алгоритма распределения регистров и докажем его корректность. Алгоритм будет считать использовать множество точек синхронизации и функцию для вычисления требуемых граничных условий в точке синхронизации Compute-Register-Mapping(J). Реализация данной функции будет рассмотрена позже.

Все множество точек синхронизации блока трансляции TB обозначим J(TB), множество всех базовых блоков, входящих в его поток управления — B(TB).

```
procedure Combined-Reg-Alloc(TB)
      for all b ∈ B(TB) do
            b^pre ← ∅
            b^post ← ∅
      end for
      for all J ∈ J(TB) do
            regmap ←Compute-Register-Mapping(J)
            for all (src, dst) ∈ J do
                  src^post ← regmap
                  dst^pre ← regmap
            end for
      end for
end procedure
```

Утверждение Множество граничных условий, полученное в ходе работы комбинированного алгоритма распределение регистров, корректно.

Методом доказательства от противного легко показать, что для любого блока b его предусловия (и, аналогично, постусловия) не могут быть установлены при обработке двух различных точек синхронизации. Далее, из того, что при

обработке одной точки синхронизации всегда устанавливаются одни и те же граничные условия, следует корректность полученных граничных условий.

3. Выбор количества регистров для использования в граничных условиях

Для выбора граничных условий в точке синхронизации сначала необходимо ответить на вопрос, сколько регистров можно занять под граничные переменные.

Предположим, что нам известно точное количество регистров, которые необходимы для генерации кода для базового блока b — обозначим его RegistersNeeded(b). Общее количество регистров целевой архитектуры обозначим TotalRegisters. Тогда если выполняется условие

$$|\mathrm{b}^{\mathrm{post}}| + |\mathrm{b}^{\mathrm{pre}}| + \mathrm{RegistersNeeded}(\mathrm{b}) \leqslant \mathrm{TotalRegisters}, \quad (1)$$

то множества регистров $\mathrm{Regs}(\mathrm{b}^{\mathrm{pre}})$, $\mathrm{Regs}(\mathrm{b}^{\mathrm{post}})$ и множество регистров, используемых внутри базового блока, можно выбрать не пересекающимися. Иными словами, сокращение числа свободных регистров из-за их использования в пред- и постусловиях не ухудшит код, сгенерированный для базового блока.

Теперь немного ослабим ограничение (1) за счет того, что возьмем множества $\mathrm{Regs}(\mathrm{b}^{\mathrm{pre}})$ и $\mathrm{Regs}(\mathrm{b}^{\mathrm{post}})$ пересекающимися. Для этого опишем, как эффективно из распределения $\mathrm{b}^{\mathrm{pre}}$ получить распределение $\mathrm{b}^{\mathrm{post}}$. Далее, если выполнено условие

$$|\mathrm{b}^{\mathrm{post}}| + \mathrm{RegistersNeeded}(\mathrm{b}) \leqslant \mathrm{TotalRegisters}, \quad (2)$$

то можно вставить соответствующий код в начало базового блока b, а затем выбрать множества регистров $\mathrm{Regs}(\mathrm{b}^{\mathrm{post}})$ и используемых при генерации кода блока b непересекающимися. Аналогично можно поступить в случае, когда выполнено условие

$$|\mathrm{b}^{\mathrm{pre}}| + \mathrm{RegistersNeeded}(\mathrm{b}) \leqslant \mathrm{TotalRegisters} \quad (3)$$

Задачей переупорядочивания регистров назовем задачу генерации кода для пустого базового блока b с граничными условиями $\mathrm{b}^{\mathrm{pre}}$ и $\mathrm{b}^{\mathrm{post}}$. Алгоритм, который решает эту задачу назовем алгоритмом переупорядочивания регистров.

Утверждение Существует алгоритм переупорядочивания регистров генерирующий код, который использует только регистры из множества $\mathrm{R} \supseteq \mathrm{Regs}(\mathrm{b}^{\mathrm{pre}}) \cup \mathrm{Regs}(\mathrm{b}^{\mathrm{post}})$.

Доказательство Проведем доказательство конструктивно, то есть опишем как получить код, удовлетворяющий указанным условиям. Для этого будем использовать три операции:

- Spill(v, r) — сохранить содержимое регистра r в переменную v,

- Load(v, r) — загрузить значение переменной v в регистр r,
- Move(r1, r2) — скопировать содержимое регистра r2 в регистр r1.

Алгоритм будет выполняться в два этапа. На первом этапе все регистры освобождаются с помощью операции Spill. На втором нужные переменные загружаются в нужные регистры. **Ч.т.д.**

Приведенный алгоритм достаточен для доказательства утверждения, однако он генерирует избыточное количество сохранений в память и чтений из нее. Далее приведем более эффективный алгоритм генерации кода для такого базового блока, поскольку он будет использован в дальнейшем как часть комбинированного алгоритма распределения регистров.

Обозначим текущее распределение регистров $\mathrm{b}^{\mathrm{cur}}$. Изначально оно совпадает с $\mathrm{b}^{\mathrm{pre}}$. Построим ориентированный граф $\mathrm{G_r}$, вершинами которого будут являться регистры целевой архитектуры. Дуга $(\mathrm{r}_1, \mathrm{r}_2)$ присутствует в графе тогда и только тогда, когда существует переменная v, такая что $(\mathrm{v}, \mathrm{r}_1) \in \mathrm{b}^{\mathrm{cur}}$, $(\mathrm{v}, \mathrm{r}_2) \in \mathrm{b}^{\mathrm{post}}$ и $\mathrm{r}_1 \neq \mathrm{r}_2$. То есть содержимое регистра r_1 необходимо переместить в регистр r_2.

По определению граничных условий в графе $\mathrm{G_r}$ в каждую вершину входит не более одной дуги и выходит также не более одной дуги. Значит граф представляет собой совокупность цепочек и циклов.

Рассмотрим, как различные операции изменяют граф.

- Операция Spill может быть применена только к регистру, в котором хранится некоторая переменная. При этом дуга, выходящая из соответствующей вершины, исчезает, если она была.
- Операция Load может быть применена только к регистру, в котором не хранится никакая переменная. При этом появится дуга, выходящая из соответствующей вершины. Случай загрузки переменной, не входящей в множество $\mathrm{Var}(\mathrm{b}^{\mathrm{post}})$, рассматриваться не будет.
- Операция Move может быть применена только к паре регистров $(\mathrm{r}_1, \mathrm{r}_2)$ таких, что в r_1 не хранится никакая переменная, а в r_2 хранится некоторая переменная. При этом если из вершины r_2 выходила дуга $\mathrm{e} = (\mathrm{r}_2, \mathrm{r})$, то она исчезнет, а вместо нее добавится дуга $\mathrm{e}' = (\mathrm{r}_1, \mathrm{r})$.

Перейдем к описанию алгоритма. Он будет состоять из нескольких шагов.

1. Все регистры, содержащие переменные, не входящие во множество $\mathrm{Vars}(\mathrm{b}^{\mathrm{post}})$, освобождаются с помощью операций Spill. После этого шага все регистры, из которых в графе $\mathrm{G_r}$ не выходит дуги являются свободными.

2. До тех пор, пока существует пара регистров $(\mathrm{r}_1, \mathrm{r}_2)$, такая что в $\mathrm{G_r}$ есть дуга из r_2 в r_1 и нет дуги исходящей из r_1, к ним применяется операция Move$(\mathrm{r}_1, \mathrm{r}_2)$. В результате это операции исчезает дуга $(\mathrm{r}_2, \mathrm{r}_1)$. После завершения данного шага в графе $\mathrm{G_r}$ не останется цепочек.

3. До тех пор, пока в графе G_r существует цикл, он «разрывается», а шаг 2 повторяется. «Разорвать» цикл можно двумя способами:
    - переместив с помощью операции Move содержимое одного из регистров, входящих в цикл, в свободный;
    - сбросив содержимое одного из регистров, входящих в цикл, в память, с помощью операции Spill.

1. Второй способ имеет смысл применять только в том случае, если свободного регистра не нашлось (то есть на регистрах из множества R хранятся |R| различных переменных). Заметим, что сбрасывать переменную на этом шаге алгоритма потребуется не более одного раза, поскольку после этого на регистрах останется |R|-1 переменная, и свободный регистр всегда найдется.

2. После завершения предыдущего шага в графе G_r не осталось ни одной дуги. Осталось загрузить на регистры недостающие переменные (то есть переменные из множества $\mathrm{Vars}(b^{post}) \setminus \mathrm{Vars}(b^{cur})$). Все нужные регистры уже свободны, так как в графе G_r к этому моменту нет ни одной дуги.

Ни один из шагов алгоритма не увеличивает в графе G_r количество дуг. Каждая итерация шага 2 уменьшает количество дуг на 1. После каждой итерации шага 3 выполняется хотя бы одна итерация шага 2. Значит алгоритм конечен.

Приведем псевдокод полученного в ходе доказательства леммы алгоритма. Будем считать, что граф G_r уже построен и что операции Spill, Load и Move корректно его обновляют. Операция Break-Cycle «разрывает» цикл в графе одним из приведенных в описании шага 3 способов.

```
procedure Reg-Reorder(b^pre, b^post, G_r)
      for all (v,r) ∈ b^pre: v ∈ Vars(b^pre) \ Vars(b^post) do
            Spill(v, r)
      end for
      while ∃(u,v) ∈ Edges(G_r): u ≠ v do
            for all
            (r_2,r_1) ∈ Edges(G_r): ∄r_3: (r_1,r_3) ∈ Edges(G_r) do
                  Move(r_1,r_2)
            end for
            if ∃c ∈ Cycles(G_r) then
                  Break-Cycle(C)
            end if
      end while
```

```
    for all (v,r) ∈ b^post: v ∈ Vars(b^post) \ Vars(b^pre) do
        Load(v, r)
    end for
end procedure
```

Посчитаем, сколько операций загрузки (L), сохранения (S) и пересылки (M) регистров генерирует данный алгоритм. Для этого сначала определим, в каком случае на шаге 3 алгоритма приходится прибегать к сбросу содержимого регистра в память. В момент выполнения шага 3 на регистрах могут находиться только переменные из множества $\mathrm{Vars}(b^{pre}) \cap \mathrm{Vars}(b^{post})$. Значит

$$|\mathrm{Vars}(b^{pre}) \cap \mathrm{Vars}(b^{post})| \geqslant |R| \geqslant |\mathrm{Regs}(b^{pre}) \cup \mathrm{Regs}(b^{post})|.$$

С другой стороны

$$|\mathrm{Vars}(b^{pre}) \cap \mathrm{Vars}(b^{post})| \leqslant |\mathrm{Vars}(b^{pre})| = |\mathrm{Regs}(b^{pre})| \leqslant \leqslant |\mathrm{Regs}(b^{pre}) \cup \mathrm{Regs}(b^{post})|,$$

$$|\mathrm{Vars}(b^{pre}) \cap \mathrm{Vars}(b^{post})| \leqslant |\mathrm{Vars}(b^{post})| = |\mathrm{Regs}(b^{post})| \leqslant \leqslant |\mathrm{Regs}(b^{pre}) \cup \mathrm{Regs}(b^{post})|.$$

Такое возможно только если во всех нестрогих неравенствах достигается равенство. То есть

$$|\mathrm{Vars}(b^{pre}) \cap \mathrm{Vars}(b^{post})| = |\mathrm{Vars}(b^{pre})| = |\mathrm{Vars}(b^{post})| \Rightarrow \mathrm{Vars}(b^{pre}) = \mathrm{Vars}(b^{post}),$$

$$|\mathrm{Regs}(b^{pre}) \cup \mathrm{Regs}(b^{post})| = |R| = |\mathrm{Regs}(b^{pre})| = |\mathrm{Regs}(b^{post})| \Rightarrow \mathrm{Regs}(b^{pre}) = \mathrm{Regs}(b^{post}) = R.$$

Возможны два случая.

- Если $b^{pre} = b^{post}$, то алгоритм переупорядочивания на шаге 3 не будет генерировать дополнительных сбросов в память.
- Если $b^{pre} \neq b^{post}$, то граф G_r будет представлять собой совокупность циклов и алгоритм переупорядочивания регистров на шаге 3 сгенерирует один дополнительный сброс в память.

Таким образом, сброс содержимого регистра в память на шаге 3 алгоритма переупорядочивания регистров происходит тогда и только тогда, когда

$$b^{pre} \neq b^{post} \wedge \mathrm{Vars}(b^{pre}) = \mathrm{Vars}(b^{post}) \wedge \mathrm{Regs}(b^{pre}) = \mathrm{Regs}(b^{post}) = R \quad (4)$$

Вернемся к вычислению величин L, S и M.

На первом шаге алгоритма произойдет $|\mathrm{Vars}(b^{pre}) \setminus \mathrm{Vars}(b^{post})|$ операций Spill.

На втором шаге алгоритма произойдет $|\mathrm{Edges}(G_r)| - 1$ операций Move в случае выполнения условия (4), либо $|\mathrm{Edges}(G_r)|$ в противном случае.

На третьем шаге алгоритма произойдет $|\mathrm{Cycles}(\mathrm{G_r})| - 1$ операций Move и одна операция Spill в случае выполнения условия (2), либо $|\mathrm{Cycles}(\mathrm{G_r})|$ операций Move в противном случае.

На четвертом шаге алгоритма произойдет $|\mathrm{Vars}(\mathrm{b^{post}}) \setminus \mathrm{Vars}(\mathrm{b^{pre}})| + 1$ операций Load в случае выполнения условия (4), либо $|\mathrm{Vars}(\mathrm{b^{post}}) \setminus \mathrm{Vars}(\mathrm{b^{pre}})|$ в противном случае.

Значит, если выполнено условие (4), то

$$\mathrm{L} = |\mathrm{Vars}(\mathrm{b^{post}}) \setminus \mathrm{Vars}(\mathrm{b^{pre}})| + 1 = 1,$$
$$\mathrm{S} = |\mathrm{Vars}(\mathrm{b^{pre}}) \setminus \mathrm{Vars}(\mathrm{b^{post}})| + 1 = 1,$$
$$\mathrm{M} = |\mathrm{Edges}(\mathrm{G_r})| + |\mathrm{Cycles}(\mathrm{G_r})| - 2.$$

Иначе

$$\mathrm{L} = |\mathrm{Vars}(\mathrm{b^{post}}) \setminus \mathrm{Vars}(\mathrm{b^{pre}})|,$$
$$\mathrm{S} = |\mathrm{Vars}(\mathrm{b^{pre}}) \setminus \mathrm{Vars}(\mathrm{b^{post}})|,.$$
$$\mathrm{M} = |\mathrm{Edges}(\mathrm{G_r})| + |\mathrm{Cycles}(\mathrm{G_r})|$$

Пусть есть два алгоритма переупорядочивания регистров. Алгоритм $\mathrm{A_1}$ генерирует $\mathrm{L_1}$ операций Load, $\mathrm{S_1}$ операций Spill и $\mathrm{M_1}$ операций Move. Алгоритм $\mathrm{A_2}$ генерирует $\mathrm{L_2}$ операций Load, $\mathrm{S_2}$ операций Spill и $\mathrm{M_2}$ операций Move. Будем считать, что алгоритм $\mathrm{A_1}$эффективнее алгоритма $\mathrm{A_2}$тогда и только тогда, когда $\mathrm{L_1} + \mathrm{S_1} < \mathrm{L_2} + \mathrm{S_2}$ либо $\mathrm{L_1} + \mathrm{S_1} = \mathrm{L_2} + \mathrm{S_2}$ и $\mathrm{M_1} < \mathrm{M_2}$.

Утверждение Описанный алгоритм переупорядочивания регистров является наиболее эффективным среди всех алгоритмов переупорядочивания регистров, использующих только регистры из множества $\mathrm{R} \supseteq \mathrm{Regs}(\mathrm{b^{pre}}) \cup \mathrm{Regs}(\mathrm{b^{post}})$.

Доказательство (от противного)

Пусть существует алгоритм A', который эффективнее описанного. Описанный алгоритм генерирует L операций Load, S операций Spill и M операций Move. Алгоритм A' генерирует L' операций Load, S' операций Spill и M' операций Move. Отдельно рассмотрим два случая: когда условие (4) выполняется, и когда оно не выполняется.

Предположим, что условие (4) не выполняется. Поскольку переменные из множества $\mathrm{Vars}(\mathrm{b^{pre}}) \setminus \mathrm{Vars}(\mathrm{b^{post}})$ должны быть сохранены, то

$$\mathrm{S}' \geqslant |\mathrm{Vars}(\mathrm{b^{pre}}) \setminus \mathrm{Vars}(\mathrm{b^{post}})| = \mathrm{S}$$

Аналогично

$$\mathrm{L}' \geqslant |\mathrm{Vars}(\mathrm{b^{post}}) \setminus \mathrm{Vars}(\mathrm{b^{pre}})| = \mathrm{L}$$

Поскольку $\mathrm{L}' + \mathrm{S}' \leqslant \mathrm{L} + \mathrm{S}$, во всех неравенствах достигается равенство. Значит в алгоритме A' никаких сохранений, кроме сохранений переменных из множества $\mathrm{Vars}(\mathrm{b^{pre}}) \setminus \mathrm{Vars}(\mathrm{b^{post}})$ нет. Данные сохранения не меняют граф $\mathrm{G_r}$. Значит количество дуг и циклов в графе $\mathrm{G_r}$, может уменьшаться только за

счет операций Move. Так как целевой регистр операции Move должен быть свободным, каждая операция может либо уменьшать количество циклов в графе на 1, либо уменьшать количество ребер в графе на 1. Значит

$$M' \geqslant |\mathrm{Edges}(G_r)| + |\mathrm{Cycles}(G_r)| = M.$$

Это противоречит тому, что алгоритм A' эффективнее описанного алгоритма.

Осталось рассмотреть второй случай. Пусть условие (4) выполняется.

Поскольку все регистры из множества R в начале работы алгоритма заняты, первой инструкцией сгенерированного кода может быть только операция Spill, примененная к одной из переменных из множества $\mathrm{Vars}(b^{post})$. Значит $S' \geqslant 1$. Однако эта переменная в конце должна располагаться на регистре. Значит она будет загружена с помощью Load, то есть $L' \geqslant 1$. Так как $L' + S' \leqslant L + S = 2$, то $L' = 1$, $S' = 1$ и $L' + S' = L + S$.

Каждая операция Spill может уменьшить количество циклов графа G_r не более чем на 1. Аналогично она может уменьшить количество дуг не более чем на 1. Тогда аналогично предыдущему случаю

$$\begin{gathered} M' \geqslant |\mathrm{Edges}(G_r)| - 1 + |\mathrm{Cycles}(G_r)| - 1 = \\ |\mathrm{Edges}(G_r)| + |\mathrm{Cycles}(G_r)| - 2 = M \end{gathered}.$$

Это противоречит тому, что алгоритм A' эффективнее описанного алгоритма.

Величина RegistersNeeded(b), используемая в условиях (2) и (3), априори неизвестна и не может быть легко вычислена. Оценим ее приближенно. Для этого введем понятие регистрового давления.

Регистровым давлением в инструкции I из базового блока b будем называть минимальное количество регистров, необходимых для генерации кода данной инструкции в предположении, что все переменные, которые живы в данной точке базового блока и используются в инструкции I или после нее, располагаются на регистрах.

$$\mathrm{RegPressure}(I, b) = |\mathrm{LiveVariables}(I, b)| + |\mathrm{ExtraRegisters}(I)|$$

В этом определении множество живых переменных LiveVariables(I,b) может быть взято из результатов анализа времени жизни переменных. Множество дополнительных регистров ExtraRegisters(I), которые нужны инструкции I, зависит только от типа самой инструкции. Так, например, для вызова функции это множество будет состоять из регистров, которые могут быть испорчены вызываемой функцией, регистра, в котором хранится возвращаемое значение, и регистров, которые будут использованы для передачи параметров.

Регистровым давлением в базовом блоке b назовем максимальное среди регистровых давлений во всех его инструкциях.

$$\mathrm{RegPressure}(b) = \max_{I \in b}(\mathrm{RegPressure}(I, b))$$

Регистровое давление является оценкой сверху на величину RegistersNeeded, поэтому перепишем условия (2) и (3) используя новое определение:

$$|b^{post}| \leqslant \mathrm{TotalRegs} - \mathrm{RegPressure}(b), \quad (5)$$

$$|b^{pre}| \leqslant \mathrm{TotalRegs} - \mathrm{RegPressure}(b) \quad (6)$$

Теперь, если использовать для граничных условий в точке синхронизации J не более

$$\mathrm{TotalRegs} - \max_{e \in J}(\mathrm{RegPressure}(\mathrm{src}(e)), \mathrm{RegPressure}(\mathrm{dst}(e)))$$

регистров, то одно из условий (5) и (6) будет обязательно выполнено во всех прилегающих к J базовых блоках.

Включение переменной в граничные условия позволяет ей пересекать границы базовых блоков на регистрах и избежать лишнего сброса этой переменной в память с последующей загрузкой ее из памяти, если она используется в нескольких базовых блоках.

Для выбора конкретных переменных для граничных условий введем функцию полезности включения переменной x в граничные условия точки синхронизации J: Usefullness(x,J). Функция полезности будет вычислять по следующей формуле:

$$\mathrm{Usefullness}(x, J) = |e{:}\ e \in J \wedge x \in \mathrm{Vars}(\mathrm{src}(e)) \wedge x \in \mathrm{Vars}(\mathrm{dst}(e))|.$$

Данная формула описывает, сколько ребер входит в точку синхронизации таких, что переменная x используется как в базовом блоке из которого данное ребро исходит, так и в базовом блоке в которое данное ребро входит.

Итоговый алгоритм вычисления граничных условий в точке синхронизации J выглядит следующим образом.

```
function Compute-Register-Mapping(J)
        p ← max_{e∈J}(RegPressure(src(e)), RegPressure(dst(e)))
        n ← TotelRegs − p
        result ← ∅
        priority ←Compute-Usefullness(J)
        for i ← 1, ndo
                result ← result ∪ {(priority[i], register[i])}
        end for
        return result
end function
```

4. Экспериментальные результаты

Описанный алгоритм был реализован в QEMU версии 1.0. При хранении графа потока управления учитывается такая особенность QEMU, что из базового блока может исходить не более двух дуг. Вычисление точек синхронизации, сбор информации о регистровом давлении в базовом блоке, вычисление функции полезности переменной в точке синхронизации и запись граничных условий делается с помощью обхода графа $G_E = \langle E, F \rangle$ в глубину. Строить его в явном виде не нужно. Все вершины, смежные с вершиной e=(src,dst) графа G_E, можно найти просмотрев все исходящие из вершины src графа G дуги, и все входящие в dst.

Тестирование данного алгоритма начнем с модельного примера. В качестве такого примера возьмем последовательность инструкций архитектуры ARM

```
addlt       r1, r1, r2
sub         r2, r2, r1
subgt       r1, r1, r2
add         r2, r2, r1
```

выполненную в цикле $b0000000_{16}$ раз. Условные инструкции обеспечат наличие нескольких базовых блоков внутри одного блока трансляции. Гостевые регистры r1 и r2 станут глобальными переменными, используемыми в нескольких базовых блоках. Выполнение данного фрагмента большое количество раз в цикле позволит провести измерение времени. Результаты тестирования приведены в таблице 1. Ускорение получается за счет того, что переменные, соответствующие гостевым регистрам r1 и r2 в случае глобального распределения регистров загружаются на регистры основной машины один раз в начале обработки инструкции addlt, а в случае локального — перед обработкой каждой инструкции. Таким образом экономится по 6 загрузок и сохранений на каждом выполнении приведенного фрагмента.

Табл. 1. Результаты тестирования глобального распределения на модельном примере

Table 1. Global register allocation testing results for model example

Без глобального распределения регистров (секунд)	С глобальным распределением регистров (секунд)	Ускорение
16.644	11.715	29.6%

Для того, чтобы определить изменение производительности на реальных программах были использованы тесты из набора SPEC CINT2000. Тестирование на них показало небольшое падение производительности на большинстве из тестов. Результаты тестирования приведены в таблице 2. На момент написания данной статьи удалось установить две существенные

причины падения производительности, которые планируется устранить в ходе дальнейших работ над данным алгоритмом.

Первая из них связана с применимостью данного алгоритма. Для того, чтобы он был применен, необходимо чтобы

- блок трансляции состоял из нескольких базовых блоков,
- некоторые глобальные переменные использовались в нескольких базовых блоках.

Табл. 2. Результаты тестирования глобального распределения на тестах из набора SPEC CINT2000

Table 2. Global register allocation testing results for SPEC CINT2000 benchmarks

Тест	Без глобального распределения регистров (секунд)	С глобальным распределением регистров (секунд)	Ускорение
164.gzip	81.004	81.840	-1%
175.vpr	144.56	148.164	-2.5%
256.bzip2	42.496	40.580	4.5%
300.twolf	36.508	36.544	0%

Табл. 3. Результаты профилирования глобального распределения решистров

Table 3. Profiling results for global register allocation

Тест	Всего блоков трансляции	Блоков трансляции состоящих из нескольких базовых блоков	Блоков трансляции, на которых произошло глобальное распределние регистров
164.gzip	3526	651	365
175.vpr	8016	1447	801
256.bzip2	3003	689	388
300.twolf	8221	1631	911

Как удалось выяснить с помощью профилирования, таких блоков всего около 10% от общего числа блоков трансляции. Результаты профилирования для трех тестов приведены в таблице 3. Данная проблема может быть устранена за

счет увеличения блоков трансляции таким образом, чтобы они могли включать несколько гостевых базовых блоков.

Вторая причина связана с недостаточным учетом граничных условий при локальном распределении регистров. Инициализации начальными условиями и обеспечения выполнения конечных условий недостаточно. Необходимо также отдавать предпочтение регистру из граничных условий при выборе регистра для переменной, а также по возможности не занимать регистры из постусловий под переменные в них не входящие. Для устранения данной причины необходимо внести дополнительные модификации в существующий алгоритм локального распределения регистров.

Список литературы

[1]. Батузов К. Задача локального распределения регистров во время динамической двоичной трансляции. Труды ИСП РАН, том 22, 2012 г., стр. 67-76. DOI: 10.15514/ISPRAS-2012-22-5

[2]. Chaitin G. J. Register allocation & spilling via graph coloring. Proceedings of the 1982 SIGPLAN symposium on Computer construction. SIGPLAN '82. New York, USA: ACM, 1982, pp. 98-105.

[3]. Poletto Massimiliano Sarkar Vivek. Linear san regsiter allocation. ACM Transaction on Programming Languages and Systems. 1999 Vol. 21, no 5, pp 895-913.

[4]. Bellard Fabrice. QEMU, a fast and portable dynamic translator. Proceedings of the annual conference on USENIX Annual Technical Conference. ATEC '05. Berkley, USA: USENIX Association, 2005, pp. 41-46.

Global register allocation during dynamic binary translation

K.A. Batuzov <batuzovk@ispras.ru>

[1] *Institute for System Programming of the Russian Academy of Sciences,*

25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. Register allocation have a significant impact on performance of generated code. This paper explores the problem of global register alloction during dynamic binary translation. Since existing algorithms are designed for compilers and they are not always suitable for dynamic binary translation, a new global register allocation was developed. This algorithm decides which variables should reside on which registers before and after each basic block (called pre- and post- conditons of this basic block) and solves local register allocation problem in these constraints. To ensure that pre- and post- conditions of different basic blocks are consistent the algorithms must choose these conditions in such a way that for each basic block b' that precides arbitrary block b it's postconditions are the same as preconditions of b. This can be achieved by finding groups of arcs in control flow graph on which these conditions should remain the same (let's call them synchronisation points) and then choosing conditions for each such synchronisation point. Synchronization points are

connected components in graph G_E which is a graph where arcs of original CFG are vertices and edges connect arcs which start or end in the same basic block. This gives an efficient way to find synchronization points. To choose which variables should reside on registers in each synchronization point the amount of available register is estimated using register pressure in incident basic blocks. Then actual variables a picked based on how often they are used in incident basic blocks. Finally the local register allocation algorithm is modified to use precondition and ensure post conditions of the basic block. The efficient algorithm to convert existing allocation to the desired postcondition at the end of basick block is presented with the proof of that it's optimal in terms of generated spills, reloads and register moves. The described algorithm showed 29.6% running time decrease on the synthetic example. The needed modifications of the algorithm to efficiently run on real life application are explored.

Keywords: global register allocation, dynamic binary translation, QEMU..

DOI: 10.15514/ISPRAS-2016-28(5)-12

For citation: K.A. Batuzov. Global register allocation during dynamic binary translation. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 5, 2016, pp. 199-214 (in Russian). DOI: 10.15514/ISPRAS-2016-28(5)-12

References

[1]. Batuzov K. Local register allocation during dynamic binary translation. Trudy ISP RAN/Proc. ISP RAS, vol. 22, 2012, pp. 67-76 (in Russian). DOI: 10.15514/ISPRAS-2012-22-5
[2]. Chaitin G. J. Register allocation & spilling via graph coloring. Proceedings of the 1982 SIGPLAN symposium on Computer construction. SIGPLAN '82. New York, USA: ACM, 1982, pp. 98-105.
[3]. Poletto Massimiliano Sarkar Vivek. Linear san regsiter allocation. ACM Transaction on Programming Languages and Systems. 1999 Vol. 21, no 5, pp 895-913.
[4]. Bellard Fabrice. QEMU, a fast and portable dynamic translator. Proceedings of the annual conference on USENIX Annual Technical Conference. ATEC '05. Berkley, USA: USENIX Association, 2005, pp. 41-46.

Платформенно-независимый и масштабируемый инструмент поиска клонов кода в бинарных файлах*

А.К. Асланян <hayk@ispras.ru>
Ш.Ф. Курмангалеев <kursh@ispras.ru>
В.Г. Варданян <vaag@ispras.ru>
М.С. Арутюнян <arutunian@ispras.ru>
С.С. Саргсян <sevaksargsyan@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. При разработке программного обеспечения разработчики часто прибегают к копированию того или иного участка кода для достижения желаемого результата. Копирование кода может привести к появлению различных ошибок, а также к увеличению размера исходного и бинарного кода. Задача поиска семантически сходных участков кода (клонов) в бинарных файлах становится более актуальной в связи с недоступностью исходного кода многих программных средств. В данной статье обсуждаются существующие методы поиска клонов бинарного кода и приводится описание разработанного нами инструмента обнаружения клонов в бинарном коде. Работа инструмента разделена на три основных этапа. Первый этап базируется на платформе Binnavi [1] и ответственен за генерацию графов зависимостей программы для каждой функции. В качестве основы для генерации графов используется платформенно-независимый язык REIL (Reverse Engineering Intermediate Language). Использование языка REIL позволяет генерировать графы сразу для нескольких целевых архитектур (x86, x86-64, ARM, MIPD, PPC), тем самым обеспечивает независимость инструмента от целевой архитектуры. На втором этапе производится поиск клонов на основе ранее созданных графов. Для каждой пары графов строится наибольший общий подграф, на основе которого определяются клоны бинарного кода. На третьем этапе полученные клоны визуализируются для удобного анализа полученных результатов.

Ключевые слова: клоны кода; семантический анализ бинарного кода; REIL; граф зависимостей программы.

DOI: 10.15514/ISPRAS-2016-28(5)-13

---

* Работа поддержана грантом РФФИ № 15-07-07541 А

Для цитирования: А.К. Асланян, Ш.Ф. Курмангалеев, В.Г. Варданян, М.С. Арутюнян, С.С. Саргсян. Платформенно-независимый и масштабируемый иструмент поиска клонов бинарного кода. Труды ИСП РАН, том 28, вып. 5, стр. 215-226, 2016 г. DOI: 10.15514/ISPRAS-2016-28(5)-13

1. Введение

Существует ряд методов поиска клонов кода, основанный на текстовом [2], лексическом [3], синтаксическом [4, 5, 6] и семантическом [6, 7, 8, 9, 10, 11, 12 13] анализе программы. Но, в основном, все эти методы основаны требуют наличия исходного кода программы. Задача поиска клонов в бинарном коде мало изучена несмотря на то, что она является более важной с точки зрения поиска ошибок в программах, учитывая тот факт, что в основном программы распространяются без исходного кода. Качественный инструмент поиска клонов бинарного кода может быть применен в задачах поиска необновленных фрагментов и вирусов в исполняемых файлах.

Клоны бинарного кода условно разделены на три основные типы. Первый тип – фрагменты кода, которые полностью совпадают. Второй тип – фрагменты кода, которые могут отличаться типами, значениями данных именами регистров. Третий тип – фрагменты кода, которые могут отличаться типами, значениями данных именами регистров, а также могут отличаться некоторыми инструкциями (в конкретном фрагменте могут присутствовать или отсутствовать некоторые инструкции).

В рис. 1 приведены примеры клонов кода в ассемблерной форме (для архитектуры x86). Клон первого типа совпадает с конкретным фрагментом. Клон второго типа от конкретного фрагмента отличается распределением регистра *ecx* в место *eax*. Клон третьего типа от конкретного фрагмента отличается распределением регистра *ecx* в место *eax* и отсутствием одной инструкции (*imul eax, ebp+var_4]*).

2. Подходы поиска клонов в бинарном коде

Существует несколько работ, посвященные поиску клонов в бинарном коде. Алгоритм, названный BitShred [14], основан на «отпечатках» фрагментов, использует фильтр для поиска ошибок, появившихся в результате дублирования кода. Фильтр – структура данных, позволяющая проводить проверку принадлежности элемента набору отпечатков. Алгоритм BitShred состоит из 3 этапов: разбиения файла, создания «отпечатков» для полученных фрагментов и сравнения этих «отпечатков» между собой. Для «отпечатков» считаются хеш коды и если они совпадают, то соответствующие им фрагменты считаются клонами. Алгоритм находит клоны только первого типа.

Фрагмент кода	**Клон типа 1**	**Клон типа 2**	**Клон типа 3**

public main
main proc near
 	
var_4= dword ptr -4
argc= dword ptr 8
argv= dword ptr 0Ch
envp= dword ptr 10h
 	
push ebp
mov ebp, esp
mov [ebp+var_4], 5
mov eax,[ebp+var_4]
imul eax,[ebp+var_4]
leave
retn
main endp
	public main
	main proc near
	
	var_4= dword ptr -4
	argc= dword ptr 8
	argv= dword ptr 0Ch
	envp= dword ptr 10h
	
	push ebp
	mov ebp, esp
	mov [ebp+var_4], 5
	mov eax,[ebp+var_4]
	imul eax,[ebp+var_4]
	leave
	retn
	main endp
		public main
		main proc near
		
		var_4= dword ptr -4
		argc= dword ptr 8
		argv= dword ptr 0Ch
		envp= dword ptr 10h
		
		push ebp
		mov ebp, esp
		mov [ebp+var_4], 10
		mov ecx, [ebp+var_4]
		imul ecx, [ebp+var_4]
		leave
		retn
		main endp
			public main
			main proc near
			
			var_1= dword ptr -4
			argc= dword ptr 8
			argv= dword ptr 0Ch
			envp= dword ptr 10h
			
			push ebp
			mov ebp, esp
			mov [ebp+var_1], 15
			mov ecx, [ebp+var_1]
			
			leave
			retn
			main endp

Рис. 1. Примеры типов клонов для архитектуры x86 (соответствующий ассемблер)

Fig. 1. Examples of clones types for x86 architecture (corresponding assembler)

A. Schulman [15] предложил систему, которая создаёт хеш для каждой функции в бинарном файле. Совпадающие хеши, встретившиеся более чем в одном файле указывают на клон кода. Алгоритм позволяет находить клоны только первого типа, так как каждый раз компилятор может разместить регистры по-разному.

Система, созданная D. Bruschi и др. [16], находит клоны в бинарных файлах для обнаружения вредоносных программ. Сначала бинарный файл дизассемблируется и нормализуется, после чего удаляется мертвый код и происходит разделение кода на фрагменты. Для каждого фрагмента строится метрика, основанная на графе потока управления. На последнем этапе происходит поиск клонов, при помощи сравнения полученных метрик.

Sæbjørnsen и др. [17] после получения ассемблера из бинарного файла, нормализируют и считают характеризующие векторы. На основе сравнивания этих векторов становится возможным нахождение клонов типов T1, T2 и T3. На основе этой работы, M. Farhadi и др. [18] создали систему для обнаружения клонов вредоносного кода в программах.

T. Dullien и др. [19] предложили систему сравнения бинарных файлов на основе структурного анализа, для поиска вредоносного кода. Алгоритм состоит из двух этапов: генерация признаков вредоносного кода и распознавание схожести между различными участками кода на основе графа потока управления.

3. Модель инструмента поиска клонов в бинарных файлах

Предлагаемый модель инструмента для поиска клонов бинарного кода был разработан с учетом следующих требований:

- нахождение T1, T2 и T3 типов клонов;
- независимость от целевой архитектуры;
- масштабируемость: размер анализируемых программ может достигать десятков МБ;
- большой процент истинных срабатываний (> 90%).

При анализе кода, учитываются параметры, задаваемые пользователем: минимальное число инструкций для конечных клонов (МЧИ) и минимальный процент схожести (МПС) клонов.

Работа инструмента делится на три основных этапа:

- Первый этап базируется на платформе Binnavi (инфраструктура для анализа бинарных файлов. В качестве инструмента для восстановления структур и потока управления программы используется дизассемблер Ida Pro [20]). На этом этапе происходит трансляция из машинного кода в представление REIL из которого, в свою очередь, генерируются ГЗП (граф зависимостей программы) для каждой функции. Узлам ГЗП соответствуют инструкции REIL, а ребрам – зависимости по данным и по управлению. В конце первого этапа все графы сериализуются.
- На втором этапе производится поиск клонов ассемблерного кода, учитывая параметры МЧИ и МПС.
- На третьем этапе демонстрируются полученные клоны и соответствующие им графы.

Промежуточная сериализация и десериализация дает два основных преимущества. Первое – для некоторых проектов работа с созданными графами может потребовать большое количество оперативной памяти. Сериализация графов позволяет извлечь и сравнивать графы парами, тем самым сэкономив память. Второе - возможность многократного применения второго этапа с различными значениями для параметров МЧИ и МПС.

На рис.2 приведена схема работы инструмента:

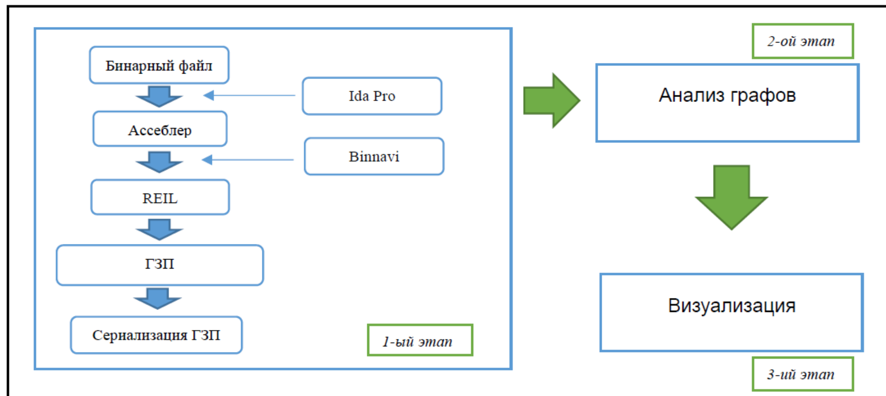


Рис. 2. Схема работы инструмента

Fig. 2. The tool architecture

3.1 Генерация ГЗП

Для генерации графов зависимостей программы используются средства и возможности платформы Binnavi. Binnavi предоставляет интерфейс для генерации и использования разных промежуточных представлений программы, основанных на языке REIL, в том числе, генерация графа потока управления, генерация графа вызовов функций и графа зависимостей по данным. В рамках разработки инструмента в платформу Binnavi был добавлен новый функционал, который позволяет для каждой функции автоматически генерировать граф потока управления и граф зависимостей по данным и объединить их в единый граф зависимостей программы (рис. 3). После чего, созданные графы сохраняются для дальнейшего анализа.

3.2 Разделение ГЗП на подграфы

После загрузки ГЗП, они могут быть разделены на единицы сравнения (ЕС). ЕС-ы представляют собой подграфы ГЗП и рассматриваются как потенциальные клоны друг друга. Так как графы изначально генерируются для функций, то для сравнения кода в рамках каждой функции необходимо разделение графов. Разработаны два метода для решения этой задачи: разделение, учитывая базовые блоки кода и разделение по слабо связанным компонентам графа. Разделение графов также позволяет увеличить эффективность алгоритмов анализа ГЗП (алгоритмы поиска максимальных изоморфных подграфов)

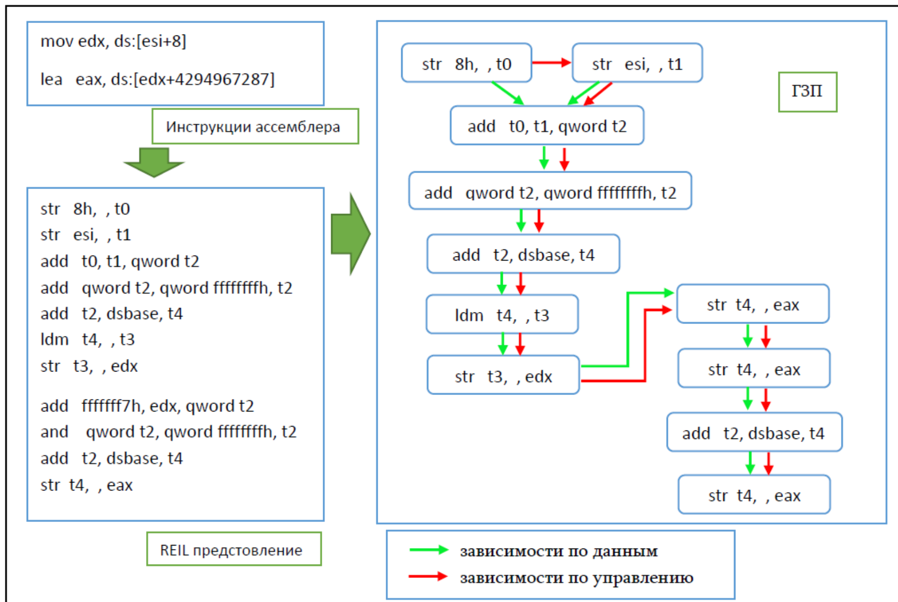


Рис. 3. Пример генерации ГЗП

Fig. 3. Example of PDG generation

3.3 Анализ ГЗП графов

На этом шагу происходит анализ ЕС для нахождения клонов кода. После загрузки ЕС они попарно сравниваются. Для каждой пары строится наибольший общий подграф. Как известно, эта проблема в общем случае NP-полная и для решения проблемы используется приближенный алгоритм. В большинстве случаев созданные графы разреженные, что и используется для более эффективного нахождения наибольшего общего подграфа.

На первом этапе алгоритма производится фильтрация пар ГЗП по параметру МЧИ. Сложность этого этапа составляет $O(n \cdot \log_2 n)$, где n - количество вершин в обеих ЕС. С помощью этого этапа эффективность алгоритма значительно повышается. Вторая часть алгоритма находит наибольший общий подграф и проверяет удовлетворение параметра МПС. Сначала сопоставляются те вершины в графах, которые не имеют входных ребер. В следующих итерациях рекурсивно рассматриваются и сопоставляются вершины связаные ребром с вершинами, которые в предыдущих итерациях сопоставлялись. Сложность этой части алгоритма не превышает $O(n^3)$, где n - количество вершин в обеих ЕС. После нахождения наибольшего общего подграфа, восстанавливаются фрагменты.

3.4 Фильтрация полученных клонов

Так как ГЗП строится из представления REIL, в некоторых случаях полученные машинные инструкции, соответствующие изоморфным подграфам, могут быть сильно разбросаны и не удовлетворять условию МПС. По этой причине возникает дополнительная необходимость фильтрации полученных клонов.

3.5 Визуализация полученных клонов

Последний этап работы инструмента – визуализация клонов. Цель созданного графического интерфейса - демонстрация ассемблерного кода полученных клонов, процент их схожести, а также соответствующие им графы (рис. 4).

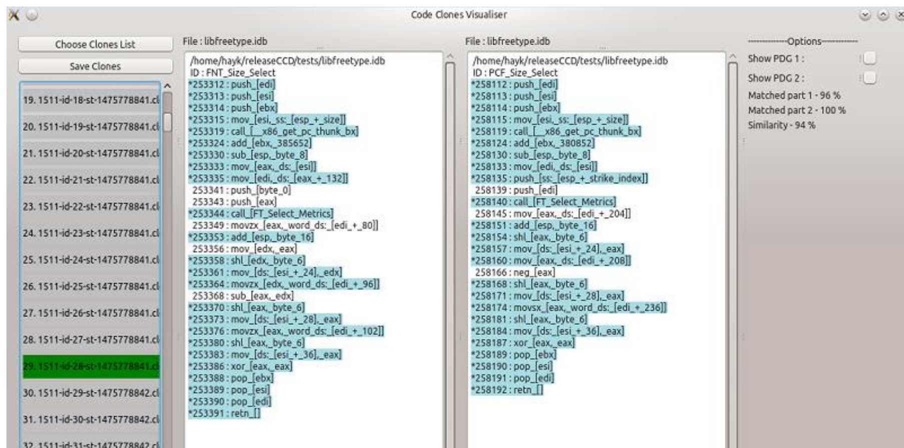


Рис. 4. Демонстрация результатов

Fig. 4. Visualization of results

4. Результаты

Для оценки эффективности инструмента произведено тестирование на разных тестовых и реальных проектах. Целевая машина – Intel Xeon, 40 ядер, ОЗУ - 128 ГБ.

Результаты приведены на таб. 1 (бинарные файлы получены компиляцией исходного кода с флагами -O0 –fno-inline) и таб. 2 (бинарные файлы получены компиляцией исходного кода с флагами –O3 –fno-inline), где МЧИ равно 30, МПС равно 90%, сравнивались функции для каждого бинарного файла между собой.

Табл. 1. Результаты тестирования

Table 1. Results

Бинарный файл	Размер бинарного файла	Количество найденных клонов	Время работы инструмента
libxml2.so	1.8 МБ	2123	1м.46 с.
libfreetype.so	816 КБ	270	57 с.
openssl	764 КБ	40	35 с.
d8	33 МБ	40397	26 м.52 с.

Табл. 2. Результаты тестирования

Table 2. Results

Бинарный файл	Размер бинарного файла	Количество найденных клонов	Время работы инструмента
libxml2.so	1.8 МБ	437	1м.25 с.
libfreetype.so	740 КБ	73	36 с.
openssl	672 КБ	71	26 с.
d8	20 МБ	19453	17 м.26 с.

5. Дальнейшая работа

Планируется применить инструмент для поисков фрагментов кода с ошибками и вредоносным кодом. При наличии базы вирусов или фрагментов кода, которые содержат ошибки, инструмент может найти соответствующие им клоны, которые с большой вероятностью тоже содержат ошибки или вредоносный код. Инструмент также может применяться для автоматического поиска однотипных ошибок.

6. Заключение

В данной работе рассмотрены существующие подходы поиска клонов для бинарного кода. Разработан платформенно-независимый, масштабируемый инструмент, который позволяет находить все три типа клонов кода, и обладает

высокой точностью. Результаты тестирования показали, что все критерии, поставленные нами, удовлетворяются.

Список литературы

[1]. https://www.zynamics.com/binnavi.html

[2]. Ducasse S., Rieger M., Demeyer S., A language independent approach for detecting duplicated code, in: Proceedings of the 15th International Conference on Software Maintenance, 1999, pp. 109-119, DOI: 10.1109/ICSM.1999.792593.

[3]. Kamiya T., Kusumoto S., Inoue K., CCFinder: A multilinguistic tokenbased code clone detection system for large scale source code, IEEE Transactions on Software Engineering, 2002, vol. 28, no. 7, pp. 654-670, DOI: 10.1109/TSE.2002.1019480.

[4]. Baxter I., Yahin A., Moura L., Anna M., Clone detection using abstract syntax trees, in: Proceedings of the 14th IEEE International Conference on Software Maintenance, IEEE Computer Society, 1998, pp. 368-377, DOI: 10.1109/ICSM.1998.738528.

[5]. Tairas R., Gray J., Phoenix-based clone detection using suffix trees, in: Proceedings of the 44th Annual Southeast Regional Conference, 2006, pp. 679-684, DOI: 10.1145/1185448.1185597.

[6]. Jiang L., Misherghi G., Su Z., Glondu S., DECKARD : Scalable and accurate tree-based detection of code clones, in: Proceedings of the 29th International Conference on Software Engineering, IEEE Computer Society, 2007, pp. 96-105, DOI: 10.1109/ICSE.2007.30.

[7]. Komondoor R., Horwitz S., Using slicing to identify duplication in source code, in: Proceedings of the 8th International Symposium on Static Analysis, 2001, pp. 40-56, DOI: 10.1007/3-540-47764-0_3.

[8]. Krinke J., Identifying similar code with program dependence graphs, in: Proceedings of the 8th Working Conference on Reverse Engineering, 2001, pp. 301-309, DOI: 10.1109/WCRE.2001.957835.

[9]. Gabel M., Jiang L., Su Z., Scalable detection of semantic clones, in: Proceedings of 30th International Conference on Software Engineering, 2008, pp. 321-330, DOI: 10.1145/1368088.1368132.

[10]. Sargsyan S., Kurmangaleev S., Baloian A.., Aslanyan H., Scalable and Accurate Clones Detection Based on Metrics for Dependence Graph, Mathematical Problems of Computer Science, Volume 42, 2014, pp. 54-62.

[11]. Avetisyan A., Kurmangaleev S., Sargsyan S., Arutunian M., Belevantsev A. LLVMBased Code Clone Detection Framework. 10th International Conference on Computer Science and Information Technologies, 2015, pp. 178-182.

[12]. Саргсян С., Курмангалеев Ш., Белеванцев А., Аветисян А. Масштабируемый и точный поиск клонов кода. Программирование, № 6, 2015, стр. 9-17.

[13]. Саргсян С., Курмангалеев Ш., Белеванцев А., Асланян А., Балоян А. Масштабируемый инструмент поиска клонов кода на основе семантического анализа программ. Труды ИСП РАН, том 27, вып. 1, 2016, стр. 39-50. DOI: 10.15514/ISPRAS-2015-27(1)-3

[14]. J. Jang and D. Brumley. Bitshred: Fast, scalable code reuse detection in binary code (cmu-cylab-10-006). CyLab, 2009 page 28.

[15]. A. Schulman. Finding binary clones with opstrings function digests: Part III. Dr. Dobb’s Journal, 30(9):64, 2005.

[16]. D. Bruschi, L. Martignoni, and M. Monga. Code normalization for self-mutating malware. IEEE Security & Privacy, 5(2):46–54, 2007.
[17]. A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In Proceedings of the 18th International Symposium on Software Testing and Analysis, ACM, 2009, pp. 117–128.
[18]. M. R. Farhadi, B. C. M. Fung, P. Charland and M. Debbabi, "BinClone: Detecting Code Clones in Malware," Software Security and Reliability (SERE), 2014 Eighth International Conference on, San Francisco, CA, 2014, pp. 78-87. doi: 10.1109/SERE.2014.21.
[19]. Thomas Dullien, Ero Carrera, Soeren-Meyer Eppler, Sebastian Porst «Automated attacker correlation for malicious code» // Technical report, DTIC Document, 2010.
[20]. https://www.hex-rays.com/products/ida/

Platform-independent and scalable tool for binary code clone detection★

[1] *H.K. Aslanyan <hayk@ispras.ru>*
[1] *S.F. Kurmangaleev <kursh@ispras.ru>*
[1] *V.G. Vardanyan <vaag@ispras.ru>*
[1] *M.S. Arutunian <aritunian@ispras.ru>*
[1] *S.S. Sargsyan <sevaksargsyan@ispras.ru>*
[1] *Institute for System Programming of the Russian Academy of Sciences,*
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. During the software development developers often copy and paste fragments of code to achieve the desired result. Copying of code can lead to variety of errors, as well as can increase the size of the source and binary code. The problem of finding semantically similar pieces of code (clones) in binary code becomes actual due to the unavailability of source code of many software programs. The first part of the article is dedicated to the analysis of the existing methods for finding code clone in binary code. In the second part we provide a newly developed tool for finding code clones in binary code. The work of the tool is divided into three main stages. The first stage is based on the Binnavi [1] framework, which is responsible for generation of program dependence graphs (PDG). Program dependence graphs are generated using REIL (Reverse Engineering Intermediate Language). The usage of REIL language allows to generate graphs for multiple architectures (x86, x86-64, ARM, MIPS, PPC), thus providing the independence of the tool from the target architecture. In the second step code clones are found based on previously created graphs. Maximum common subgraph is built for each pair of graphs and based on it, code clones are detected. In the third stage, the detected clones are visualized for convenient analysis of the results.

Keywords: code clone; semantic analysis of binary code; REIL; program dependence graph.

★ The paper is supported by RFBR grant 15-07-07541 A

DOI: 10.15514/ISPRAS-2016-28(5)-13

For citation: H.K. Aslanyan, S.F. Kurmangaleev, V.G. Vardanyan, M.S. Arutunian, S.S. Sargsyan. Platform-independent and scalable tool for binary code clone detection. *Trudy ISP RAN/Proc. ISP RAS*, vol. 1, issue 2, 2016. pp. 215-226 (in Russian). DOI: 10.15514/ISPRAS-2016-28(5)-13

References

[1]. https://www.zynamics.com/binnavi.html

[2]. Ducasse S., Rieger M., Demeyer S., A language independent approach for detecting duplicated code, in: Proceedings of the 15th International Conference on Software Maintenance, 1999, pp. 109-119, DOI: 10.1109/ICSM.1999.792593.

[3]. Kamiya T., Kusumoto S., Inoue K., CCFinder: A multilinguistic tokenbased code clone detection system for large scale source code, IEEE Transactions on Software Engineering, 2002, vol. 28, no. 7, pp. 654-670, DOI: 10.1109/TSE.2002.1019480.

[4]. Baxter I., Yahin A., Moura L., Anna M., Clone detection using abstract syntax trees, in: Proceedings of the 14th IEEE International Conference on Software Maintenance, IEEE Computer Society, 1998, pp. 368-377, DOI: 10.1109/ICSM.1998.738528.

[5]. Tairas R., Gray J., Phoenix-based clone detection using suffix trees, in: Proceedings of the 44th Annual Southeast Regional Conference, 2006, pp. 679-684, DOI: 10.1145/1185448.1185597.

[6]. Jiang L., Misherghi G., Su Z., Glondu S., DECKARD : Scalable and accurate tree-based detection of code clones, in: Proceedings of the 29th International Conference on Software Engineering, IEEE Computer Society, 2007, pp. 96-105, DOI: 10.1109/ICSE.2007.30.

[7]. Komondoor R., Horwitz S., Using slicing to identify duplication in source code, in: Proceedings of the 8th International Symposium on Static Analysis, 2001, pp. 40-56, DOI: 10.1007/3-540-47764-0_3.

[8]. Krinke J., Identifying similar code with program dependence graphs, in: Proceedings of the 8th Working Conference on Reverse Engineering, 2001, pp. 301-309, DOI: 10.1109/WCRE.2001.957835.

[9]. Gabel M., Jiang L., Su Z., Scalable detection of semantic clones, in: Proceedings of 30th International Conference on Software Engineering, 2008, pp. 321-330, DOI: 10.1145/1368088.1368132.

[10]. Sargsyan S., Kurmangaleev S., Baloian A.., Aslanyan H., Scalable and Accurate Clones Detection Based on Metrics for Dependence Graph, Mathematical Problems of Computer Science, Volume 42, 2014, pp. 54-62.

[11]. Avetisyan A., Kurmangaleev S., Sargsyan S., Arutunian M., Belevantsev A. LLVMBased Code Clone Detection Framework. 10th International Conference on Computer Science and Information Technologies, 2015, pp. 178-182.

[12]. Sargsyan S., Kurmangaleev S., Belevantsev A., Avetisyan A. Scalable and accurate detection of code clones. Programming and Computer Software, 2016, issue 1. pp. 27-33. DOI: 10.1134/S0361768816010072

[13]. Sargsyan S., Kurmangaleev S., Belevantsev A., Aslanyan H., Baloian A. [Scalable tool for code clone detection based on semantic analysis of program]. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 1, pp. 39-50 (in Russian). DOI: 10.15514/ISPRAS-2015-27(1)-3

[14]. J. Jang and D. Brumley. Bitshred: Fast, scalable code reuse detection in binary code (cmu-cylab-10-006). CyLab, 2009 page 28.

[15]. A. Schulman. Finding binary clones with opstrings function digests: Part III. Dr. Dobb's Journal, 30(9):64, 2005.

[16]. D. Bruschi, L. Martignoni, and M. Monga. Code normalization for self-mutating malware. IEEE Security & Privacy, 5(2):46–54, 2007.

[17]. A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In Proceedings of the 18th International Symposium on Software Testing and Analysis, ACM, 2009, pp. 117–128.

[18]. M. R. Farhadi, B. C. M. Fung, P. Charland and M. Debbabi, "BinClone: Detecting Code Clones in Malware," Software Security and Reliability (SERE), 2014 Eighth International Conference on, San Francisco, CA, 2014, pp. 78-87. doi: 10.1109/SERE.2014.21.

[19]. Thomas Dullien, Ero Carrera, Soeren-Meyer Eppler, Sebastian Porst «Automated attacker correlation for malicious code» // Technical report, DTIC Document, 2010.

[20]. https://www.hex-rays.com/products/ida/

Оптимизация читаемости тестов порождаемых при символьных вычислениях

И.А. Якимов <ivan.yakimov.research@yandex.ru>
А.С. Кузнецов <ASKuznetsov@sfu-kras.ru>
Институт Космических и Информационных Технологий, Сибирский Федеральный Университет,
660074, Россия, г. Красноярск, ул. Академика Киренского, д. 26

Аннотация. Занимая около половины времени разработки, тестирование остается наиболее распространенным методом контроля качества ПО, и его недостаток может приводить к финансовым потерям. При систематическом подходе тестовый набор считается полным, если он обеспечивает определенное покрытие кода. На данный момент существует большое количество систематических генераторов тестов, направленных на поиск стандартных ошибок. Подобные инструменты порождают огромное количество трудночитаемых тестов, обладающих высокой ценой проверки человеком. Представленный в данной работе метод позволяет улучшить читаемость тестов, автоматически сгенерированных при помощи символьных вычислений, обеспечивая качественное снижение данной цены. Экспериментальные исследования генератора тестов, включающего данный метод в качестве заключительной фазы работы, были проведены на 12-ти строковых функциях из репозитория Linux. Оценка степени читаемости строк, содержащихся в оптимизированных тестах, сопоставима со случаем использования слов натурального языка, что положительно сказывается на процессе верификации результатов тестирования человеком.

Ключевые слова: автоматическая генерация тестов; символьные вычисления; цена проверки тестов человеком; биграммная модель языка.

DOI: 10.15514/ISPRAS-2016-28(5)-14

Для цитирования: И.А. Якимов, А.С. Кузнецов. Оптимизация читаемости тестов порождаемых при символьных вычислениях. Труды ИСП РАН, том 28, вып. 5, 2016 г., стр. 227-238. DOI: 10.15514/ISPRAS-2016-28(5)-14

1. Введение

1.1 Обзор литературы

Занимая около половины времени разработки [1], тестирование остается наиболее распространенным методом контроля качества ПО, и его недостаток может приводить к финансовым потерям [2]. При систематическом подходе тестовый набор считается полным, если он обеспечивает определенное покрытие кода [3]. На данный момент существует большое количество систематических генераторов тестов, направленных на поиск стандартных ошибок [3, 4]. Подобные инструменты порождают огромное количество трудночитаемых тестов, обладающих высокой ценой проверки человеком [5]. В предлагаемой работе в целях качественного снижения данной цены использована концепция улучшения читаемости тестов при помощи биграммной модели естественного языка [6]. Данная концепция перенесена с генерации тестов на основе мета-эвристического поиска (SBST) на динамические символьные вычисления (DSE). На сколько известно авторам данной статьи, вышеуказанная концепция ранее не применялась к DSE и предложенный в данной работе метод является первым подобным алгоритмом оптимизации читаемости тестов при помощи модели натурального языка в контексте DSE.

Биграммная модель дает оценку вероятности P принадлежности строки корпусу языка. Согласно модели, степень читаемости строки растет по мере увеличения P. Для строки из n символов данная оценка выражается

$$\hat{P}(c_1^n) \approx \prod_{i=1}^{n} P(c_i|c_{i-1}) \quad (1)$$

следующей формулой:

где $P(c_i|c_{i-1})$ — вероятность появления символа c_{i-1} после c_{i-1} [6]. Так как оценка вероятности появления строки в корпусе языка выражается в виде произведения вероятностей появления отдельных биграмм, составляющих слово, то данная оценка для слов большей длины всегда ниже оценки для слов меньшей длины. В целях снижения влияния данного эффекта на оценку читаемости производится ее нормализация, что позволяет сравнивать строки разной длины. Оценка читаемости строки определяется [6] как нормализованное значение N оценки P:

При использовании SBST [7] цель тестирования (например, покрытие кода) формулируется в виде функции приспособленности, отображающей входные

$$N = \hat{P}(c_1^n)^{1/n} \quad (2)$$

данные на некоторый количественный показатель. Генерация тестов сводится к многократному запуску программы с подстройкой входных данных для оптимизации функции приспособленности до тех пор, пока не будет

достигнута поставленная цель тестирования. Для улучшения читаемости значение N включается в функцию приспособленности [6].

По мере выполнения символьных вычислений [8, 9] на переменные программы накладываются ограничения, образуя ограничение пути (PC), характеризующее класс эквивалентности входных данных, проводящих программу по текущему пути. В каждой точке принятия решения вычисления разветвляются, обеспечивая покрытие кода. Тесты генерируются через решение PC. В отличие от метода оптимизации читаемости на основе SBST [6] в данной работе оптимизация производится над PC.

1.2 Постановка задачи

Задачей данной работы является разработка и экспериментальное исследование метода оптимизации читаемости строк, входящих в состав тестовых случаев, порождаемых во время DSE. Оптимизация читаемости строки сводится к максимизации степени читаемости данной строки, с сохранением случайной природы порождаемых при этом слов.

2. Методы

2.1 Краткое описание

Работа генератора с оптимизацией читаемости выполняется в две фазы. На первой фазе проводятся символьные вычисления, и для каждого рассмотренного пути формируется соответствующее PC. После завершения символьных вычислений, на второй фазе над полученным PC производится оптимизация читаемости.

Возможность оптимизации читаемости исходит из того, что PC представляет собой класс эквивалентности входных данных, проводящих программу по одному пути выполнения. Для каждого отдельного пути выполнения множество решений PC может включать входные векторы со строками, содержащими печатные символы и целые слова. Предложенный в данной статье метод последовательно конкретизирует значения входящих в строки символов, по мере возможности выстраивая их согласно биграммной модели.

2.2 Пример

Для начала рассмотрим работу предложенного метода оптимизации на примере функции strlen. В данном примере (Рис. 1а) имеется одна строка — {a_1,a_2,a_3,'\0'}, где a_i — символьная переменная типа char без ограничений.

На первом проходе удается сузить диапазон значений переменных a_1-a_3 до диапазона букв (рисунки 1б и 1в). Последний символ строки '\0' изменить невозможно, так его значение фиксировано. На втором проходе удается произвести конкретизацию пар переменных (a_1, a_2) и (a_2, a_3) согласно биграммной модели, при этом получаются биграммы ('k', 'e') и ('e', 'y')

соответственно. Для пары (a3, '\0') вместо второго элемента нельзя подставить букву вследствие фиксированности '\0'. Итоговая строка имеет вид "key".

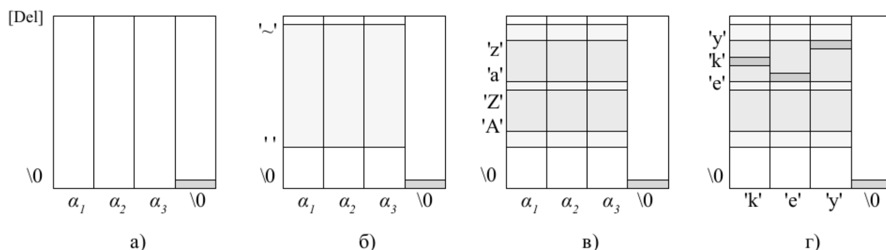


Рис. 1. Визуализация работы алгоритма оптимизации читаемости для функции ***strlen***

Fig. 1. Example of readablility optimization for the ***strlen*** *function*

2.3 Описание алгоритма

Псевдокод алгоритма оптимизации читаемости представлен на листинге 1.

Внутренним представлением тестовых данных является граф памяти M [10]. Узлами M могут быть скаляры (как символьные, так и конкретные), указатели на узлы, массивы узлов. Оптимизация читаемости осуществляется процедурами Сужение и Конкретизация, в каждой из которых происходит обход M в глубину, при этом рассматриваются только строки.

Процедура Сужение производит последовательные пробы ограничений для каждого отдельного элемента строки. Сначала производится попытка сужения диапазона значений элемента до множества печатных символов (ограничение вида ' ' $\leq a_i \leq$ '~'), при успешной попытке осуществляется проба ограничения до диапазона букв (ограничение вида 'A' $\leq a_i \leq$ 'Z' $\vee$ 'a' $\leq a_i \leq$ 'z').

В процедуре Конкретизация используется биграммная модель. На подготовительном шаге производится попытка присвоить элементу a_1 значение произвольной буквы. Далее последовательно просматриваются биграммы (a_i, a_{i+1}). Для первого элемента a_i биграммы при вызове Знач берется значение, именуемое fst, при этом учитывается, является ли данный элемент изначально конкретным или символьным. Если fst — буква, и при этом следующий за a_i элемент a_{i+1} является символьным, то происходит попытка конкретизировать значение a_{i+1}, то есть приравнять его такой букве snd, которая с большой вероятностью может следовать за fst, образуя биграмму (fst, snd). Это осуществляется при помощи вызовов След и Проба.

Процедура Сужение(граф памяти **M**)

Для каждого узла **A** графа памяти **M**

Если **A** — **char[n]**, то для каждого **i** = **1...n:**

Пусть $\mathbf{a}_i$ **= i**-й элемент массива **A**, пусть **printable** = **Проба**(' ' $\leq \mathbf{a_i} \leq$ '~')

  Если **printable**, то **Проба**(('A' ≤ $\mathbf{a_i}$ ≤ 'Z') \/ ('a' ≤ $\mathbf{a_i}$ ≤ 'z'))

Процедура Конкретизация(граф памяти **M**)
Для каждого узла **A** графа памяти **M**
  Если **A** — **char** [**n**], то
    Если a_1 — символьный, то
      Проба($\mathbf{a_1}$ = **alpha**), где **alpha** — произвольная буква
    Для каждого **i** = **1...n-1**:
      Пусть $\mathbf{a_i}$ $\mathbf{a_{i+1}}$ **—** пара соседних элементов массива, пусть **fst** = **Знач**(a_i)
        Если **fst** — буква и $\mathbf{a_{i+1}}$ — символьный, то
          Пусть **snd** = **След**(**fst**)
          Проба($\mathbf{a_{i+1}}$ = **snd**)

Процедура Знач(Узел **a** графа памяти такой, что **a** — скаляр)
  Если **a** — конкретное, то вернуть значение **a**
  Если **a** — символьное, то запросить у решателя значение **a** и если оно существует вернуть его

Процедура Проба(Ограничение **e**) : булево значение
  Протолкнуть пространство имен в стек решателя
    Пусть **PC' = PC /\ e**
    Пусть **sat** = **true**, если для **PC'** существует модель
  Вытолкнуть пространство имен из стека решателя
  Если **sat**, то **PC** ← **PC'**, вернуть **true**
  Иначе — вернуть **false**

Процедура След(ASCII символ **a**) : ASCII символ
Используя биграммную модель и алгоритм рулетки вернуть символ **b**, который может следовать за **a**

Листинг 1. Алгоритм оптимизации читаемости

Algorithm 1. Readability optimization

Для биграммы (fst, snd) процедура След(fst) : snd генерирует символ snd, который с большой вероятностью может следовать за символом fst. При этом используется метод рулетки, схожий со стратегией селекции, применяемой в генетических алгоритмах [11]. Колесо рулетки содержит 26 областей, по одной для каждого символа алфавита. Размер каждой области S_b определяется по следующей формуле:

$$S_b = P(b)/T \text{, где } T = \sum_{1}^{n} P(c_i|b) \quad (3)$$

Символ snd отбирается при помощи «запуска» рулетки, то есть генерации числа от нуля до T. Таким образом, символы с более высокой вероятностью вхождения в биграмму вида (fst | _) имеют больше шансов быть возвращенными процедурой След.

2.4 Консервативность алгоритма

Консервативность данного алгоритма заключается в том, что каждый оптимизированный тест проводит программу по тому же пути, что и соответствующий неоптимизированный вариант. К PC добавляются только такие ограничения (образуя PC'), которые не нарушают его выполнимости. Данное свойство достигается за счет использования процедуры Проба. Таким образом, решение для PC' всегда является решением для PC. Отсюда следует, что для цепочки PC, PC_1, …, PC_n, где PC — исходное (неоптимизированное) ограничение пути, PC_n — конечное (оптимизированное), решение для PC_n всегда будет решением для PC. В то же самое время, обратное неверно.

Следствием консервативности алгоритма является следующее. Если для данного PC в качестве решения не существует входного вектора, включающего строки с печатными символами, то оптимизации не происходит. Тогда тест остается без изменений, и работа завершается на проходе Сужение. Если же для данного PC существуют входные векторы, включающие подобные строки, то: во-первых, все символы, которые возможно, данный алгоритм переведет в буквы или, по крайней мере, печатные символы (проход Сужение); во-вторых, для всех строк и подстрок, где это возможно, будет применена биграммная модель для дальнейшей оптимизации читаемости (проход Конкретизация).

3. Результаты

3.1 Описание генератора тестов

Для апробации метода разработан DSE-генератор на базе инструментов LLVM [12] и CVC4 [13]. При этом использована биграммная модель естественного языка, полученная при анализе ~183 млн. слов [14].

Для начала работы пользователю достаточно написать простой драйвер на языке Си. Тесты выводятся в виде строк f: a_1 a_2 … a_n => r, где f — имя функции, a_i — значение i-го аргумента, а r — результата. Скаляры кодируются целыми числами, указатели — символом '&', строки имеют формат «$a_1a_2...a_n$»{...}, где a_i — символы слева от терминатора, {…} — дамп памяти.

3.2 Описание эксперимента

Экспериментальные исследования генератора проведены на ряде библиотечных функций из репозитория Linux [15]. Результаты эксперимента представлены в таблице 1. В первом столбце указано имя функции, во втором

— количество сгенерированных тестов, в третьем закодированы передаваемые аргументы. Строка символьных переменных представлена числом в скобках, например «[6]» обозначает строку из пяти символьных переменных и терминатора (исключение составляет strpbrk, для нее второй аргумент это строковый литерал «aeiou»). Конкретные значения (размер буфера или длина строки) приведены как целочисленные литералы без скобок. Четвертый столбец содержит процент покрытия инструкций, полученный при помощи gcov. В последующих столбцах указаны результаты усредненной оценки читаемости для оптимизированных тестов.

Табл. 1. Результаты экспериментальных исследований

Table 1. Experimantal results

Функ	#	***Арг***	***Покр***	***Нет***	***Баз***	***Прос***	***СП***	***Рул***	***Случ.***
M			**95%**	**-**	**0,07**	**0,08**	**0,08**	**0,07**	**0,06**
σ			**8%**	**-**	**0,03**	**0,03**	**0,03**	**0,03**	**0,02**
strlen	5	[6][6]	100%	-	0,08	0,10	0,10	0,09	0,08
strnlen	6	[6][6]5	100%	-	0,08	0,10	0,10	0,09	0,08
strcmp	15	[6][6]	100%	-	0,04	0,05	0,05	0,04	0,04
strncmp	16	[6][6] 5	100%	-	0,04	0,05	0,05	0,04	0,04
sysfs_ streq	39	[6][6]	100%	-	0,05	0,07	0,06	0,06	0,05
strcpy	35	[6][6]	100%	-	0,08	0,10	0,10	0,09	0,07
strncpy	36	[6][6] 5	100%	-	0,08	0,10	0,10	0,09	0,07
strcat	40	[10][5]	100%	-	0,07	0,08	0,08	0,07	0,06
strncat	41	[10][5] 4	100%	-	0,07	0,08	0,08	0,07	0,06
strstr	19	[6][3]	90%	-	0,12	0,14	0,14	0,13	0,11
strnstr	4	[6][3]2	80%	-	0,09	0,10	0,10	0,09	0,07
strpbrk	10	[6][6]	80%	-	0,03	0,03	0,03	0,03	0,03

Столбец «Нет» содержит данные по запуску без оптимизации. Неоптимизированный вывод не содержит буквы и оценка читаемости для него не определена. В качестве базового метода «Баз.» взят проход Сужение без последующей конкретизации. Условный «простейший» метод применения

биграммной модели отображен в столбце «Прост.», здесь для каждой строки первый символ берется без дополнительных ограничений, все последующие выстраиваются по биграммной модели. Модифицированной для эксперимента процедурой След возвращается символ с *наибольшим* значением вероятности вхождения в биграмму. В эксперименте «СП» включена предусмотренная методом попытка случайной генерации первого символа строки. Столбец «Рул.» отображает результаты итоговой версии метода – здесь в процедуре След использован метод рулетки. Завершающий столбец «Случ.» содержит данные эксперимента, в котором процедура След возвращает случайную букву, т.е. биграммная модель не задействована. Так как в использованной биграммной модели значения приведены с точность 2 знака после запятой, тот же формат принят и в таблице.

Для экспериментов, результаты которых представлены в столбцах «СП», «Рул» и «Случ», где используется генерация случайных чисел, было проведено 5 запусков. Среднеквадратическое отклонение результатов данных запусков от среднего близко к нулю (при рассмотрении двух цифр после запятой это 0), поэтому данный показатель не был включен в таблицу, а указано лишь среднее значение по 5 запускам. Для столбцов, отображающих покрытие кода, а также столбцов с оценками читаемости по разным экспериментам, приведено усредненное значение M результатов по отдельным функциям и среднеквадратическое отклонение σ данных результатов от M.

3.3 Обсуждение

Наибольшая оценка читаемости получена для условно «простейшего» метода. Однако с его применением связаны некоторые неудобства со стороны пользователя. Для наглядности рассмотрим тесты для функции strcpy. Примером неоптимизированного вывода является «&"\x01\x01\x01\x01\x01"{\0} &"\x01\x01\x01"{\0,\0,\0} :=> &"\x01\x01\x01"{\0,\x01,\0}». По данному тесту при ручной проверке без дополнительной информации об исходном коде, затруднительно установить, произошло ли копирование в действительности. Так как strcpy производит копирование данных из одного буфера в другой, не накладывая дополнительных ограничений на символы в строке, то CVC4 в качестве решения для каждой символьной переменной выдает одни и те же значения. Далее, рассмотрим вывод базового метода: «&"aaaaa"{\0} &"aaa"{\0,p,\0} :=> &"aaa"{\0,a,\0}». Читаемость вывода улучшилась, однако проверить корректность также затруднительно. Также стоит отметить, что человеку легче воспринимать слова, нежели последовательности повторяющихся букв. В случае использования «простейшего» метода получаются тесты вида «&"athes"{\0} &"ath"{\0,p,\0} :=> &"ath"{\0,s,\0}». Читаемость тестов повышается, однако проблема проверки остается. Данная проблема решается случайной генерацией первого символа строки. В эксперименте «СП»

примером вывода является «&"kesth"{\0} &"pre"{\0,p,\0} :=> &"pre"{\0,h,\0}». Видно, что содержимое буфера назначения действительно изменилось, как и ожидалось. Однако данный метод имеет тенденцию к зацикливанию, т. к. результат процедуры След в данном случае строго детерминирован. Например, может получиться строка «thesthes...», при этом без дополнительных ограничений она зациклится. Для того чтобы избежать данной проблемы используется метод рулетки. Так как метод рулетки не является детерминированным, и в то же время опирается на вероятность следования символов, заложенную в биграммную модель, он улучшает читаемость по сравнению со случайной генерацией символов, одновременно ликвидируя зацикливание. Примером вывода по данному методу является: «&"fmf"{\0,\x01,\x01} &"emslo"{\0} 5 :=> &"emslo"{\0}». Наконец, для сравнения использована стратегия «Случ» случайного выбора символа, пример «&"jbg"{\0,\x01,\x01} &"ttabn"{\0} :=> &"ttabn"{\0}».

3.4 Достоверность

Для оценки читаемости тестов использованы усредненные значения всех тестов одной функции. Данная стратегия была выбрана вследствие того, что читаемость входных строк зависит от вида функции, и в то же самое время в тестах для одной функции сочетается множество комбинаций строк различной длины (т. к. длина отличается от размера буфера и задается символом '\0'). Различие в длине строк частично компенсируется нормализацией, однако это ведет к тому, что строки меньшей длины получают завышенные оценки. На наш взгляд, в первом приближении этого достаточно для оценки результатов, однако для их уточнения необходимы дальнейшие исследования.

Также стоит отметить, что целью работы не является исследование биграммной модели в изоляции, в отрыве от задачи автоматизации тестирования. Однако для валидации метода оценки читаемости были взяты 100 наиболее частотных слов английского языка [16], а также сто случайных строк длины 2-5 символов. В первом случае полученная оценка составляет 0,10, во втором — 0,07. Эксперименты показывают, что результаты работы оптимизатора сопоставимы со значениями, полученными изолированно.

Традиционно критерием полноты тестирования в символьных вычислениях является покрытие инструкций. В проведенных экспериментах оно в среднем составило 95% и для 9 из 12 функций — 100%. В функциях с неполным покрытием не был рассмотрен вариант, при котором возвращается NULL.

В заключение добавим, что так как символьные вычисления генерируют тесты, систематически исследуя развилки в коде программы, то количество тестов зависит от количества пройденных развилок. В случае функций strlen и strnlen покрытие кода для заданных аргументов максимально возможно — рассмотрены все возможные пути выполнения. В остальных случаях сказалась неполнота теории использованной для поиска решений РС.

4. Заключение

Представленный метод позволяет улучшить читаемость тестов, автоматически сгенерированных при помощи символьных вычислений. Экспериментальные исследования системы, реализующей данный метод, были проведены на 12-ти функциях по работе со строками из репозитория Linux. Проведенные исследования демонстрируют преимущества данного метода при генерации тестов для функций, включающих строковые данные, в том случае, когда они должны проверяться человеком. Оценка степени читаемости строк, содержащихся в оптимизированных тестах, сопоставима со случаем использования слов натурального языка, что, как показывают исследования в смежной с DST области SBST [6], положительно сказывается на процессе верификации тестов человеком. В то же время неоптимизированные тесты включают трудночитаемые последовательности произвольных символов внутри широкого диапазона ASCII-кодировки. Данный метод может быть встроен в другие системы DSE-генераторы. Настройки метода, приведенные в экспериментах, могут быть перенесены в подобную систему в качестве параметров командной строки. В дальнейшем планируется работа по расширению возможностей генератора, а также экспериментальные исследования его эффективности для более широкого класса программ.

Список литературы

[1]. Hambling B. Realistic and Cost-effective Software Testing. Kelly, Management and Measurement of Software Quality, UNICOM SEMINARS, Middlesex, UK, 1993, pp. 95-112.

[2]. Tassey G. The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology, Final Report, May 2002.

[3]. Saswat A., Burke E.K., Chen T.Y., Clark J., Cohen M.B., Griskamp W., Harman M., Harrold M.J., McMinn P. An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software, 2013, vol. 86, issue 8, pp. 1978-2001.

[4]. Cadar C., Godefroid P., Khundish S., Păsăreanu C.S., Sen Koushik, Tillman N., Visser W. Symbolic Execution for Software Testing in Practice – Preliminary Assessment. ICSE'11 Proceedings of the 33rd International Conference on Software Engineering, 2011, pp. 1066-1071.

[5]. Barr E.T., Harman M., Shahbaz M., Yoo S. The Oracle Problem in Software Testing: A Survey. IEEE Transactions on Software Engineering, 2015, vol. 41, issue 5, pp. 507-525.

[6]. Afshan S., McMinn P., Stevenson M. Evolving readable string test inputs using a natural language model to reduce human oracle cost. International Conference on Software Testing, Verification and Validation, 2013.

[7]. Tracey N., Clark J., Mander K., McDermid J. An automated framework for structural test-data generation. Proceedings of the International Conference on Automated Software Engineering, 1998, pp. 285–288.

[8]. King J.C. Symbolic execution and program testing. Communication of The ACM, 1976, vol. 19, issue 17, pp. 385-394.

[9]. Cadar C., Ganesh V., Pawlowski P.M., Dill D.L., Engler D.R. EXE: automatically generating inputs of death. Proceedings of the 13th ACM conference on Computer and Communication security, 2006, pp 322-335.
[10]. Sen K., Marinov D., Agha G.. CUTE: a concolic unit testing engine for C. ESEC/FSE-13 Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundationsof software engineering, 2006, vol. 30, issue 5, pp. 263-272
[11]. J. H. Holland. Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor, 1975.
[12]. Компиляторная инфраструктура LLVM. Режим доступа: http://llvm.org/
[13]. SMT-решатель CVC4. Режим доступа: http://cvc4.cs.nyu.edu/web/
[14]. Jones M. N., Mewhort D.J.K. Case-sensitive letter and bigram frequency counts from large-scale English corpora, 2004, vol. 36, issue 3, pp. 388
[15]. Исходный код ядра Linux. Режим доступа: https://github.com/torvalds/linux/tree/master/kernel
[16]. 100 наиболее частотных английских слов. Режим доступа: https://en.oxforddictionaries.com/explore/what-can-corpus-tell-us-about-language

Test Readability Optimization in Context of Symbolic Execution

I.A. Yakimov <ivan.yakimov.research@yandex.ru>
A.S. Kuznetsov <ASKuznetsov@sfu-kras.ru>
Institute of space and informatic technologies, Siberian Federal University, Akademika Kirenskogo 26 st., Krasnoyarsk, 660074, Russia.

Abstract. Software testing is a time consuming process. In general, software companies spend about 50% of development time on testing. On the other hand, lack of testing implies financial and other risks. In the case of systematic testing a suitable test suite should provide an appropriate code coverage. A lot of code-based test generators have been developed in order to provide systematic code coverage. Such tools tend to produce lots of almost unreadable test suites which hard to verify manually. This problem is formulated as a Human Oracle Cost Problem. This work introduces a method for readability optimization of automatically generated test suites in context of Symbolic Execution. It uses natural language model in order to optimze each test case. This conception has been applied first in the Search Based Testing paradyghm. In contrast of existing search based tool proposed DSE-based tool uses SMT-solver in order to incrementally improve readability of a single test for a single program path. To validate proposed method a tool on top of LLVM and CVC4 frameworks is created. Experimental evaluation on 12 string processing routins from the Linux repository shows that this method can improve test inputs gracefully.

Keywords: code-based test generation; symbolic execution; human oracle cost; bigram language model;

DOI: 10.15514/ISPRAS-2016-28(5)-14

For citation: I.A. Yakimov, A.S. Kuznetsov. Test Readability Optimization in Context of Symbolic Execution. *Trudy ISP RAN/Proc. ISP RAS*, vol. 1, issue 2, 2016. pp. 227-238 (in Russian). DOI: 10.15514/ISPRAS-2016-28(5)-14

References

[1]. Hambling B. Realistic and Cost-effective Software Testing. Kelly, Management and Measurement of Software Quality, UNICOM SEMINARS, Middlesex, UK, 1993, pp. 95-112.

[2]. Tassey G. The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology, Final Report, May 2002.

[3]. Saswat A., Burke E.K., Chen T.Y., Clark J., Cohen M.B., Griskamp W., Harman M., Harrold M.J., McMinn P. An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software, 2013, vol. 86, issue 8, pp. 1978-2001.

[4]. Cadar C., Godefroid P., Khundish S., Păsăreanu C.S., Sen Koushik, Tillman N., Visser W. Symbolic Execution for Software Testing in Practice – Preliminary Assessment. ICSE'11 Proceedings of the 33rd International Conference on Software Engineering, 2011, pp. 1066-1071.

[5]. Barr E.T., Harman M., Shahbaz M., Yoo S. The Oracle Problem in Software Testing: A Survey. IEEE Transactions on Software Engineering, 2015, vol. 41, issue 5, pp. 507-525.

[6]. Afshan S., McMinn P., Stevenson M. Evolving readable string test inputs using a natural language model to reduce human oracle cost. International Conference on Software Testing, Verification and Validation, 2013.

[7]. Tracey N., Clark J., Mander K., McDermid J. An automated framework for structural test-data generation. Proceedings of the International Conference on Automated Software Engineering, 1998, pp. 285–288.

[8]. King J.C. Symbolic execution and program testing. Communication of The ACM, 1976, vol. 19, issue 17, pp. 385-394.

[9]. Cadar C., Ganesh V., Pawlowski P.M., Dill D.L., Engler D.R. EXE: automatically generating inputs of death. Proceedings of the 13th ACM conference on Computer and Communication security, 2006, pp 322-335.

[10]. Sen K., Marinov D., Agha G.. CUTE: a concolic unit testing engine for C. ESEC/FSE-13 Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundationsof software engineering, 2006, vol. 30, issue 5, pp. 263-272

[11]. J. H. Holland. Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor, 1975.

[12]. LLVM: http://llvm.org/

[13]. CVC4: http://cvc4.cs.nyu.edu/web/

[14]. Jones M. N., Mewhort D.J.K. Case-sensitive letter and bigram frequency counts from large-scale English corpora, 2004, vol. 36, issue 3, pp. 388

[15]. Linux: https://github.com/torvalds/linux/tree/master/kernel

[16]. The 100 commonest English words https://en.oxforddictionaries.com/explore/what-can-corpus-tell-us-about-language

Декларативный язык FlexT – инструмент анализа и документирования бинарных форматов данных

А.Е. Хмельнов <hmelnov@icc.ru>
И.В. Бычков <bychkov@icc.ru>
А.А. Михайлов <mikhailov@icc.ru>
Институт динамики систем и теории управления имени Матросова СО РАН, 664033, Россия, г. Иркутск, ул. Лермонтова, д. 134

Аннотация. Язык FlexT разработан для спецификации бинарных форматов данных. Язык является декларативным, рассчитанным на хорошее восприятие человеком, его основными конструкциями являются определения типов данных, которые напоминают определения типов в императивных языках программирования, но являются более гибкими. В работе сделан обзор возможностей современных проектов, направленных на спецификацию бинарных форматов файлов. Далее рассматриваются особенности языка FlexT, отдельно описываются возможности языка, позволяющие работать с форматами кодирования машинных команд. Кратко описаны реализованные программные системы, использующие интерпретатор FlexT и некоторые новые возможности поиска информации в бинарных файлах, которые даёт использование спецификаций.

Ключевые слова: спецификация бинарных форматов данных, спецификация кодирования машинных команд, декларативный язык, дизассемблер

DOI: 10.15514/ISPRAS-2016-28(5)-15

Для цитирования: А.Е. Хмельнов, И.В. Бычков, А.А. Михайлов. Декларативный язык FlexT – инструмент анализа и документирования бинарных форматов данных. Труды ИСП РАН, том 28, вып. 5, 2016 г., стр. 239-268. DOI: 10.15514/ISPRAS-2016-28(5)-15

1. Введение

При выборе существующих или разработке собственных форматов файлов для сохранения информации на диске, а также при необходимости организовать файловый обмен данными между различными приложениями, разработчики программного обеспечения встают перед выбором между бинарными и текстовыми форматами представления информации. Бинарные форматы в большинстве случаев являются более компактными, программный код для

работы с бинарными данными также может быть значительно более простым по сравнению с кодом для обработки текста (например, одной командой можно прочитать из бинарного файла запись, содержащую сразу несколько десятков полей). Однако, текстовые файлы являются более прозрачными: для их просмотра можно воспользоваться любым текстовым редактором. Поэтому, в настоящее время разработчики часто жертвуют эффективностью ради упрощения контроля правильности структуры файла, и выбирают текстовые форматы. Часто такие форматы основываются на синтаксисе XML, поскольку для этого синтаксиса существуют готовые библиотеки и инструментальные средства, облегчающие разработку алгоритмов чтения/записи информации, и автоматизирующие контроль правильности структуры файла.

Для поддержки использования нового формата данных в разрабатываемой программе можно подключить библиотеку для работы с этим форматом или написать собственный код с использованием спецификации формата. Любая библиотека навязывает разработчику свои структуры данных, которые, если программа пишется не с нуля, скорее всего, будут отличаться от уже используемых. В результате потребуется написать дополнительный программный код для конвертации данных, что скажется на скорости работы программы. Кроме того, подходящие библиотеки могут и отсутствовать. При отсутствии готовых библиотек, а также при желании получить более эффективную реализацию, придётся написать код для работы с форматом данных по его спецификации. Используемая спецификация может быть написана на естественном языке, кроме того, для изучения формата данных могут служить исходные тексты программ работающих с этим форматом и даже примеры относящихся к формату файлов. Рассмотрим особенности каждого из этих источников информации о формате более подробно:

- Спецификации на естественном языке практически всегда содержат неточности и неоднозначности толкования. То, что было очевидно автору спецификации, может быть непонятно читателю, или, может быть понято им превратно. Содержащаяся в такой спецификации информация не была верифицирована путём её использования для реальной работы с данными, поэтому очень часто она является ненадёжной.
- С другой стороны, исходные тексты программ для работы с рассматриваемыми файлами, прошли проверку на реальных данных и, скорее всего, не содержат подобных ошибок. Однако, в этих исходных текстах информация о структурах данных файла разбросана по коду, предназначенному для конкретного способа работы с этими данными, поэтому изучение такого кода в качестве документации по формату является очень непростой задачей.
- Ещё более сложной является задача восстановления недокументированного формата данных по примерам файлов этого

формата. Она требует выполнения многочисленных попыток разбора файлов при некоторых предположениях об их структуре с выводом всей извлекаемой информации для анализа. Программа, выполняющая такую работу, будет существенно сложнее программы, просто считывающей информацию во внутреннее представление.

Кроме того, при разработке программы, записывающей данные в бинарный формат, часто очень остро встаёт проблема её отладки: сложно понять, что требуется исправить, когда используемое для проверки правильности полученного файла приложение выдаёт неинформативное сообщение, рассчитанное на конечного пользователя, например, просто пишет: «Ошибка в формате файла» (без указания места и других подробностей).

Таким образом, является целесообразной разработка специальных инструментов для задачи анализа содержимого бинарных файлов с использованием спецификаций форматов данных. В статье рассматривается разработанный авторами язык FlexT, предназначенный для описания бинарных форматов данных, и реализованные с использованием интерпретатора этого языка инструменты, предназначенные для анализа бинарных файлов. Преимуществами использования формальных спецификаций являются:

- компактность, отсутствие не относящихся к способам хранения данных сведений, что облегчает их восприятие человеком;
- верифицируемость посредством их применения для анализа относящихся к формату корректных данных;
- возможность использования спецификаций для локализации ошибок в сгенерированных файлах.

Далее в главе 2 будет сделан обзор существующих средств спецификации бинарных форматов данных, в главе 3 будут рассмотрены особенности и основные конструкции языка FlexT. В главе 4 будут описаны типы данных языка FlexT, ориентированные на спецификацию кодирования машинных команд. В главе 5 будут рассмотрены разработанные программы, использующие спецификации для исследования бинарных файлов и ряд возможностей инструментов, использующих интерпретатор FlexT.

2. Обзор средств спецификации бинарных форматов данных

К моменту создания языка FlexT (конец 1990-х) задачей спецификации бинарных форматов данных почти никто не занимался. В качестве обзора литературы тогда удалось найти только одну хоть немного действующую ссылку на проект BFF (Binary File Format Definition), сейчас страница проекта существует и находится по адресу [1]. Первоочередной задачей разработки языка BFF являлось обратное проектирование формата AutoCAD DWG. Автор использовал термин «грамматика», т.е. отталкивался от концепции

формальных грамматик при проектировании языка спецификаций. Другие бинарные форматы, помимо DWG, описаны не были, уровень описания формата DWG также не удаётся оценить.

За прошедшее время уровень понимания необходимости использования спецификаций для работы с бинарными данными существенно вырос. Рассмотрим появившиеся проекты, в основном в порядке их упоминания в выдаче поисковой системы.

Стандарт DFDL (Data Format Description Language) [2][3] разрабатывается для описания текстовых и бинарных данных, используемых в GRID-системах, судя по приводимым примерам, в основном табличных (на уровне возможностей формата CSV). Стандарт DFDL был опубликован, как рекомендованные предложения форума Open Grid в 2011 г. Целью разработки стандарта является унификация обработки (чтения и записи) из текстовых (XML, CSV) и бинарных данных содержащегося в них «набора информации» (information set). Описания форматов кодируются на XML. Имеется коммерческая реализация парсера DFDL в составе продукта IBM Integration Bus [4], а также открытая реализация [5]. Сами описания бинарных данных являются достаточно примитивными (на уровне записей с полями постоянного размера в терминах FlexT), в силу ограничений стандарта, а также потому, что существующий парсер поддерживает не все предусмотренные этим стандартом возможности [6]. Тем не менее, эта реализация позволяет, например, использовать для обмена данными простой бинарный формат, сопроводив его спецификацией на DFDL. Никакие описания настоящих форматов данных, таких как DBF, на DFDL найти не удалось. Популярность DFDL объясняется большим объёмом подготовленной документации по формату и участием в его разработке гиганта IBM.

Аналогичную задачу решает язык MFL (Message Format Language) в составе продукта WebLogic Integration фирмы Oracle [7]. В данном случае обмен информацией с использованием специальных бинарных форматов выполняется между приложениями, использующими эту платформу. Язык также основан на XML и позволяет описывать форматы с последовательной формой хранения информации (без указателей в терминах FlexT). Допускается использование повторяющихся элементов, а также необязательных полей. Для построения спецификации формата используется специальный интерактивный редактор Format Builder. Так же, как и DFDL, MFL не предназначен для спецификации произвольных бинарных форматов, а лишь тех, которые укладываются в рамки его возможностей, с учётом которых и должен проектироваться формат.

Существует ряд проектов, направленных на описание сетевых протоколов, например, NetPDL [8] и BinPAC [9]. Язык NetPDL предназначен для документирования форматов пакетов сетевых протоколов и основан на синтаксисе XML. Язык BinPAC направлен на генерацию кода для чтения пакетов с использованием генератора синтаксических анализаторов Bison.

Поскольку сама природа сетевых протоколов предполагает последовательное чтение данных, такие языки также позволяют описать лишь форматы с последовательной формой хранения информации (без указателей в терминах FlexT). Поддерживаются все основные конструкции, предназначенные для описания потоков данных (массивы, записи, варианты в терминах Flext), параметризация типов. В языке NetPDL имеются простые битовые типы, в языке BinPAC битовых типов нет.

В ходе выполнения ранее упоминавшегося обзора в конце 1990-х годов остался незамеченным язык EAST (Enhanced Ada SubseT) [10], часть информации о котором была опубликована ещё в 1997. Язык разработан консультативным комитетом систем космических данных (CCSDS), в котором участвует и Российское космическое агентство, для облегчения обмена информацией, передаваемой космическими системами. В основу языка положена подсистема определения типов данных языка Ada. В языке EAST поддерживаются поля записей переменного размера, массивы с задаваемой в некотором поле длиной, а также со стоп-маркером; вариантные части, содержимое которых определяется некоторым внешним полем (выбор по внутреннему содержимому не предусмотрен). Более сложные, чем значение некоторого поля, выражения для параметров типов не поддерживаются. Большое внимание уделяется битовым типам данным и учёту порядка байтов. Отсутствуют указатели, как нехарактерные для данных, передаваемых последовательно. Используемый способ передачи параметров типов данных не слишком гибок и не очень удобен.

Язык HUDDL [11] разработан для описания форматов представления гидрографических данных. Язык также основан на синтаксисе XML и ориентирован на описание потоков бинарных данных (уровень записей и массивов в терминах FlexT). Спецификации используются для генерации кода для работы с данными на различных языках программирования. Авторам известен проект DFDL, но они предпочитают разработать свой язык, ориентированный на особенности гидрографических данных.

В проекте «Advanced Language Processing Technology Applied to Digital Records» [12] сотрудники военного НИИ из США рассматривают задачу интеграции разнородной цифровой информации, используемой для принятия военных решений. Предлагается использовать аппарат атрибутивных грамматик для описания бинарных форматов данных [13]. Далее используется генератор синтаксических анализаторов (парсеров) ANTLR [14] для автоматического построения по спецификации-грамматике кода чтения бинарного файла. Все приведённые примеры имеют дело с форматами с последовательной записью данных (в терминах FlexT: без использования указателей). Такой подход позволяет частично ускорить разработку модулей чтения информации для ряда форматов.

В проекте DataScript [15] спецификации форматов данных используются для генерации библиотек для чтения данных на языке Java. Вместе с исходными

текстами [16] опубликованы два описания форматов: файла класса Java (без описания машинных команд) и DVI (оба формата являются последовательными, без указателей в терминах FlexT). Кроме того, в статье [15] приводятся фрагменты спецификации формата ELF, для описания которого требуются указатели. Вместо указателей в DataScript используются метки, содержащие выражения с адресом фрагмента файла. С 2003 г. исходные тексты проекта не обновлялись. Спецификация рассматривается, как набор определений типов данных, этот подход наиболее близок используемому в языке FlexT. При этом для работы с битовыми типами определены отдельные (отличные от байтовых) типы данных (битовые поля), в полном объёме битовые типы не поддерживаются. Для описания вариантных блоков используется конструкция union, в которой происходит выбор того варианта, заданные для которого ограничения выполняются, сами ограничения могут задаваться в виде ожидаемых значений отдельных полей. Этот подход не позволяет описать выбор вариантов по задаваемому извне значению, например, когда тип блока задаётся в той записи, где находится указатель на блок.

Целью проекта Miraplacid Binary DOM (Binary Document Object Model) [17] является реализация универсальной библиотеки для доступа к бинарным и текстовым данным в стиле DOM, используемой для HTML-документов. В спецификациях бинарных данных (файлы с расширением `bdd`) используются массивы записи и указатели (в терминах FlexT), но не удалось найти никакой поддержки вариантных блоков. Также заметно отсутствие битовых типов данных. Спецификация представляет собой набор типов данных. На сайте [18] описан 21 формат, некоторые из них – текстовые (INI, RTF, XML). Реализована программа для просмотра содержимого бинарного файла по спецификации (отображается дерево обнаруженных структур данных и шестнадцатеричный дамп, в котором выделяется блок памяти, соответствующий выбранному узлу дерева)

Проект Kaitai Struct [19] ориентирован на создание парсеров бинарных данных по спецификациям. Спецификация в файле с расширением `ksi` по своей структуре напоминает формат JSON, в котором отступы используются вместо скобок. Основной структурой данных является последовательность (запись в терминах FlexT), которая может иметь повторяющиеся элементы (массивы в терминах FlexT). Поддерживаются указатели, а вместо вариантных типов данных могут использоваться условия вхождения полей в записи, что может иногда оказаться не слишком удобным, в особенности при задании условия для поля, соответствующего ветви варианта ELSE (там необходимо записать конкатенацию отрицаний условий всех остальных ветвей). Также не поддерживаются битовые типы данных.

Программа Synalyze It! [20] является средством просмотра/шестнадцатеричным редактором бинарных файлов. Эта программа предназначена для работы под macOS (распространяется платно), существует

её бесплатная версия для других платформ под названием Hexinator [21]. Программа может использовать спецификации бинарных форматов для формирования дерева найденных в файле структур данных и выделения цветом соответствующих им фрагментов файла. Спецификации форматов данных называются грамматиками и хранятся в основанном на XML представлении. Для разработки спецификаций реализован интерактивный редактор. В списке имеющихся описаний [22] в основном представлены специфичные для macOS форматы, но есть и более распространённые, такие как DBF, Shape или PE EXE. Используемый язык спецификаций нельзя считать полностью декларативным, поскольку описания форматов содержат достаточно большие вставки на языке Python.

Таким образом, к настоящему времени во многих предметных областях осознана необходимость использования спецификаций форматов данных. При этом иногда ставится задача не описания произвольных форматов, а лишь облегчения работы с некоторым их подмножеством. В некоторых случаях задача описания произвольных форматов ставится, но реализуются не все конструкции, необходимые для её решения. Часто для спецификаций выбирается представление XML, что затрудняет восприятие текста спецификации человеком и требует использования специальных средств просмотра и интерактивных редакторов, которые, хотя и облегчают выполнение ряда операций редактирования, но затрудняют выполнение некоторых других действий, например, не позволяют одновременно увидеть и изменить характеристики сразу нескольких узлов. В некоторых проектах о спецификации бинарного формата думают, как о грамматике, а в других спецификация рассматривается, как набор типов данных. Многие языки спецификаций были созданы для облегчения написания кода для работы с данными, поэтому эту задачу можно считать востребованной. Также очень заметно, что в статьях по тематике очень редко встречаются ссылки на аналоги, упоминались как аналоги лишь нескольких проектов: EAST, DFDL и DataScript.

3. Особенности языка FlexT

Бóльшая часть информации о формате данных на языке FlexT задаётся при помощи набора определений типов данных. При этом, по сравнению с типами данных в императивных языках программирования, типы данных FlexT могут содержать составляющие, размер которых определяется конкретными данными, представленными в этом формате. Т.е. можно сказать, что типы гибко подстраиваются под данные, этим объясняется название языка **FlexT** (от англ. **Flex**ible **T**ypes). После определения типов данных необходимо задать размещение в памяти некоторых элементов данных, относящихся к каким-то из этих типов. Такие элементы данных по аналогии с императивными языками программирования будем называть переменными (хотя они и описывают неизменяемые данные).

Синтаксис языка выбран так, чтобы он хорошо воспринимался человеком. При проектировании некоторых из рассмотренных в обзоре языков, например, языка EAST, такая задача явно формулировалась в требованиях к разработке. Рассмотрим основные особенности и принципы создания языка FlexT, которые делают его применимым для описания широкого круга форматов.

3.1 Уровни языков спецификации форматов данных

По возможностям работы с описываемыми данными можно различать два уровня языков спецификации данных: *спецификация интерпретации* и *спецификация редактирования*. Язык спецификации интерпретации должен позволять *легко* описывать интерпретацию произвольного (в рамках определённых ограничений) *класса данных*. Под интерпретацией здесь понимается не преобразование всех данных в некоторый конкретный формат, а наличие в спецификации информации о способах извлечения из описанных данных значений свойств рассматриваемого класса данных. Например, спецификация формата файла растровой графики должна позволять извлечь из такого файла информацию о размерах изображения (ширина, высота) и цвете каждого пикселя, расположенного в границах изображения.

Язык спецификации редактирования должен кроме определения функций-наблюдателей для считывания информации давать определения конструкторов, позволяющих создавать по заданным свойствам новые экземпляры данных. При этом приходится учитывать дополнительные детали, например, распределение памяти, порядок порождения элементов данных, соглашения по их выравниванию, способ заполнения пропусков, алгоритм генерации идентификаторов новых объектов, и т.д. В текущей версии язык FlexT не поддерживает описание таких спецификаций.

Слова "легко", "простота" и т.п. часто употребляется при определении понятия "спецификация". Подразумевается, что запись некоторой информации на языке спецификации должна быть проще, чем на обычном языке программирования. Более существенным здесь является не субъективное понятие простоты, а отсутствие в спецификации избыточной информации. Язык спецификации должен быть как можно более декларативным: он должен описывать то, как размещаются данные, а не то, как их надо читать.

Одни и те же данные могут интерпретироваться разными способами. Например, файл одноканального растрового формата может отображаться, как изображение в тонах серого, или как матрица высот, или как двумерная таблица. В качестве базовой интерпретации, пригодной для описания любого формата данных, можно предложить такую, которая приписывает тип каждому элементу данных — выполняет *идентификацию типов данных*. Такая интерпретация является базовой и минимальной, поскольку при использовании некоторого элемента данных в интерпретации более высокого уровня всё равно придётся определить, к какому типу этот элемент относится. В качестве класса данных, в который они отображаются при идентификации

типов, используется набор взаимосвязанных *элементов данных*. Каждый элемент данных характеризуется своим размещением (*адресом* и *размером*) и *типом*, и может содержать ссылки на другие элементы данных, а также необходимую для полного описания других элементов данных информацию. Размер элемента данных определяется его типом. Элементы данных могут быть составными — в этом случае внутри них можно выделить элементы меньшего размера. Элемент данных можно сравнить с термом, а идентификацию типов данных — с эрбрановой интерпретацией в логике первого порядка. Интерпретация типов данных может быть *неполной*. В этом случае она содержит элементы данных, которым не сопоставлен тип или, точнее, в этом случае можно считать, что им сопоставлен примитивный тип — *сырые данные*.

Рассматриваемый в данной работе язык спецификации форматов данных FlexT позволяет описывать интерпретацию статических данных, в первую очередь — в виде идентификации типов. Данный язык применяется для описания форматов данных в программах просмотра/дизассемблере бинарных файлов, а также в программе ExeXpl, предназначенной для исследования кода и структур данных в исполняемых файлах Windows.

3.2 Динамические и статические типы данных

Под статическими типами данных здесь мы будем понимать аналоги большинства традиционных типов данных, реализованных в процедурных языках программирования. Их отличительной особенностью является то, что размер элемента данных такого типа и внутреннее размещение составляющих его элементов определены в момент компиляции и не зависят от конкретных данных. В процедурных языках программирования, по крайней мере, в тех, которые реально используются в настоящее время, составные типы данных могут содержать только статические составляющие.

Размер элемента данных динамического типа и внутреннее размещение его составляющих могут зависеть от конкретных данных. Далее слово "динамический" будет использоваться именно в этом смысле. Примером динамических типов в традиционных процедурных языках являются строковые константы. Например, для ASCIIZ строк, чтобы определить занимаемый ими размер, нужно просмотреть всю строку в поисках нулевого символа.

Динамические типы данных непригодны в качестве типов переменных, по крайней мере, если этот тип изменяемый (в терминах [23]), поскольку присваивание новых значений элементам таких данных может означать изменение размера, а такие операции ни один компилятор не сможет эффективно поддержать. Для полей типов данных, содержащих те же строки или записи с вариантами, память сразу выделяется по максимуму. Примером поддержки компилятором динамического перераспределения памяти, занимаемой значением переменной, являются huge-строки Delphi, память под

которые автоматически запрашивается в куче, при этом сами переменные такого типа фактически являются указателями и всегда занимают 4 байта.

В то же время, если рассматривать статические данные, которые программа может только читать, то здесь иногда используются весьма изощрённые способы их кодировки. В качестве примера статических данных в коде программы можно привести RTTI Delphi — способ представления метаинформации о типах данных:

```
PTypeInfo = ^TTypeInfo;
TTypeInfo = record
  Kind: TTypeKind;
  Name: ShortString;
 {TypeData: TTypeData}
end;

function AfterString(Str: PShortString): Pointer; inline;
begin
  Result := PByte(Str) + Length(Str^) + 1;
end;

function GetTypeData(TypeInfo: PTypeInfo): PTypeData;
begin
  Result := AfterString(@TypeInfo^.Name);
end;
```

Листинг 1. Фрагменты файла TypeInfo.pas

Listing 1. Excerpts from the file TypeInfo.pas

Информация о типе данных закодирована в записи типа `TTypeInfo`, в которой за кодом вида типа данных и его именем следует запись `TypeData` с остальной информацией о типе. Определение поля `TypeData` заключено в комментарий, т.к. на самом деле эти данные находятся непосредственно после последнего символа строкового поля Name, размер которого зависит от длины имени конкретного типа. Поскольку смещение поля TypeData зависит от конкретных данных — имени типа, такую структуру нельзя непосредственно представить на Паскале. Приходится писать специальный код для доступа к этому полю – функцию `GetTypeData`.

Подобные трудности испытывают все авторы книг по форматам файлов данных при попытке описать эти данные, например, на языке C. Причина всех этих проблем — *в использовании для спецификации форматов данных языка, предназначенного для описания типов переменных*. В то же время, если отвлечься от необходимости придерживаться ограничений на типы переменных, то можно естественным образом поддержать в языке описание

достаточно сложных зависимостей между элементами данных. Именно этот подход использован при проектировании языка FlexT.

3.3 Использование механизма определения типов данных

Таким образом, в качестве основного элемента языка спецификации форматов данных мы будем рассматривать механизм определения типов. В процедурных языках программирования механизм определения типов можно считать отдельным вложенным языком, т.к. типы данных влияют, например, на семантику и синтаксис процедур для работы с данными этих типов, но не наоборот. Если в процедурных языках программирования основная информация программы содержится в коде процедур, то при описании способов хранения информации основная нагрузка ляжет именно на механизм определения типов. Рассмотрим, какими возможностями должен обладать такой механизм.

При идентификации типов данных любые физические данные будем рассматривать как набор элементов данных. Каждый такой элемент характеризуется своим адресом, размером и интерпретацией (типом). Составные элементы данных разбиваются на более мелкие элементы. Всю информацию, которую должна предоставить спецификация идентификации типов данных о каждом элементе данных, можно разбить на набор утверждений о том:

1. На какие составляющие, каких типов этот элемент разбивается;
2. Где находятся эти составляющие (каковы их адреса);
3. Сколько места они занимают.

При этом из того, что в рассматриваемых данных содержится некоторый элемент составного типа, выводится информация о типах и размещении его составляющих (из которой, в свою очередь, может выводиться информация о размере этого составного элемента).

Составляющие элемента данных могут размещаться либо на некотором фиксированном смещении от его начала, либо на некотором смещении от конца другой составляющей. Составной элемент некоторого типа может содержать либо фиксированное число составляющих, либо это число может определяться динамически, т.е. отличаться у разных экземпляров одного типа. Если число составляющих фиксировано, то их можно поименовать, в этом случае элемент данных можно описать как запись с полями — составляющими. При динамическом определении набора составляющих можно рассмотреть два основных случая: 1) когда содержание части полей определяет, какие ещё поля входят в состав записи — этот случай описывается при помощи вариантных типов данных; 2) когда число составляющих определяется динамически — при этом, т.к. размер спецификации конечен, она должна содержать итеративные элементы. Эти

итеративные элементы можно выделить в массив. Более сложные случаи могут быть описаны при помощи комбинации массивов и вариантов. Т.е. можно рассчитывать, что для описания произвольных структурированных составных элементов данных достаточно использовать конструкторы записи, массива и варианта (в широком смысле).

Кроме механизма описания структуры отдельного элемента данных нужны ещё механизмы для описания зависимостей между различными элементами. Наиболее важным случаем такой зависимости является указатель — элемент данных, в котором закодирована информация об адресе и типе другого элемента данных. Также следует учесть, что для правильной интерпретации одних элементов данных может потребоваться информация, содержащаяся в других элементах. Например, запись может содержать поля `Счётчик` и `Таблица`, где поле `Таблица` является массивом, число элементов которого записано в поле `Счётчик`. Для описания подобных случаев используется механизм параметризации типов данных.

3.4 Краткое описание языка FlexT

В текущей реализации интерпретатора FlexT программа компилируется после загрузки разбираемых данных. Это создаёт некоторые сложности при попытке использования спецификаций для других целей, например, для генерации кода чтения. С другой стороны, при таком подходе упрощается написание спецификаций за счёт возможности использования динамических выражений, например, в директивах условной компиляции, а именно максимальная простота написания спецификаций и является целью создания языка.

Дадим несколько пояснений, необходимых для понимания следующих далее фрагментов кода. Язык FlexT является нечувствительным к регистру. Перевод строки может быть разделителем, если он происходит в месте возможного окончания конструкции (величина отступа при этом значения не имеет). Вложенные определения типов можно брать в круглые скобки, тем самым решается вопрос о принадлежности блоков дополнительной информации о типе.

3.4.1 Выражения, параметры и свойства типов

Типы данных могут иметь ряд свойств, набор которых зависит от конструктора этого типа, например, размер и число элементов у массива, или номер случая у варианта. Каждый тип имеет свойство Size (Размер). Свойства могут задаваться конкретным значением при определении типа, либо некоторым выражением, которое вычисляет значение данного свойства через значения и/или свойства вложенных элементов данных сложного типа, а также, м.б. через значения параметров типа. Некоторые правила для вычисления свойства Размер могут автоматически добавляться компилятором, если они не указаны явно, это касается, например, случая, когда известен

размер всей записи и всех её полей, кроме последнего. Параметры в объявлении типа представляют ту информацию, которую необходимо указать дополнительно при использовании (вызове) данного типа.

Выражения для значений параметров и свойств типов оцениваются интерпретатором в ленивом режиме [24], т.е. они вычисляются только при необходимости использования такого значения. В некоторых случаях это позволяет избежать переполнения стека.

3.4.2 Ссылки на свойства в выражениях

Выражения вычисляются в контексте элемента данных некоторого типа, ссылка на этот элемент обозначается символом `'@'`, при этом `'@' ':' <Имя>` означает ссылку на параметр или свойство <Имя> данного элемента. Также в выражениях могут использоваться специальные конструкции: `<Адресное выражение>'@'` — ссылка на родительский элемент, т.е. на элемент данных того типа, в контексте которого данный тип определён. Пример:

```
T1 struc
  int Cnt1
  int Cnt2
  array[@.Cnt1] of (array[@@.Cnt2] of word) Tbl
ends
```

Выражения для параметра конструктора типа вычисляются в контексте вызова этого конструктора, поэтому значение свойства `Count` конструктора главного массива `Tbl` вычисляется в контексте типа `T1`, а для вложенного массива — в контексте типа массива `Tbl`, поэтому требуется написать `@@.Cnt2`, чтобы сослаться на поле `Cnt2` в записи – родителе главного массива.

`<Переменная>':' '@'` — ссылка на элемент даных-хозяин, т.е. на элемента данных, в который вложен рассматриваемый элемент. Т.к. некоторый тип может использоваться в разных составных типах, чтобы воспользоваться этой конструкцией, необходим оператор приведения к типу с проверкой: `<Переменная> 'AS' <Тип>`.

`'@' ':' '#'` - номер вложенного элемента данных в содержащем его элементе, например, индекс массива.

3.4.3 Блоки с дополнительной информацией о типе

Для определения различных типов данных в языке существуют конструкторы (массива, записи, варианта, и т.д.), синтаксис записи которых существенно различается. В то же время есть ряд задач, которые требуется решать для любых типов, независимо от того, при помощи каких конструкторов они определяются. Для этих целей используются блоки с дополнительной

информацией о типе, которые могут быть записаны через `':'` после любого определения типа. К этим блокам относятся:

Табл. 1. Блоки с дополнительной информацией о типе

Table 1. The blocks of data type additional information

Блок	**Пример**	**Описание**
утверждений	`:[@:Size=@.Len, @.offset:Cnt=@.count]`	Позволяет задать значения свойств типа данных или параметров его составляющих с использованием параметров типа или информации о его составляющих
условий корректности	`:assert[@.Op>=0x80]`	Задаёт логическое условие, которое должно выполняться для данных, относящихся к этому типу
отображения	`:displ=('#',HEX(2*@))`	Позволяет переопределить способ отображения значений типа
именования	`:autoname=('sec_',@.tag)`	Аналогичен displ, но служит для задания имён переменных
дополнительного свойства типа	`:let Val=(@.0)exc(@.1)`	Позволяет задать способ вычисления свойств типа, которые далее могут использоваться в выражениях

Блок утверждений можно использовать, чтобы связать параметры типа со значениями его свойств в том случае, когда конструктор типа не позволяет это сделать. Например, конструктор массива позволяет задать количество элементов, но не суммарный размер массива. Также этот блок используется, чтобы задать свойства определяемого типа по значениям его составляющих. Т.к. компилятор FlexT является однопроходным, в выражениях для

параметров типа нельзя сослаться на ещё не прочитанные составляющие, поэтому, чтобы, например, задать значение параметра поля записи по значению следующего за ним другого поля следует использовать блок утверждений записи.

Блок именования может использоваться для переменных, положение которых в памяти было найдено при помощи указателей. По умолчанию в этом случае переменная обозначается квалификатором доступа к ней через цепочку указателей, который может оказаться очень длинным и не слишком информативным. При наличии блока именования вместо этого квалификатора будет использоваться сформированная этим блоком строка. Эту возможность удобно использовать, например, когда в состав данных типа входит информация о некотором имени объекта, который эти данные описывают.

Блок дополнительного свойства типа позволяет описать способ вычисления некоторого значения, закодированного этим типом данных. Например, во многих бинарных форматах используется приём, когда целое число может быть представлено различным количеством байтов в зависимости от его значения. Выражение, описывающее декодирование такого числа, может быть задано, как дополнительное свойства этого типа данных. При работе с данными значение этого свойства так же, как и остальных свойств, вычисляется в ленивом режиме: при первом обращении с последующим запоминанием.

3.5 Типы данных языка FlexT

Основные конструкторы типов данных языка FlexT приведены в следующей таблице:

Табл. 2. Типы данных языка FlexT

Table 2. The FlexT language data types

Тип	Пример	Описание
Целые числа	`num-(6)`	Характеризуются размером и наличием знака, для знаковых чисел используется дополнительный код.
Пустой	`Void`	Тип с размером 0, позволяет пометить место в памяти.
Символьные	`char, wchar, wcharr`	В выбранной кодировке или Unicode с разными порядками байтов

Перечислимый	**enum** byte (A=1,B,C)	Задаёт наименования константам базового типа данных

Перечисление термов	**enum** TBit8 **fields** (
	  R0: TReg @0.3,...)
	of(
	  rts(R0) = 000020_,...)
		Облегчает описание кодирования машинных команд, т.к. в основном для них используется этот приём: в составе целого числа выделяются битовые поля, использование которых определяется остальными битами числа

Множество	**set** 8 **of** (OLD ^ 0x02, ...)	Даёт наименование битам, биты могут задаваться своим номером (символ '=' после имени) или маской (символ '^')

Запись	**struc**
	  Byte Len
	  array[@.Len]**of** Char S
	ends
		Последовательное размещение именованных и, возможно, разнотипных составляющих в памяти

Вариант	**case** @.Kind **of** vkByte: Byte **else** ulong **endc**	Выбор типа содержимого по заданной в параметрах (внешней) информации

Проверка	**try**
	  FN: TFntNum
	  Op: TDVIOp
	Endt
		Выбор типа содержимого по внутренней информации: выбирается первый тип, для которого выполняется условие корректности

Массив	**array**[@.Len]**of** str **array of** str ?@[0]= 0!byte;	Последовательное размещение однотипных составляющих в памяти (размеры которых могут различаться). Может задаваться количество элементов, общий размер массива, или стоп-условие.

Сырые данные	`raw[@.S]`	Неинтерпретируемые данные, отображаются как hex-дамп
Выравнивание	`align 16 at &@;`	Пропуск неиспользуемых данных для выхода на кратное заданному значению смещение относительно базового адреса
Указатель	`^TTable near=DWORD, ref=@:Base+@;`	Использует значение базового типа, для задания адреса (для файлов – смещения от начала) данных указанного типа в памяти.
Предваритель-ное объявление	`Forward`	Используется при циклических зависимостях между типами данных
Машинные команды	`codes of TOpPDP ?(@.Op >=TWOpCode.br)and...;`	Используется для дизассемблирования машинных команд

Помимо рассмотренных конструкторов типов данных вызов типа с подстановкой фактических параметров на место формальных рассматривается как специальный конструктор типа и может сопровождаться своими блоками дополнительной информации, например, блоком отображения. Синтаксически вызов типа отличается от ссылки на тип по наличию круглых скобок после его имени (даже когда у типа нет параметров). При вызове типа параметры могут указываться либо позиционно, либо по имени (как при вызове методов интерфейсов COM). Таким образом, часть параметров вызова типа может задаваться при его определении, а другая часть – в блоке утверждений.

Все типы данных могут иметь как байтовое, так и битовое размещение, в зависимости от настроек блока типов. Кроме того, при определении нового типа учитывается установленный порядок байтов. Порядок байтов влияет на вновь определяемые типы, но не на их вызовы. В большинстве форматов используется конкретный порядок байтов, поэтому его достаточно установить один раз в начале спецификации, если это – MSB, а LSB не требует и этих действий, т.к. он установлен по умолчанию. Существуют такие форматы, как TIFF, где порядок байтов выбирается для всего файла в зависимости от его сигнатуры. Однако пример формата Shape показывает, что существуют и форматы со смешенным использованием разных порядков байтов.

3.6 Блоки определений

Программа на FlexT состоит из нескольких видов блоков определений, в которых определения разделяются переводом строки. Основные блоки определений:

1) Блок констант

```
'const' nl
{<Имя константы> '=' <Int Expr> ';', nl}*
```

В выражениях для вычисления значений констант могут использоваться переменные из блока данных, поэтому значения некоторые из констант могут быть динамическими, т.е. зависящими от обрабатываемых данных.

2) Блок типов

```
'type' ['bit'] nl
{<Имя типа> <Определение типа>, nl}*
```

содержит определения типов. Признак `bit` указывает на то, что все определяемые в этом блоке типы будут иметь битовое, а не байтовое размещение.

3) Блок данных

```
'data' [<Блок>] nl
{<Int Expr> [';'] <Определение типа> <Имя>, nl}*
```

содержит определения переменных описываемого формата. `<Int Expr>` задаёт адрес переменной в гдавном адресном пространстве, если имя блока не указано, или смещение в блоке памяти `<Блок>`. Так же, как и при определении констант, выражение для адреса может содержать обращения к другим переменным, так что адрес переменной может определяться динамически. `<Определение типа>` должно задавать тип без свободных параметров, т.е. должны быть определены все свойства базового конструктора этого типа и типов его составляющих, если таковые имеются. После выражения для адреса можно ставить признак его окончания ';'. Это необходимо делать лишь в том случае, если начало <Определения типа> может быть воспринято, как продолжение выражения.

4) Блок кода

```
'code' ['(' <Имя типа кода> ')'] [<Блок>] nl
{<Int Expr> [';'] <Имя>, nl}*
```

содержит начальные адреса и названия частей кода в блоке памяти `<Блок>` или в главном адресном пространстве, если имя блока не указано. При указании имени типа кода память, начиная с указанных точек входа, разбирается в соответствии с этим типом данных. Тип кода должен быть задан конструктором `CODES` и быть полностью

определённым, при этом он описывает кодирование машинных инструкций и признаки команд, завершающих код (таких как RET или JMP). Если тип кода не задан, то используется код процессоров Intel 80x86.

Специальное имя `'-'` означает, что указанный адрес задаёт не начало, а конец части кода. Эта возможность оказывается полезной в тех случаях, когда дизассемблер не может самостоятельно правильно определить окончание последовательности команд. Это может произойти, например, в следующем случае:

```
CALL Terminate
<Данные>
```

Здесь процедура Terminate всегда вызывает функцию API для завершения работы программы.

3.7 Условная компиляция

Для описания форматов данных, которые могут иметь вариации, зависящие от версии формата или других его особенностей удобно использовать условную компиляцию. В условиях для выбора вариантов кода могут использоваться выражения, зависящие от обрабатываемых данных, поэтому версии структур данных автоматически определяются по уже прочитанным переменным. Этот механизм является очень эффективным при использовании спецификаций на FlexT для описания сложных форматов с долгой историей. Он не увеличивает сложность структур данных интерпретатора в отличие от использования вариантных типов данных. Однако, если в спецификации используется условная компиляция, то её очень неудобно обрабатывать в отсутствии примера данных. Таким образом, роль условной компиляции для FlexT можно сравнить с ролью препроцессора для языка C: позволяет получить эффективный конечный результат, но усложняет анализ кода. Для описания большинства форматов данных условная компиляция не требуется, но в некоторых случаях её нечем заменить. Возможно, что для получения универсального кода чтения данных по спецификации, содержащей условную компиляцию, должны быть разработаны более сложные алгоритмы трансляции, добавляющие вариантные элементы в структуры данных и модифицирующие выражения со ссылками на такие вариантные составляющие.

```
data
  Ident:Size TEHdr Hdr

type
%$IF Hdr.e_machine=TEMachine.EM_386;
  include elf_386.rfi
%$ELSIF Hdr.e_machine=TEMachine.EM_SPARC;
  include elfSPARC.rfi
%$ELSIF Hdr.e_machine=TEMachine.EM_M32;
  include elf_M32.rfi
%$ELSIF Hdr.e_machine=TEMachine.EM_PPC;
  include elf_ppc.rfi
%$ELSE
type
  TE_R_TYPE byte
  TE_Machine_Flags EWord
%$END Machine
```

Рис. 1. Пример использования условной компиляции

Fig. 1. An example of usage of conditional compilation

На врезке приведён пример использования условной компиляции (выдержка из описания формата ELF – загрузочных и объектных модулей Unix). Здесь в зависимости от типа процессора подключается соответствующая спецификация перечислимого типа `TE_R_TYPE` и множества `TE_Machine_Flags`, используемых в описании таблицы перемещений.

3.8 Алгоритм работы интерпретатора

Процесс разбора содержимого бинарного файла по спецификации интерпретатором FlexT состоит из следующих шагов:

1) Отображение (file mapping) бинарного файла в память.

2) Чтение спецификации, в ходе которого создаются объявленные в блоках `data` переменные и запоминаются адреса кода из блоков `code` (создаются части кода с нулевой длиной). Ранее объявленные переменные могут использоваться в зависящих от данных выражениях спецификации. Если в этих выражениях используется разыменование указателей, то на процесс чтения могут влиять и отличные от объявленных переменных элементы данных, доступные через эти указатели.

3) Обход элементов данных в поисках указателей. Выполняется только для элементов данных тех типов, в состав которых указатели входят. При обнаружении указателя создаётся переменная того типа, на который ссылается этот указатель, если такая переменная не была

создана раньше. После этого выполняется обход составляющих новой переменной. Кроме того, если структуры данных содержат указатели на код, то обнаруженные адреса добавляются в список частей кода.

4) Статическое дизассемблирование кода. Для каждой ещё не обработанной части кода выполняется обход её команд. Обход заканчивается по достижении, либо команды с выполненным стоп-условием (например, `ret` или `jmp`), либо следующей части кода. При обходе команды проверяются на наличие переходов и ссылок на данные. Для обнаруженных переходов (ссылок на код) также создаются новые части кода, если они не попадают внутрь существующих, иначе существующая часть кода разбивается на две. В результате формируется список частей кода, каждая из которых содержит непрерывную последовательность команд без внутренних точек входа.

5) Вывод результатов. Выполняется обход с распечаткой содержимого обнаруженных переменных и частей кода в порядке их адресов. Оставшиеся неразобранными блоки памяти между последовательными переменными и частями кода распечатываются в виде шестнадцатеричного дампа (при желании отображение этих фрагментов памяти можно отключить).

4. Использование спецификаций для анализа машинного кода

В предшествующих главах уже упоминались типы данных и шаги алгоритма разбора, предназначенные для работы с машинными командами. Рассмотрим эти возможности языка FlexT в одном месте и более подробно. Машинные команды используются не только для файлов с исполняемым кодом для реальных процессоров (таких, как Intel 80x86, ARM), но и для виртуальных машин (Java VM, .NET MSIL). Кроме того, некоторые форматы, например, шрифты TTF или файлы инсталляторов, могут содержать байт-код для бинарного представления используемых там сценариев. Таким образом, исследование файлов многих форматов будет неполным без отображения содержащихся там машинных инструкций.

В большинстве случаев представление машинной команды состоит из постоянной части, задающей вид команды и переменной части, задающей её аргументы, если таковые имеются. В качестве описания постоянной части для байт-кодов виртуальных машин, как правило, достаточно использовать перечислимый тип данных, поскольку там просто содержится целочисленный код команды (формат TTF является исключением из этого правила), при этом переменную часть удобно описать при помощи вариантного типа по коду команды.

Для машинных команд реальных процессоров используется более плотное кодирование, поэтому в постоянной части для некоторых команд выделяются битовые поля, содержащие часть аргументов, например, номера регистров. Будем называть сочетания таких полей с кодом инструкции термами. Описать кодирование термов можно было бы при помощи типа данных «проверка» и ряда вспомогательных битовых типов, но такое описание будет очень громоздким и неэффективным (при попытке сделать это на практике терпения хватило лишь на описание десятка команд). Поэтому, под влиянием проекта [25][26] в язык FlexT включён специальный тип данных «перечисление термов». В определении такого типа сначала задаётся список всех битовых полей, применяемых для кодирования команд (каждое поле характеризуется битовыми смещением, размером и типом), а затем описываются сами термы с указанием того, какие из этих полей они используют. Вместе с термом указывается его база – число, задающее значения не входящих в поля битов. В записи базы терма допускается использовать подчерки для обозначения переменной части — битов, входящих в состав полей (такая маска проверяется на соответствие упоминаемым в терме полям). В реализации типа данных «перечисление термов» используется дерево решений, позволяющее быстро найти терм по его коду. В выражениях можно обращаться к любым полям перечисления термов, как к полям записи, однако, если у выбранного терма это поле не используется, то такое обращение вызывает ошибку вычисления этой части выражения.

Если системы команд RISC-процессоров состоят только из постоянной части, то для описания команд CISC-процессоров требуется задавать кодирование переменной части в зависимости от терма, выбранного в перечислении термов. Для этих целей реализована разновидность типа данных «вариант» с выбором содержимого по значению перечисления термов. В определении такого типа база терма в списке выбора обозначает все соответствующие этому терму значения постоянной части. Для выбора варианта в этом случае также строится дерево решений, но лишь по тем термам, которые упоминаются в определении типа.

После описания кодирования отдельной команды необходимо определить кодовый тип. В интерпретаторе FlexT кодовый тип реализуется на базе массива со стоп-условием и записывается аналогичным образом, за исключением использования идентификатора `codes` вместо `array`. При этом стоп-условие используется для выделения команд, завершающих последовательность, например, команд безусловного перехода или выхода из процедуры. В описаниях команд, вызывающих переход, должны использоваться указатели на определяемый кодовый тип, которые будут отслеживаться дизассемблером для обнаружения других частей кода. Кроме того, у типа команд может быть определено, как в приведённом примере, вычисляемое свойство `isCall`, которое в этом случае используется

дизассемблером для различения команд перехода и вызовов процедур (влияет на префиксы генерируемых меток).

```
type bit
TBit num+(1)
TBit2 num+(2)
…
TsBit24 num-(24)
TRN enum TBit4 (R0,R1,… R12,SP,LR,PC)
…
TShiftOp enum TBit2 (SHL,SHR,ASR,ROR)
TBrOfs24 ^TOpARMSeq NIL- =TsBit24 REF=
  (&(@:@)+4+@*4);
type
TARMOpCode enum TBit32 fields (
  Cond: TCond @28.4, //Op.execution condition
  Rn: TRN @16.4, //Source register
  Rd: TRN @12.4, //Destination register
  Shift: TShiftOp @5.2,
…
  SM: TBit @20.1
) of (
 //Data processing rotate right with extend
  _DPRRX(cond,ArOp,S,Rn,Rd,Rm) =
    0b____000_I________I____0000I0110____,
 //Move status register to register
  MSR(cond,R,Rd) =
    0b____0001I0_001111I____0000I00000000,
  …
 //Branch and Branch with link
  Branch(cond,BL,BrOfs) =
    0b____101_I________I________I________,
 //Software interrupts
  SWI(cond,SWIOfs) =
    0b____1111I________I________I________
):let IsStop=(((@.BL=0)and(@.cond>=
  TCond.AL)) exc 0
);:let isCall=((@.BL=1) exc 0);
TOpARMSeq codes of TARMOpCode ?@:isStop;:
  displ=('(', ShowArray(@,(NL,HEX(&@,4),': ',
  @)),NL,')')
```

Рис. 2. Фрагменты спецификации кодирования команд процессора ARM

Fig. 2. Excerpts from the instruction encoding specification of the ARM processor

На врезке приведены фрагменты спецификации кодирования команд процессора ARM. Многоточием обозначены пропущенные фрагменты, аналогичные оставшимся соседним. Сначала определяются битовые типы данных, используемых в командах полей, включая перечислимые типы для декодирования значений тех из этих полей, которые задают регистры, конкретные операции из семейства однотипных, условия перехода и т.д. Также на базе типа 24-битных знаковых чисел определяется тип указателя

`TBrOfs24`, используемый в командах перехода. В выражении для вычислении адреса, на который ссылается указатель, в качестве базы используется адрес владельца (`&(@:@)`), т.е. перечисления термов, в состав которого входит указатель. Далее определяются перечисление термов `TARMOpCode`, описывающее кодирование отдельной команды и кодовый тип данных `TOpARMSeq` на базе этих команд.

Описания кодирования машинных команд позволяют использовать разработанные инструменты анализа бинарных файлов для новых процессоров. Использование описаний кодирования машинных команд, унифицированных с описаниями остальных типов данных, позволяет применять для исследования команд все средства, разработанные для анализа данных. Например, можно воспользоваться механизмами поиска структур данных для обнаружения фрагментов кода заданного вида и получения отчёта по найденным адресам. Имеется опыт применения этого подхода для получения описаний типов данных интерпретатора посредством поиска всех вызовов процедуры регистрации типа в коде исполняемого файла интерпретатора. В некоторых случаях непосредственные аргументы команд могут интерпретироваться, как указатели, что позволяет выделять в памяти соответствующие переменные. С использованием блока отображения для типа данных команды можно при необходимости довести способ отображения команды до принятого в дизассемблерах, а можно добавить вывод дополнительных сведений о её свойствах или составляющих.

С другой стороны, во многих исполняемых файлах присутствуют структуры данных, например RTTI или таблицы виртуальных методов, содержащие адреса кода. Описание таких элементов данных на FlexT как указателей на код позволяет обнаружить этот код и пометить его как относящийся к определённым данным, что существенно облегчает дальнейшее понимание программы.

5. Применение основанных на FlexT инструментов для анализа бинарных файлов

Для работы с бинарными данными реализован ряд инструментов, основанных на использовании написанных на FlexT спецификаций: BinView, BinExpl, ExeXpl, а также Web-приложение, доступное по ссылке [27].

Консольная программа BinView позволяет получить результат разбора бинарного файла в одном из текстовых форматов: просто текст, HTML, RTF, TeX. Формат обрабатываемого файла определяется автоматически по его расширению и содержимому. Для автоматического поиска спецификации формата используется следующая логика: проверяется файл с именем <Расширение>`.rfh`, если таковой имеется в текущем каталоге или в каталоге библиотеки спецификаций. Далее проверяются все описания форматов, сопоставленные расширению в файле `ref.cfg` из библиотечного каталога.

Для проверки соответствия обрабатываемого файла некоторой спецификации используются блоки утверждений `assert` из файла спецификации: заданные в этих блоках логические выражения оцениваются сразу после чтения и, если они оказываются ложными или ошибочными, то попытка применить спецификацию прекращается. Таким образом, если, например, блок утверждения находится сразу после объявления переменной для сигнатуры файла и значение этой переменной не соответствует ожидаемому, то на этом чтение спецификации заканчивается. В результате выбирается первая спецификация, для которой все утверждения выполняются. После чтения спецификации формата выполняется поиск дополнительной спецификации структур данных обрабатываемого файла, которая может находиться в файле <Имя файла>`.ref`. Таким образом, при необходимости можно описать структуры данных, обнаруженные в конкретном файле, но не отражённые в общей спецификации формата.

Сгенерированный программой BinView листинг может оказаться очень большим. В результате, например, браузер будет очень долго открывать полученный HTML файл. Программа BinExpl служит для интерактивного отображения результатов разбора бинарных файлов, по тем же спецификациям, что и BinView. При этом динамически генерируются фрагменты листинга, что позволяет просматривать содержимое очень больших файлов (основное ограничение на размер: файл должен целиком помещаться в 32-разрядное пространство адресов при использовании FileMapping).

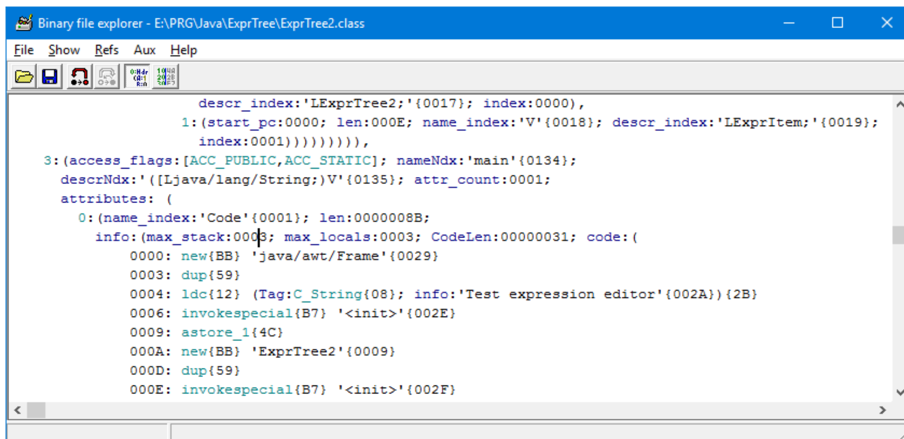


Рис. 3. Окно программы BinExpl при разборе файла класса Java

Fig. 3. The main form of the program BinExpl when parsing a Java class file

Программа ExeXpl служит для интерактивного исследования исполняемых файлов Windows. При этом спецификации на FlexT используются для

описания структур данных исследуемых программ и, в том числе, структур данных, характерных для конкретного компилятора (RTTI, таблиц виртуальных методов, обработчиков ошибок и т.д.). В ходе изучения программы пользователь может добавлять в спецификацию наименования для исследованных частей кода, что облегчает понимание тех фрагментов кода, которые используют уже описанные. Т.к. код выделяется по результатам статического анализа, не могут быть автоматически обнаружены, например, фрагменты, на которые ссылаются таблицы перехода, сгенерированные для операторов выбора (switch/case). Но такие таблицы могут быть описаны, как содержащие указатели на код структуры данных, что позволяет обнаружить оставшиеся неразобранными фрагменты кода.

С использованием спецификаций на FlexT реализован ряд механизмов поиска, которые невозможно выполнить другим способом: поиск структур данных, генерация кода по сценариям поиска данных, поиск элементов данных в файлах.

Поиск структур данных позволяет обнаружить фрагменты памяти, соответствующие заданному условию на выбранный пользователем тип данных. В ходе такого поиска алгоритм проверяет каждый адрес на возможность разместить в этом месте указанную структуру данных так, чтобы она отвечала заданному критерию, например, условию корректности `assert` искомого типа данных. Таким образом можно, например, найти все таблицы виртуальных методов классов. Этот процесс поиска можно оформить в виде сценария генерации отчётов об обнаруженных структурах данных. Для применения таких сценариев необходимо подключить к спецификации модули с определениями используемых при поиске типов данных, после чего в процессе поиска будет сформирован текстовый отчёт обо всех обнаруженных адресах в заданной в сценарии поиска форме (для этого применяется синтаксис блоков отображения типов). Этот механизм может использоваться для генерации фрагмента спецификации с описаниями найденных данных, например, на базе информации о генерируемых конкретным компилятором структурах данных. Также приходилось использовать этот механизм для получения информации о реализованных в интерпретаторе языка сценариев классах с информацией об их членах.

Механизм поиска элементов данных в файлах позволяет найти среди файлов с указанным расширением те, которые содержат данные указанного типа, отвечающие заданному условию. Здесь поиск идёт не по всей памяти, а среди составляющих переменных, найденных в файле при помощи спецификации. Этот механизма позволяет, например, найти метафайлы, в которых содержатся команды рисования окружности определённого размера.

6. Заключение

В данной работе рассмотрен язык спецификации интерпретации данных FlexT, который позволяет описывать достаточно широкий набор форматов данных при помощи простых синтаксических конструкций, являющихся расширением характерного для традиционных процедурных языков программирования набора конструкторов типов данных. Возможно также его использование для спецификации кодирования машинных команд. Реализованный интерпретатор использует спецификации для идентификации типов данных.

На языке FlexT с различной степенью завершённости было описано около сотни форматов данных, в том числе, исполняемых и объектных файлов, содержащих программный код. Приведём расширения некоторых из этих форматов: EXE (MZ,NE,LE), ELF, CLA, TPU, OBJ, Mach-o, TTF, HLP, SHP, DBF. Также конструкции языка использовались при декодировании различных программ и описании характерных для конкретных компиляторов форматов данных, например, RTTI Delphi и Visual Studio. Многие форматы удавалось описать практически с одного прохода, т.е. по мере чтения документации получалось сразу переводить содержащиеся там сведения в конструкции языка FlexT. При работе с описаниями форматов в редком из них не были обнаружены ошибки. Т.е. переход от описания для человека к описанию для машины, которое можно сразу же проверить на настоящих данных, приводит к существенному повышению достоверности информации, и в этом может состоять один из наиболее важных результатов применения рассматриваемого языка. Возможность интерпретации результатов разбора в качестве теста на соответствие спецификации и реальных данных, может применяться, как для проверки корректности данных по уже отлаженной спецификации, так и для проверки спецификации на соответствие примерам данных и, в том числе, для обратного проектирования недокументированных форматов.

Наиболее существенным ограничением для текущей версии языка является невозможность полного описания таких форматов, при интерпретации которых необходимо строить сложные вспомогательные структуры данных, непосредственно не представленные в файле. Например, такая необходимость часто возникает при описании сжатых данных, при декомпрессии которых используется динамическое построение словарей (как LZW), дерева Хаффмана, моделей контекстов (PPM) и других подобных структур. Более подробную информацию о языке FlexT можно найти по адресу [27].

Список литературы

[1]. Faase F.J. BFF: A grammar for Binary File Formats [Электронный ресурс] URL: http://www.iwriteiam.nl/Ha_BFF.html

[2]. Data Format Description Language (DFDL) [Электронный ресурс] URL: https://www.ogf.org/ogf/doku.php/standards/dfdl/dfdl

[3]. IBM Knowledge Center [Электронный ресурс] Data Format Description Language (DFDL) URL: http://www.ibm.com/support/knowledgecenter/SSMKHH_10.0.0/com.ibm.etools.mft.doc/df20060_.htm

[4]. IBM Integration Bus [Электронный ресурс] URL: http://www-03.ibm.com/software/products/en/ibm-integration-bus/

[5]. Daffodil: Open Source DFDL [Электронный ресурс] URL: https://opensource.ncsa.illinois.edu/confluence/display/DFDL

[6]. IBM Knowledge Center [Электронный ресурс] Unsupported features URL: http://www.ibm.com/support/knowledgecenter/SSMKHH_10.0.0/com.ibm.etools.mft.doc/df00150_.htm

[7]. WebLogic Integration 7.0 [Электронный ресурс] Building Format Definitions. URL: https://docs.oracle.com/cd/E13214_01/wli/docs70/diuser/fmtdef.htm

[8]. NetPDL Language Specification. http://www.nbee.org/doku.php?id=netpdl:index

[9]. BinPAC. https://www.bro.org/sphinx/components/binpac/README.html

[10]. The data description language EAST specification (CCSD0010). [Электронный ресурс] URL: http://mtc-m16c.sid.inpe.br/col/sid.inpe.br/mtc-m18@80/2009/07.21.13.31/doc/CCSDS%20644.0-B-2.pdf

[11]. Calder B.R., Masetti G. Huddler: a multi-language compiler for automatically generated format-specific data drivers. U.S. Hydrographic Conference (US HYDRO) 2015 Доступно по ссылке: http://www.hypack.com/ushydro/2015/papers/pdf/Calder_Huddler_for_automatic_data_drivers.pdf

[12]. Georgia Tech Research Institute [Электронный ресурс] Digital Archives Research URL: http://perpos.gtri.gatech.edu/

[13]. Underwood W. Grammar-Based Specification and Parsing of Binary File Formats. The International Journal of Digital Curation Vol. 7, No. 1, 2012, pp. 95-106 Доступно по ссылке: http://www.ijdc.net/index.php/ijdc/article/viewFile/207/276

[14]. Parr T. ANTLR (ANother Tool for Language Recognition) [Электронный ресурс] URL: http://www.antlr.org/

[15]. Godmar Back. 2002. DataScript - A Specification and Scripting Language for Binary Data. In Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering (GPCE '02), Don S. Batory, Charles Consel, and Walid Taha (Eds.). Springer-Verlag, London, UK, UK, 66-77.

[16]. DataScript [Электронный ресурс] URL: http://datascript.sourceforge.net/

[17]. Binary data definition language [Электронный ресурс] URL: http://www.binarydom.com/sdk/doc/bddl.shtml

[18]. Binopedia [Электронный ресурс] URL: http://binopedia.org/

[19]. Kaitai Struct [Электронный ресурс] URL: http://kaitai.io/

[20]. Synalyze It! [Электронный ресурс] URL: https://www.synalysis.net/

[21]. Hexinator [Электронный ресурс] URL: https://hexinator.com/hexinator-windows/

[22]. Synalyze It! [Электронный ресурс] The Grammar Page. https://www.synalysis.net/formats.xml

[23]. Лисков Б., Гатэг Дж. Использование абстракций и спецификаций при разработке программ: Пер. с англ. - М.:Мир, 1989.

[24]. Филд А., Харрисон П. Функциональное программирование: Пер. с англ. - М.:Мир, 1993

[25]. Ramsey N., Fernandez M.F. 1995. The New Jersey machine-code toolkit. In Proceedings of the USENIX 1995 Technical Conference Proceedings (TCON'95). USENIX Association, Berkeley, CA, USA, 24-24.

[26]. Ramsey N., Fernandez M.F. The New Jersey Machine-Code Toolkit [Электронный ресурс] URL: http://www.cs.tufts.edu/~nr/toolkit/

[27]. Хмельнов А.Е. Главная страница по языку FlexT. http://hmelnov.icc.ru/FlexT/

A declarative language FlexT for analysis and documenting of binary data formats

A.Y. Hmelnov <hmelnov@icc.ru>
I.V. Bychkov <bychkov@icc.ru>
A.A. Mikhailov <mikhailov@icc.ru>
Matrosov Institute for System Dynamics and Control Theory of the Siberian Branch of the Russian Academy of Sciences,
134, Lermontova st., Irkutsk, 664033, Russia

Abstract. The language FlexT (Flexible Types) is intended for specification of binary data formats. The language is declarative and designed to be well understood for human readers. Its main elements are the data type declarations, which look very much like the usual type declarations of the imperative programming languages, but are more flexible. In the article we first give a review of the capabilities of the modern projects oriented to specification of binary file formats. Then we consider the main features of the FlexT language and, in particular, the features that help to describe the formats of encoding of machine instructions. Finally we briefly describe the software developed, which is based upon the FlexT interpreter and some new capabilities of information search, which makes possible the use of the specifications.

Keywords: specifications of binary data formats, specification of encoding of machine instructions, declarative language, disassembler

DOI: 10.15514/ISPRAS-2016-28(5)-15

For citation: A.Y. Hmelnov, I.V. Bychkov, A.A. Mikhailov. A declarative language FlexT for analysis and documenting of binary data formats. *Trudy ISP RAN/Proc. ISP RAS*, vol. 28, issue 5, 2016. pp. 239-268 (in Russian). DOI: 10.15514/ISPRAS-2016-28(5)-15

References

[1]. Faase F.J. BFF: A grammar for Binary File Formats. http://www.iwriteiam.nl/Ha_BFF.html
[2]. Data Format Description Language (DFDL). https://www.ogf.org/ogf/doku.php/standards/dfdl/dfdl
[3]. IBM Knowledge Center. Data Format Description Language (DFDL). http://www.ibm.com/support/knowledgecenter/SSMKHH_10.0.0/com.ibm.etools.mft.doc/df20060_.htm
[4]. IBM Integration Bus. http://www-03.ibm.com/software/products/en/ibm-integration-bus/
[5]. Daffodil: Open Source DFDL. https://opensource.ncsa.illinois.edu/confluence/display/DFDL

[6]. IBM Knowledge Center. Unsupported features. http://www.ibm.com/support/knowledgecenter/SSMKHH_10.0.0/com.ibm.etools.mft.doc/df00150_.htm

[7]. WebLogic Integration 7.0. Building Format Definitions. https://docs.oracle.com/cd/E13214_01/wli/docs70/diuser/fmtdef.htm

[8]. NetPDL Language Specification. http://www.nbee.org/doku.php?id=netpdl:index

[9]. BinPAC. https://www.bro.org/sphinx/components/binpac/README.html

[10]. The data description language EAST specification (CCSD0010). http://mtc-m16c.sid.inpe.br/col/sid.inpe.br/mtc-m18@80/2009/07.21.13.31/doc/CCSDS%20644.0-B-2.pdf

[11]. Calder B.R., Masetti G. Huddler: a multi-language compiler for automatically generated format-specific data drivers. U.S. Hydrographic Conference (US HYDRO) 2015 Available at URL: http://www.hypack.com/ushydro/2015/papers/pdf/Calder_Huddler_for_automatic_data_drivers.pdf

[12]. Georgia Tech Research Institute. Digital Archives Research. http://perpos.gtri.gatech.edu/

[13]. Underwood W. Grammar-Based Specification and Parsing of Binary File Formats. The International Journal of Digital Curation Vol. 7, No. 1, 2012, pp. 95-106 Available at URL: http://www.ijdc.net/index.php/ijdc/article/viewFile/207/276

[14]. Parr T. ANTLR (ANother Tool for Language Recognition). http://www.antlr.org/

[15]. Godmar Back. 2002. DataScript - A Specification and Scripting Language for Binary Data. In Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering (GPCE '02), Don S. Batory, Charles Consel, and Walid Taha (Eds.). Springer-Verlag, London, UK, UK, 66-77.

[16]. DataScript. http://datascript.sourceforge.net/

[17]. Binary data definition language. http://www.binarydom.com/sdk/doc/bddl.shtml

[18]. Binopedia. http://binopedia.org/

[19]. Kaitai Struct. http://kaitai.io/

[20]. Synalyze It!. https://www.synalysis.net/

[21]. Hexinator. https://hexinator.com/hexinator-windows/

[22]. Synalyze It! The Grammar Page. https://www.synalysis.net/formats.xml

[23]. B. Liskov, J. Guttag, Abstraction and Specification in Program Development, The MIT Press, 1986.

[24]. Field A.J., Harrison P.G. Functional Programming, Addison-Wesley, Wokingham, UK, 1988

[25]. Ramsey N., Fernandez M.F. 1995. The New Jersey machine-code toolkit. In Proceedings of the USENIX 1995 Technical Conference Proceedings (TCON'95). USENIX Association, Berkeley, CA, USA, 24-24.

[26]. Ramsey N., Fernandez M.F. The New Jersey Machine-Code Toolkit. http://www.cs.tufts.edu/~nr/toolkit/

[27]. Hmelnov A.Y. The home page of FlexT. http://hmelnov.icc.ru/FlexT/