Санкт-Петербургский Государственный Университет
Математико-механический факультет

Кафедра системного программирования

# Верификация дизассемблера x86-64

Курсовая работа студента 361 группы
*Тенсина Егора Дмитриевича*

Научный руководитель: .....
ст. преподаватель *Баклановский М. В.*

Санкт-Петербург

2012

# Содержание

# 1 Введение

*Дизассемблирование* — трансляция машинного кода в код на языке ассемблера. Дизассемблирование выполняется *дизассемблером*. Дизассемблирование инструкции состоит из двух этапов:

- *декодирование* инструкции из последовательности байт в памяти;
- представление декодированной инструкции на языке ассемблера.

Декодер можно представить в виде конечного автомата, который последовательно читает байты в памяти, пока не окажется в одном из конечных состояний:

- инструкция декодирована и допустима;
- инструкция недопустима.

Ясно, что такой декодер имеет огромное количество состояний, что повышает вероятность ошибки и сложность программирования.

## 1.1 История проекта

В рамках проекта «Профайлер ядра MS WS 2008» [13] [11] [12] от компании EMC [1] был разработан прототип профайлера, инструментирующий функции kernel API, предоставляющие доступ к спинлокам (*spin locks*). Используемая техника требует дизассемблер для декодирования инструкций, общая длина которых должна превзойти заданное число $n$. Было принято решение о написании собственного дизассемблера инструкций архитектуры x86-64. В связи с сложностью отладки встал вопрос о автоматической верификации произвольного дизассемблера инструкций архитектуры x86-64.

## 1.2 Обзор дизассемблеров

В таб. 1 приведен краткий список популярных дизассемблеров. Ни для одного из этих продуктов не было найдено исследований, подтвер-

| *Имя* | *Версия* | *Автор* | *Примечания* |
|---|---|---|---|
| OllyDbg [5] | 2.01 | Oleh Yuschuk | User-mode дебаггер; дизассемблер поддерживает только x86. |
| Objconv [4] | 2.12 | Agner Fog | Конвертер объектных файлов форматов COFF/PE, OMF, ELF и Mach-O. |
| IDA Pro [2] | 6.x | Hex-Rays | Профессиональный, дорогой дизассемблер. |
| gdb [8] | 7.4.1 | FSF | Стандартный дебаггер проекта GNU. |
| PEBrowse [6] | 10.1.4.4 | Russel Osterlund | Дизассемблер и парсер файлов формата PE. |

Таблица 1: Популярные дизассемблеры

ждающих корректность их работы и соответствие заявленому охвату пространства инструкций.

# 2 Постановка задачи

Пусть *Valid* — множество допустимых инструкций x86-64, и *Decoded* — множество декодированных дизассемблером инструкций. Тогда для верификации дизассемблера необходимо вычислить:

- $Valid \setminus Decoded$ — множество допустимых, но не декодированных инструкций;
- $Decoded \setminus Valid$ — множество ошибочно декодированных, недопустимых инструкций.

Для каноничного декодера получим пустое множество в обоих случаях.

# 3 Алгоритм

## 3.1 Эталонный декодер

*Эталонным декодером* является декодер процессора, реализующего архитектуру x86-64. Таким образом, множество *Valid* удобно строить с помощью процессора, на котором проводится верификация.

Конкретнее, для последовательности байт $\{x_i\},\ i = 1, 2, \ldots, n$ необходимо реализовать программу, проверяющую, является ли эта последовательность допустимой инструкцией.

Необходим судить о том, является ли эта последовательность допустимой инструкцией, по косвенным признакам. В этом качестве выступают *исключения* — оповещения процессора о исключительной ситуации (такой, например, как недопустимая инструкция). Интересующие нас исключения можно обрабатывать программно (см. 3.1.3).

### 3.1.1 Ограничения эталонного декодера

Заметим, что верно следующее: пусть $\{x_i\},\ i = 1, 2, \ldots, m,\ m < n$ — допустимая инструкция. Тогда $\{x_i\},\ i = 1, 2, \ldots, n$ — недопустимая инструкция. Аналогично, пусть $\{x_i\},\ i = 1, 2, \ldots, m,\ m \leq n$ — недопустимая инструкция. Тогда $\{x_i\},\ i = 1, 2, \ldots, n$ — недопустимая инструкция.

Пусть процессор декодировал и исполнил $m$ байт последовательности. Необходимо, чтобы $m < n$. В самом деле, пусть процессор исполнил $m > n$ байт. Тогда для какого-нибудь $n > 1$ существует такой $n_1 < n$, что $\{x_i\},\ i = 1, 2, \ldots, n_1$ — допустимая инструкция и $\{x_i\},\ i = n_1 + 1, n_1 + 2, \ldots, n, \ldots, n_2,\ n_2 \leq m$ — допустимая инструкция. Таким образом, процессор декодирует и исполнит 2 инструкции вместо 1-ой.

Несложно видеть, что тестирование инструкции длиной $n$ байт ни к чему не приводит: тестируемая последовательность может кодировать,

к примеру, ровно 2 допустимых инструкции. Аналогично, ни к чему не приводит и тестирование инструкции длиной $n-1$ байт $\forall n > 1$. Продолжая мысль, тестирование необходимо начинать с подпоследовательности длиной 1, продолжить с подпоследовательностью длиной 2 и т. д., произведя в итоге $m \leq n$ тестов.

Пусть далее $start_i$ — адрес первого байта декодируемой инструкции длины $n_i$, $end_i = start_i + n_i$.

Реализовать ограничения можно с помощью выделения для тестирования 2 смежных страниц в памяти с ограничениями доступа. Пусть далее $page_1$ — адрес первой страницы, $page_2$ — адрес второй страницы. Первая страница получает разрешение только на чтение и исполнение, ко второй же любой доступ запрещается. Изначально, инструкция длины $n$ размещается в $page_2 - 1$, сдвигаясь на 1 байт влево на каждом тесте. Таким образом, процессор на $i$-ом тесте ($i = 1, 2, \ldots, n$) не сможет исполнить более чем $i$ байт (помешают ограничения доступа ко второй странице).

### 3.1.2 Группы исключений

Исключения, возникающие во время декодирования и исполнения процессором тестируемой инструкции, можно поделить на группы:

- тестируемая инструкция недопустима (группа исключений FAIL);
- инструкция длины $m_i \leq n_i$ декодирована и выполнена (группа исключений SUCCESS);
- не удалось завершить выборку (группа исключений MORE).

Исключения групп FAIL и SUCCESS позволяют завершить работу программы. Исключение группы FAIL означает, что тестируемая инструкция не принадлежит множеству *Valid*. Исключение группы SUCCESS обрабатывается следующим образом:

- если $n_i = n$, то последовательность кодирует допустимую инструкцию и принадлежит множеству *Valid*;
- иначе последовательность кодирует недопустимую инструкцию и не принадлежит множеству *Valid*.

Исключения группы MORE обрабатываются следующим образом:

- если $n_i = n$ (нет больше байт для выборки), то последовательность кодирует недопустимую инструкцию и не принадлежит множеству *Valid*;
- иначе переходим к следующей подпоследовательности.

### 3.1.3 Обработка исключений с помощью SEH

*Structured Exception Handling* [7] — это механизм обработки аппаратных и программных исключений. С помощью SEH можно идентифицировать исключения и сортировать по группам (см. 3.1.2).

Исключение `access violation`, к примеру, возникает, если:

- инструкция пытается прочитать ячейку памяти без разрешения на чтение,
- инструкция пытается записать в ячейку памяти без разрешения на запись,
- процессор производит выборку из памяти без разрешения на чтение.

При чтении и записи ячейки памяти исключение принадлежит группе SUCCESS (возникло при выполнении уже декодированной инструкции). В случае с выборкой SEH предоставляет два свойства исключения: адрес инструкции, вызвавшей исключение ($addr_{instr}$), и адрес недоступной ячейки памяти ($addr_{unavailable}$).

- Если $addr_{unavailable} \neq end_i$ (произошла передача управления на адрес $addr_{unavailable}$), то это исключение относится к группе SUCCESS.

- Иначе:
  - если $addr_{instr} = start_i$ (выборка байт для тестируемой инструкции), то исключение принадлежит группе MORE;
  - иначе ($addr_{instr} = end_i$; тестируемая инструкция декодирована и выполнена) инструкция принадлежит группе SUCCESS.

Исключения `breakpoint` и `privileged instruction`, с другой стороны, всегда относится к группе SUCCESS, а исключение `illegal instruction` всегда относится к группе FAIL.

## 3.2 Программный декодер

Программный декодер не передает декодированную инструкцию на исполнение, что позволяет во многом упростить построение множества *Decoded* по сравнению с построением множества *Valid*.

В случае с программным декодером, благодаря наличию API для декодирования ровно одной инструкции, нет нужды в проведении нескольких тестов для декодируемой инструкции. Во время работы программного декодера может возникнуть лишь `access violation` при чтении по адресу $page_2$. В таком случае тестируемая инструкция не принадлежит *Decoded*.

После благополучного завершения работы программного декодера тестируемая инструкция добавляется в *Decoded* в зависимости от результата:

- если программный декодер декодировал ровно $n$ байт, тестируемая инструкция добавляется в *Decoded*;
- иначе, тестируемая инструкция не принадлежит множеству *Decoded*.

### 3.2.1 Автоматизированная верификация

Ясно, что верификация дизассемблера должна проводиться по возможности автоматизированно. С этой целью верификация программно-

го декодера производится единым способом — вызовом функции с заданным именем из динамически связываемой библиотеки. Таким образом, разработчику дизассемблера необходимо лишь предоставить библиотеку с реализацией требуемой функции.

## 3.3 Генерация тестируемых инструкций

Покрытие пространства возможных инструкций обеспечивается рациональной генерацией тестируемых инструкций. Полное покрытие пространства возможных инструкций возможно лишь с помощью *полного перебора* двоичных последовательностей, не превышающих заданную длину. Учитывая максимальную длину допустимой инструкции (15 байт согласно [9]), полный перебор займет $2^{120}$ итераций, что недопустимо.

*Генерация случайной последовательности*, с другой стороны, не обеспечивает покрытия пространства возможных инструкций. В свете того, однако, что нам предстоит *оптимизировать* случайный перебор, генерация случайных последовательностей позволяет протестировать инструкции, пропущенные в *оптимизированным полным перебором.*

### 3.3.1 Кодирование инструкции x86-64

Инструкция архитектуры x86-64 кодирует свою семантику и операнды. Семантика инструкции кодируется с помощью *префиксов* и *опкода.*

Опкод инструкции — последовательность байт, главным образом определяющая семантику инструкции. К примеру, опкод `0x90` идентифицирует инструкцию `nop`, а `0x0f05` — `syscall`.

Префиксы — набор байт, предшествующих опкоду. Префиксы влияют на семантику инструкции. К примеру, префикс `lock` (`0xf0`) гарантирует эксклюзивность доступа к ячейке памяти.

Операнд принадлежит к одному из следующих множеств:

- ячейки в памяти;
- значения в регистрах;
- целые числа.

Целые числа кодируются в самой инструкции как соответствующее представление числа в виде последовательности байт. Ячейки в памяти и значения в регистрах кодируются с помощью более сложного механизма.

Ячейка в памяти вида [значение в регистре + смещение] кодируется с помощью байта *ModRM*. Байт ModRM делится на:

- поле *mod* (2 бита), кодирующее длину смещения;
- поле *reg* (3 бита);
- поле *rm* (3 бита), кодирующее базовый регистр.

Ячейка в памяти вида [значение в регистре + множитель * значение в регистре + смещение] кодируется с помощью байтов ModRM и *SIB*. Байт SIB делится на:

- поле *scale* (2 бита), кодирующее множитель;
- поле *index* (3 бита), кодирующее домножаемый регистр;
- поле *base* (3 бита), кодирующее базовый регистр.

Значения в регистрах кодируются либо с помощью поля ModRM.reg, либо с помощью поля ModRM.rm (если ModRM.mod == `11b`). Таким образом, максимум два операнда могут быть значениями регистров.

Точно кодирование инструкции в архитектуре x86-64 определено в [10] и [9].

### 3.3.2 Оптимизация полного перебора

Перед оптимизацией полного перебора стоит задача уменьшить количество итераций, требуемых для покрытия пространства возможных

инструкций. Это означает, что для каждого байта последовательности длины $n$ необходимо выделить подмножество значений $|\{x_i\}| < 2^8$. Заметим, что, согласно [9] и [10], префиксы могут принимать лишь фиксированное количество значений.

Ясно также, что не имеет смысла перебирать все возможные операнды для заданного опкода и набора префиксов. Стоит ограничить, к примеру, значения, принимаемые операндом типа целое число следующими значениями (возможно, генерируемыми случайно):

- $0 < x < 32$ (для операций сдвига);
- значение, значительно превышающее 32;
- 0;
- отрицательное значение.

Аналогично, нет нужды перебирать все значения смещения при переборе операндов типа ячейка в памяти. Достаточно ограничиться, к примеру, следующими:

- 0;
- значение, превышающее длину страницы;
- значение, лежащее в интервале от 0 до длины страницы;
- отрицания к предыдущим двум.

Наконец, то же относится и к байтам ModRM и SIB. Можно перебирать лишь некоторые комбинации этих байт, сделав при этом обход следующих аргументов:

- [значение регистра + смещение] с подстановкой двух различных регистров и для всех длин смещений;
- [значение регистра + множитель * значение регистра + смещение] с подстановкой 3-х случайных регистров в различных комбинациях в оба вхождения для всех множителей и длин смещений;
- два «значения регистра» с подстановкой 3-х случайных регистров в различных комбинациях.

Заметим, что оптимизации такого рода неприменимы к байтам, кодирующим опкод инструкции.

# 4 Результаты

В результате работы был разработан автоматизированный способ (см. 3.2.1), автоматической верификации заданного дизассемблера. В проектном дизассемблере (см. 1.1) было обнаружено и исправлено большое количество критических к работе ошибок.

# 5 Дальнейшее развитие

В дальнейшем планируется:

- провести верификацию популярных дизассемблеров с оформлением количественных результатов;
- развитие проектного дизассемблера (см. 1.1) вместе с системой верификации как отдельный проект партнерской программы с Intel [3] под названием «XXI Decoder».

# 6 Список литературы


[1] EMC Corporation. http://emc.com/.

[2] IDA Pro. http://www.hex-rays.com/products/ida/index.shtml.

[3] Intel Corporation. http://intel.com/.

[4] Objconv. http://www.agner.org/optimize/#objconv.

[5] OllyDbg. http://www.ollydbg.de/.

[6] PEBrowse Professional. http://www.smidgeonsoft.prohosting.com/pebrowse-pro-file-viewer.html.

[7] Structured Exception Handling. http://msdn.microsoft.com/en-us/library/windows/desktop/ms680657(v=vs.85).aspx.

[8] The GNU Project Debugger. http://sources.redhat.com/gdb/.

[9] AMD. *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions*.

[10] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes 2A, 2B, and 2C: Instruction Set Reference, A-Z*.

[11] Одеров Р. С, Серко С. А. Исследование и тестирование семплирующего метода профайлинга на примере профилировщика производительности Intel VTune Amplifier XE 2011. In *Сборник тезисов конференции СПИСОК-2012*. СПбГУ, 2012.

[12] Одеров Р. С., Тенсин Е. Д. Способы размещения своего кода в ядре ОС Microsoft Windows Server 2008. In *Сборник трудов межвузовской научно-практической конференции «Актуальные проблемы организации и технологии защиты информации»*. СПбГУ, 2012.

[13] Тенсин Е. Д. Верификация дизассемблера x86-64. In *Сборник тезисов конференции СПИСОК-2012*. СПбГУ, 2012.