Средства симуляции ЦП и ОС и изучение поведения программ

Лекция №10

Державин Андрей
Шурыгин Антон

- **Квиз**
- Введение
- Динамический бинарный анализ
- Valgrind
- Свой инструмент в Valgrind

Отсканируйте QR-код или
перейдите на

play.myquiz.ru

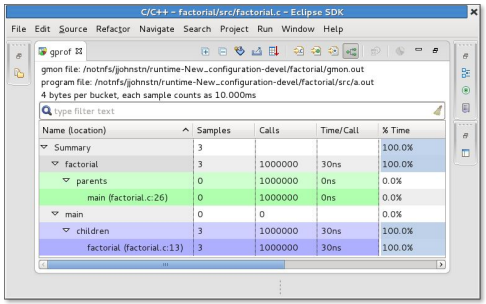
Введите код

00733693

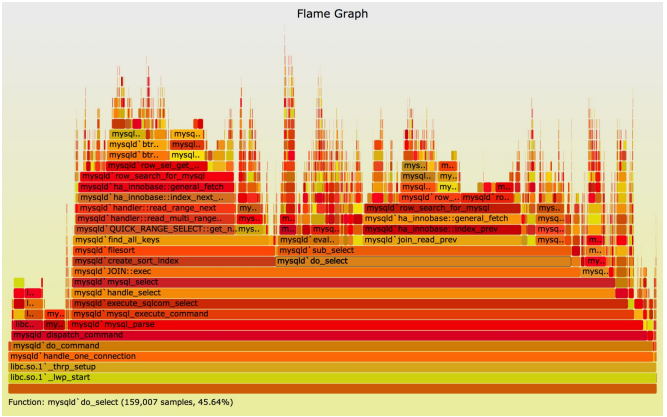
- Квиз
- **Введение**
- Динамический бинарный анализ
- Valgrind
- Свой инструмент в Valgrind

Введение

- Важно уметь производить анализ работы приложений
  - Анализ производительности
  - Низкоуровневые оптимизации



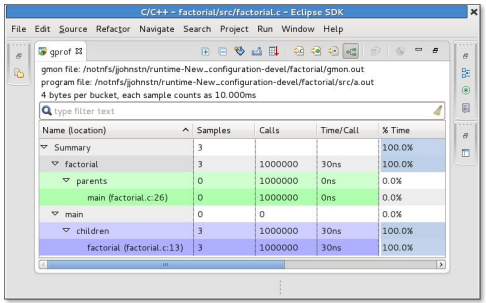
Eclipse Wiki., Eclipse C/C++ SDK, GProf profile view



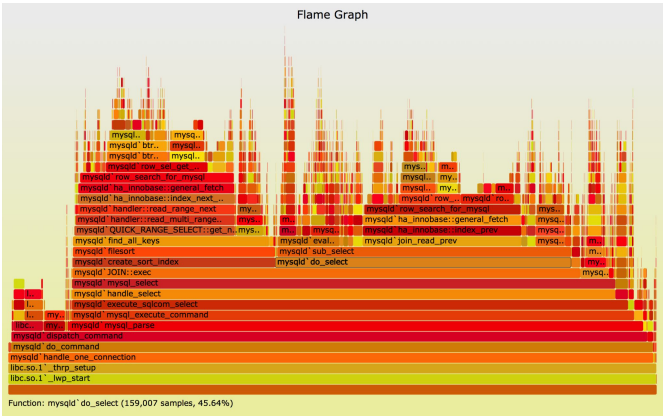
Linux perf, flame graph view

Введение

- Важно уметь производить анализ работы приложений
  - Анализ производительности
  - Низкоуровневые оптимизации
- Анализ программ можно разделить на две группы в зависимости от того, когда он проводится:
  - Статический анализ программ
  - Динамический анализ



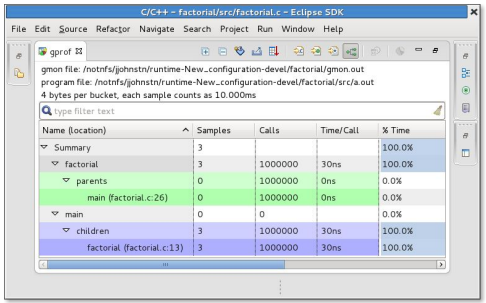
Eclipse Wiki., Eclipse C/C++ SDK, GProf profile view



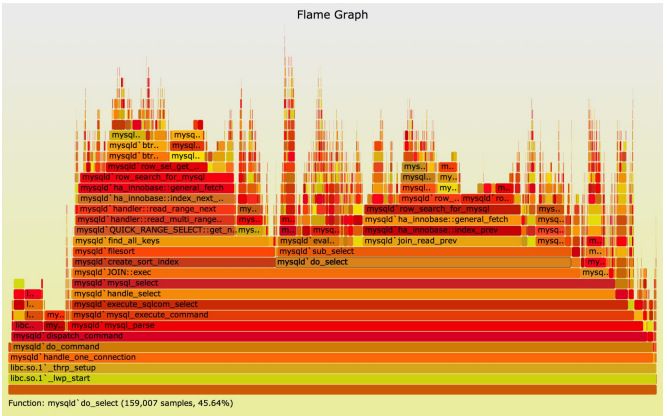
Linux perf, flame graph view

Введение

- Важно уметь производить анализ работы приложений
  - Анализ производительности
  - Низкоуровневые оптимизации
- Анализ программ можно разделить на еще две группы в зависимости от типа анализируемого кода:
  - Анализ исходного кода
  - Анализ бинарного кода



Eclipse Wiki., Eclipse C/C++ SDK, GProf profile view



Linux perf, flame graph view

Введение

- Таким образом, миру известно четыре вида анализа работы приложений:

	Static	Dynamic
Source	Static source analysis	Dynamic source analysis
Binary	Static binary analysis	Dynamic binary analysis

Введение

- Давайте попробуем подобрать примеры приложений:

	Static	Dynamic
Source	???	
Binary		

Введение

- Давайте попробуем подобрать примеры приложений:

	Static	Dynamic
Source	Compilers, Checkers	???
Binary		

Введение

- Давайте попробуем подобрать примеры приложений:

	Static	Dynamic
Source	Compilers, Checkers	GDB, Strace
Binary	???	

Введение

- Давайте попробуем подобрать примеры приложений:

	Static	Dynamic
Source	Compilers, Checkers	GDB, Strace
Binary	Disassemblers, Decompilers	???

Введение

- Давайте попробуем подобрать примеры приложений:

	Static	Dynamic
Source	Compilers, Checkers	GDB, Strace
Binary	Disassemblers, Decompilers	DynamoRIO, PIN, Valgrind

- Рассмотрим подробнее средства динамического бинарного анализа

- Квиз
- Введение
- **Динамический бинарный анализ**
- Valgrind
- Свой инструмент в Valgrind

Динамический бинарный анализ

- Идея DBA (англ. Dynamic Binary Analysis) построена на инструментировании кода приложения
- Динамическая бинарная инструментация – технология при которой анализирующий код добавляется к исходному коду клиентской программы во время исполнения

Динамический бинарный анализ

- Идея DBA (англ. Dynamic Binary Analysis) построена на инструментировании кода приложения
- Динамическая бинарная инструментация – технология при которой анализирующий код добавляется к исходному коду клиентской программы во время исполнения
  - ➕ Не требует перекомпиляции, повторной линковки клиентской программы
  - ➕ 100%-ное инструментальное покрытие кода пользовательского режима, не требуя исходников приложения
  - ➖ Накладные расходы инструментации
  - ➖ Сложность реализации

- Квиз
- Введение
- Динамический бинарный анализ
- **Valgrind**
- Свой инструмент в Valgrind

Valgrind

- Valgrind — это не только ПО для обнаружения утечек памяти и т.д..
- Строго говоря, Valgrind – среда для создания DBA инструментов
- Релиз Valgrind 1.0 состоялся в июле 2002
- В первой версии по умолчанию Valgrind проверял ошибки по памяти, а с опцией `--cachesim` вызывался инструмент Cachegrind

Valgrind

- Valgrind — это среда для создания DBA инструментов
- Знаете ли Вы, сколько различных инструментов предлагает Valgrind?

https://valgrind.org/info/tools.html

Valgrind

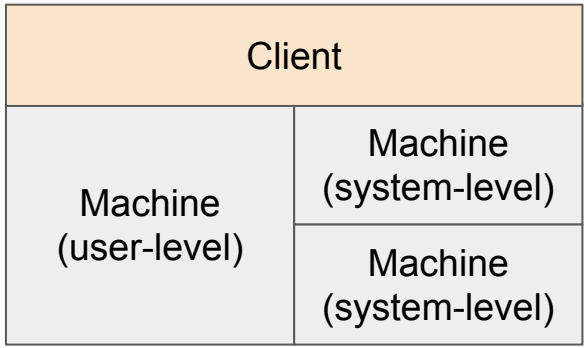
- Valgrind — это среда для создания DBA инструментов
- Знаете ли Вы, сколько различных инструментов предлагает Valgrind?
- Ответ – много, но можно выделить основные: Memcheck, Cachegrind, Callgrind, Massif, Helgrind, DRD, DHAT

https://valgrind.org/info/tools.html

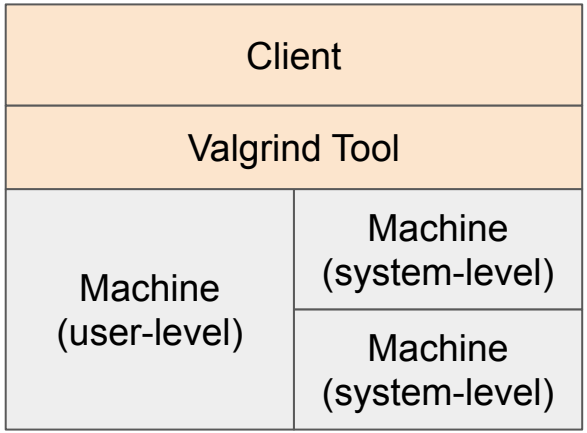
Valgrind

- DBA инструменты реализуются как плагины.
- Инструменты Valgrind работают по одному и тому же базовому принципу, хотя информация, которую они выдают, различается

Valgrind core + tool plug-in = Valgrind tool



Нормальный сценарий исполнения программы



Исполнение программы по инструментом Valgrind

Valgrind

- Как вы думаете, что здесь происходит?

```
(anton| ~/code/valgrind(git*≠master)) valgrind --tool=memcheck date
==27536== Memcheck, a memory error detector
==27536== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==27536== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==27536== Command: date
==27536==
Mon Dec  2 18:07:11 MSK 2024
==27536==
==27536== HEAP SUMMARY:
==27536==     in use at exit: 128 bytes in 1 blocks
==27536==   total heap usage: 242 allocs, 241 frees, 27,958 bytes allocated
==27536==
==27536== LEAK SUMMARY:
==27536==    definitely lost: 128 bytes in 1 blocks
==27536==    indirectly lost: 0 bytes in 0 blocks
==27536==      possibly lost: 0 bytes in 0 blocks
==27536==    still reachable: 0 bytes in 0 blocks
==27536==         suppressed: 0 bytes in 0 blocks
==27536== Rerun with --leak-check=full to see details of leaked memory
==27536==
==27536== For lists of detected and suppressed errors, rerun with: -s
==27536== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Запуск инструмента Memcheck из Valgrind

Valgrind

- Как вы думаете, что здесь происходит?
- Если очень вкратце, то примерно следующее:
  - –tool=<tool_name> запускается, загружая программу в тот же процесс
  - Инструмент перекомпилирует машинный код клиентской программы по одному блоку
  - Ядро Valgrind разбирает блок кода на промежуточнее представление (англ. англ. Intermediate Representation)
  - IR инструментируется с помощью анализирующего кода, предоставляемый модулем инструмента
  - Инструментированный IR транслируется обратно в машинный код
  - Полученная трансляция сохраняется в кэше для повторного запуска
  - Исходный код приложения не запускается

Valgrind

- А что за IR?

Valgrind

- А что за IR?
- Существует два основных способа представления кода и проведения инструментации:
  - D&R – disassemble-and-resynthesise
  - C&A – copy-and-annotate

Valgrind

- А что за IR?
- Существует два основных способа представления кода и проведения инструментации:
  - ➢ D&R – disassemble-and-resynthesise
    - Машинный код преобразуется в промежуточное представление
    - Каждая инструкция сопоставляется одной или нескольким IR операциям
    - IR инструментируется, затем обратно преобразуется в машинный код
  - ○ C&A – copy-and-annotate

Valgrind

- А что за IR?
- Существует два основных способа представления кода и проведения инструментации:
  - D&R – disassemble-and-resynthesise
  - ➢ C&A – copy-and-annotate
    - Входящие инструкции последовательно копируются
    - Каждая инструкция аннотируется с помощью внутренних структур данных (например, DynamoRIO) или API-запроса (например, PIN)

Valgrind

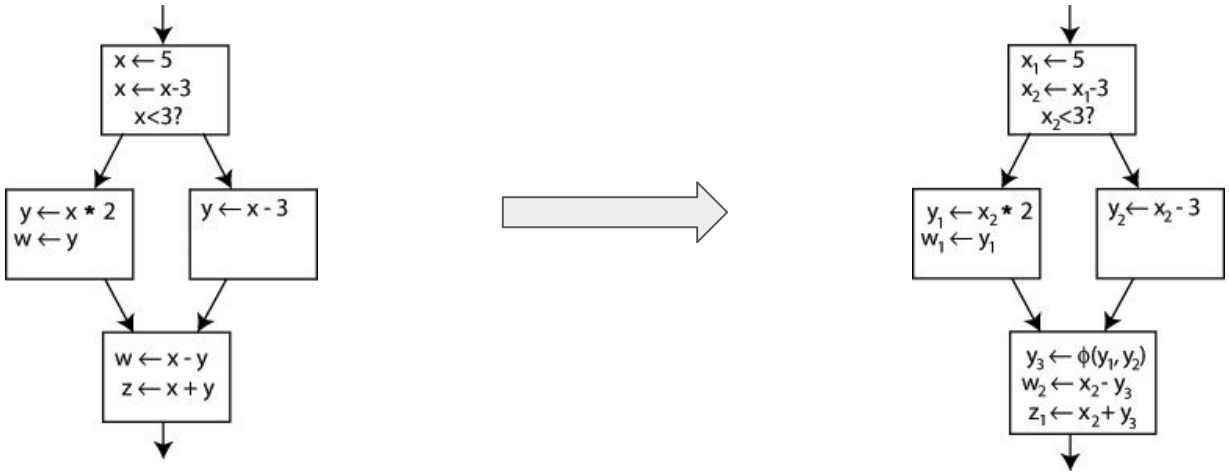
- А что за IR?
- Существует два основных способа представления кода и проведения инструментации:
  - D&R – disassemble-and-resynthesise
  - C&A – copy-and-annotate
- Ранние версии Valgrind использовали гибридный подход D&R для целочисленных инструкций, C&A для FP & SIMD инструкций.

Valgrind

- Valgrind’s IR – VEX IR
  - ➢ Архитектурно-независимый IR, D&R
    - ■ До версии 3.0.0 (август 2005 г.) Valgrind имел частично D&R, частично C&A, подобный ассемблерному x86 коду IR, в котором единицами трансляции были базовые блоки.

Valgrind

- Valgrind’s IR – VEX IR
  - Архитектурно-независимый IR, D&R
  - ➢ SSA (single static assignment) – тип промежуточного представления, в котором каждая переменная присваивается ровно один раз



Valgrind

- Valgrind’s IR – VEX IR
  - Архитектурно-независимый IR, D&R
  - SSA (single static assignment)
  - ➢ Блоки IR – суперблоки, фрагменты кода с одним входом, несколькими выходами

```
   "IRSB" stands for "IR Super Block".
*/
typedef
   struct {
      IRTypeEnv* tyenv;
      IRStmt**   stmts;
      Int        stmts_size;
      Int        stmts_used;
      IRExpr*    next;
      IRJumpKind jumpkind;
      Int        offsIP;
   }
   IRSB;
```

IR Super Block definition from valgrind/VEX/pub/libvex_ir.h

Valgrind

- Valgrind’s IR – VEX IR
  - Архитектурно-независимый IR, D&R
  - SSA (single static assignment)
  - Блоки IR
  - ➢ Каждый блок содержит список операторов, являющиеся операциями с side-effects

...

```
typedef
   struct _IRStmt {
      IRStmtTag tag;
      union {
         /* A no-op (usually resulting from IR optimisation).  Can be
            omitted without any effect.

            ppIRStmt output: IR-NoOp
         */
         struct {
      } NoOp;
```

IR Statement definition (partially) from valgrind/VEX/pub/libvex_ir.h

Valgrind

- Valgrind’s IR – VEX IR
  - Архитектурно-независимый IR, D&R
  - SSA (single static assignment)
  - Блоки IR
  - Каждый блок содержит список операторов, являющиеся операциями с side-effects
  - ➢ Операторы содержат выражения, которые представляют чистые значения без side-effects

```
struct _IRExpr {
   IRExprTag tag;
   union {
      /* Used only in pattern matching within Vex.  Should not be seen
         outside of Vex. */
      struct {
         Int binder;
      } Binder;
```

IR Expression definition (partially) from valgrind/VEX/pub/libvex_ir.h

Valgrind

- Valgrind’s IR – VEX IR
  - Архитектурно-независимый IR, D&R
  - SSA (single static assignment)
  - Блоки IR
  - Каждый блок содержит список операторов, являющиеся операциями с side-effects
  - Операторы содержат выражения, которые представляют чистые значения без side-effects
- Детальнее VEX IR можно изучить в VEX/pub/libvex_ir.h

Valgrind

- Создавая блок трансляции, Valgrind следует по инструкциям, пока не выполнено одно из условий:
  - Достигнут предел на размер блока кода (~50 инструкций, зависит от архитектуры)
  - Выполнен условный переход
  - Выполнен косвенный переход (англ. indirect branch)
  - Выполнено более трех безусловных переходов

Valgrind

- Понятие “теневых значений” (англ. shadow values)
  - Для каждого значения из регистра и памяти, исключительно программно, создается “теневая копия”
  - Memcheck использует теневые значения, чтобы отслеживать, какие биты не определены (не инициализированы или получены из неопределенных значений)
  - Данное решение позволяет ему достаточно точно обнаруживать уязвимости по памяти

Valgrind

- Понятие “теневых значений” (англ. shadow values)
- Guest & Host регистры
  - Valgrind работает на реальной машине, или же хозяйском процессоре (англ. host CPU)
  - Концептуально запуск клиентской программы происходит на эмулируемом гостевом процессоре (англ. guest CPU)
  - “Теневые” регистры – это теневые значения гостевых регистров

Valgrind

- Понятие “теневых значений” (англ. shadow values)
- Guest & Host регистры
- Valgrind предоставляет блок памяти на клиентский поток, так называемый **ThreadState**

Valgrind

- Как следствие D&R у Valgrind восемь фаз трансляции:
  1. Дизассемблирование и трансляция в неоптимизированный (tree) IR
  2. Оптимизация IR: tree IR → flat IR
  3. Инструментация IR
  4. Оптимизация инструментированного IR
  5. flat IR → tree IR
  6. Instruction Selection
  7. Register Allocation
  8. Assembly

Valgrind

- Пример IR
  - ➢ Операторы 1, 4, 14 – IMarks, это пустые операции, которые указывают на начало инструкции, ее адрес, длину в байтах

```
0x24F275:  movl -16180(%ebx,%eax,4),%eax
 1: ------ IMark(0x24F275, 7) ------
 2: t0 = Add32(Add32(GET:I32(12),# get %ebx and
      Shl32(GET:I32(0),0x2:I8)), # %eax, and
      0xFFFFC0CC:I32)            # compute addr
 3: PUT(0) = LDle:I32(t0)       # put %eax

0x24F27C:  addl %ebx,%eax
 4: ------ IMark(0x24F27C, 2) ------
 5: PUT(60) = 0x24F27C:I32      # put %eip
 6: t3 = GET:I32(0)             # get %eax
 7: t2 = GET:I32(12)            # get %ebx
 8: t1 = Add32(t3,t2)           # addl
 9: PUT(32) = 0x3:I32           # put eflags val1
10: PUT(36) = t3                # put eflags val2
11: PUT(40) = t2                # put eflags val3
12: PUT(44) = 0x0:I32           # put eflags val4
13: PUT(0) = t1                 # put %eax

0x24F27E:  jmp*l %eax
14: ------ IMark(0x24F27E, 2) ------
15: PUT(60) = 0x24F27E:I32      # put %eip
16: t4 = GET:I32(0)             # get %eax
17: goto {Boring} t4
```

Дизассемблирование: машинный код→ VEX IR

Valgrind

- Пример IR
  - Операторы 1, 4, 14 – IMarks, это пустые операции, которые указывают на начало инструкции, ее адрес, длину в байтах
  - ➢ Оператор 2 присваивает временному t0 дерево выражений IR операций, соответствующие одной CISC инструкции

```
0x24F275:  movl -16180(%ebx,%eax,4),%eax
 1: ------ IMark(0x24F275, 7) ------
 2: t0 = Add32(Add32(GET:I32(12),# get %ebx and
      Shl32(GET:I32(0),0x2:I8)), # %eax, and
      0xFFFFC0CC:I32)            # compute addr
 3: PUT(0) = LDle:I32(t0)        # put %eax

0x24F27C:  addl %ebx,%eax
 4: ------ IMark(0x24F27C, 2) ------
 5: PUT(60) = 0x24F27C:I32       # put %eip
 6: t3 = GET:I32(0)              # get %eax
 7: t2 = GET:I32(12)             # get %ebx
 8: t1 = Add32(t3,t2)            # addl
 9: PUT(32) = 0x3:I32            # put eflags val1
10: PUT(36) = t3                 # put eflags val2
11: PUT(40) = t2                 # put eflags val3
12: PUT(44) = 0x0:I32            # put eflags val4
13: PUT(0) = t1                  # put %eax

0x24F27E:  jmp*l %eax
14: ------ IMark(0x24F27E, 2) ------
15: PUT(60) = 0x24F27E:I32       # put %eip
16: t4 = GET:I32(0)              # get %eax
17: goto {Boring} t4
```

Дизассемблирование: машинный код→ VEX IR

Valgrind

- Пример IR
  - Операторы 1, 4, 14 – IMarks, это пустые операции, которые указывают на начало инструкции, ее адрес, длину в байтах
  - Оператор 2 присваивает временному t0 дерево выражений IR операций, соответствующие одной CISC инструкции
  - ➢ Оператор 3 – запись гостевого регистра обратно в его слот ThreadState

```
0x24F275:  movl -16180(%ebx,%eax,4),%eax
 1: ------ IMark(0x24F275, 7) ------
 2: t0 = Add32(Add32(GET:I32(12),# get %ebx and
      Shl32(GET:I32(0),0x2:I8)), # %eax, and
      0xFFFFC0CC:I32)            # compute addr
 3: PUT(0) = LDle:I32(t0)       # put %eax

0x24F27C:  addl %ebx,%eax
 4: ------ IMark(0x24F27C, 2) ------
 5: PUT(60) = 0x24F27C:I32      # put %eip
 6: t3 = GET:I32(0)             # get %eax
 7: t2 = GET:I32(12)            # get %ebx
 8: t1 = Add32(t3,t2)           # addl
 9: PUT(32) = 0x3:I32           # put eflags val1
10: PUT(36) = t3                # put eflags val2
11: PUT(40) = t2                # put eflags val3
12: PUT(44) = 0x0:I32           # put eflags val4
13: PUT(0) = t1                 # put %eax

0x24F27E:  jmp*l %eax
14: ------ IMark(0x24F27E, 2) ------
15: PUT(60) = 0x24F27E:I32      # put %eip
16: t4 = GET:I32(0)             # get %eax
17: goto {Boring} t4
```

Дизассемблирование: машинный код→ VEX IR

Valgrind

- Пример IR
  - Операторы 1, 4, 14 – IMarks, это пустые операции, которые указывают на начало инструкции, ее адрес, длину в байтах
  - Оператор 2 присваивает временному t0 дерево выражений IR операций, соответствующие одной CISC инструкции
  - Оператор 3 – запись гостевого регистра обратно в его слот ThreadState
  - ➢ Операторы 9-12 – запись четырех значений в ThreadState (condition codes)

```
0x24F275:  movl -16180(%ebx,%eax,4),%eax
 1: ------ IMark(0x24F275, 7) ------
 2: t0 = Add32(Add32(GET:I32(12),# get %ebx and
      Shl32(GET:I32(0),0x2:I8)), # %eax, and
      0xFFFFC0CC:I32)            # compute addr
 3: PUT(0) = LDle:I32(t0)       # put %eax

0x24F27C:  addl %ebx,%eax
 4: ------ IMark(0x24F27C, 2) ------
 5: PUT(60) = 0x24F27C:I32      # put %eip
 6: t3 = GET:I32(0)             # get %eax
 7: t2 = GET:I32(12)            # get %ebx
 8: t1 = Add32(t3,t2)           # addl
 9: PUT(32) = 0x3:I32           # put eflags val1
10: PUT(36) = t3                # put eflags val2
11: PUT(40) = t2                # put eflags val3
12: PUT(44) = 0x0:I32           # put eflags val4
13: PUT(0) = t1                 # put %eax

0x24F27E:  jmp*l %eax
14: ------ IMark(0x24F27E, 2) ------
15: PUT(60) = 0x24F27E:I32      # put %eip
16: t4 = GET:I32(0)             # get %eax
17: goto {Boring} t4
```

Дизассемблирование: машинный код→ VEX IR

Valgrind

- Пример IR
  - Операторы 1, 4, 14 – IMarks, это пустые операции, которые указывают на начало инструкции, ее адрес, длину в байтах
  - Оператор 2 присваивает временному t0 дерево выражений IR операций, соответствующие одной CISC инструкции
  - Оператор 3 – запись гостевого регистра обратно в его слот ThreadState
  - Операторы 9-12 – запись четырех значений в ThreadState (condition codes)
  - ➢ Оператор 17 – безусловный переход к адресу в t4.

```
0x24F275:  movl -16180(%ebx,%eax,4),%eax
 1: ------ IMark(0x24F275, 7) ------
 2: t0 = Add32(Add32(GET:I32(12),# get %ebx and
      Shl32(GET:I32(0),0x2:I8)), # %eax, and
      0xFFFFC0CC:I32)            # compute addr
 3: PUT(0) = LDle:I32(t0)        # put %eax

0x24F27C:  addl %ebx,%eax
 4: ------ IMark(0x24F27C, 2) ------
 5: PUT(60) = 0x24F27C:I32       # put %eip
 6: t3 = GET:I32(0)              # get %eax
 7: t2 = GET:I32(12)             # get %ebx
 8: t1 = Add32(t3,t2)            # addl
 9: PUT(32) = 0x3:I32            # put eflags val1
10: PUT(36) = t3                 # put eflags val2
11: PUT(40) = t2                 # put eflags val3
12: PUT(44) = 0x0:I32            # put eflags val4
13: PUT(0) = t1                  # put %eax

0x24F27E:  jmp*l %eax
14: ------ IMark(0x24F27E, 2) ------
15: PUT(60) = 0x24F27E:I32       # put %eip
16: t4 = GET:I32(0)              # get %eax
17: goto {Boring} t4
```

Дизассемблирование: машинный код→ VEX IR

Valgrind

- Оптимизация и инструментация VEX IR

```
0x24F275:  movl -16180(%ebx,%eax,4),%eax
 1: ------ IMark(0x24F275, 7) ------
 2: t0 = Add32(Add32(GET:I32(12),# get %ebx and
      Shl32(GET:I32(0),0x2:I8)), # %eax, and
      0xFFFFC0CC:I32)            # compute addr
 3: PUT(0) = LDle:I32(t0)        # put %eax

0x24F27C:  addl %ebx,%eax
 4: ------ IMark(0x24F27C, 2) ------
 5: PUT(60) = 0x24F27C:I32       # put %eip
 6: t3 = GET:I32(0)              # get %eax
 7: t2 = GET:I32(12)             # get %ebx
 8: t1 = Add32(t3,t2)            # addl
 9: PUT(32) = 0x3:I32            # put eflags val1
10: PUT(36) = t3                 # put eflags val2
11: PUT(40) = t2                 # put eflags val3
12: PUT(44) = 0x0:I32            # put eflags val4
13: PUT(0) = t1                  # put %eax

0x24F27E:  jmp*l %eax
14: ------ IMark(0x24F27E, 2) ------
15: PUT(60) = 0x24F27E:I32       # put %eip
16: t4 = GET:I32(0)              # get %eax
17: goto {Boring} t4
```

Дизассемблирование: машинный код→ VEX IR

```
* 1: ------ IMark(0x24F275, 7) ------
  2: t11 = GET:I32(320)          #  get sh(%eax)
* 3: t8  = GET:I32(0)            # *get %eax
  4: t14 = Shl32(t11,0x2:I8)     #  shadow shll
* 5: t7  = Shl32(t8,0x2:I8)      # *shll
  6: t18 = GET:I32(332)          #  get sh(%ebx)
* 7: t9  = GET:I32(12)           # *get %ebx
  8: t19 = Or32(t18,t14)         #  shadow addl 1/3
  9: t20 = Neg32(t19)            #  shadow addl 2/3
 10: t21 = Or32(t19,t20)         #  shadow addl 3/3
*11: t6  = Add32(t9,t7)          # *addl
 12: t24 = Neg32(t21)            #  shadow addl 1/2
 13: t25 = Or32(t21,t24)         #  shadow addl 2/2
*14: t5  = Add32(t6,0xFFFFC0CC:I32) # *addl
 15: t27 = CmpNEZ32(t25)         #  shadow loadl 1/3
 16: DIRTY t27 RdFX-gst(16,4) RdFX-gst(60,4)
       ::: helperc_value_check4_fail{0x380035f4}()
                                 #  shadow loadl 2/3
 17: t29 = DIRTY 1:I1 RdFX-gst(16,4) RdFX-gst(60,4)
       ::: helperc_LOADV32le{0x38006504}(t5)
                                 #  shadow loadl 3/3
*18: t10 = LDle:I32(t5)          # *loadl
```

Оптимизированный и инструментированный VEX IR

- Квиз
- Введение
- Динамический бинарный анализ
- Valgrind
- ➢ **Свой инструмент в Valgrind**

Свой инструмент в Valgrind

1. Руководство к действию можно найти прямо в документации Valgrind – **Writing a New Valgrind Tool**

Давайте напишем в Valgrind инструмент, который будет считать число исполненных инструкций.

Свой инструмент в Valgrind

1. Руководство к действию можно найти прямо в документации Valgrind – **Writing a New Valgrind Tool**
2. Вкратце разберём мануал из документации:
   a. Выбираем имя и аббревиатуру новому инструменту. Например **icounter**, **IC**.
   b. Из корня проекта valgrind создаем три директории `icounter/`, `icounter/tests`, `icounter/docs`
   c. В тестовой директории создаем пустой файл `icounter/tests/Makefile.am`

Свой инструмент в Valgrind

1. Руководство к действию можно найти прямо в документации Valgrind – **Writing a New Valgrind Tool**
2. Вкратце разберём мануал из документации:
   a. Выбираем имя и аббревиатуру новому инструменту. Например **icounter**, **IC**.
   b. Из корня проекта valgrind создаем три директории `icounter/`, `icounter/tests`, `icounter/docs`
   c. В тестовой директории создаем пустой файл `icounter/tests/Makefile.am`
3. В проекте Valgrind есть специальный демонстрационный инструмент, Nulgrind, в директории valgrind/none
   a. Из `valgrind/none` копируем файлы `valgrind/none/nl_main.c` и `valgrind/none/Makefile.am`, заменив все вхождения none на icounter, nl_ и nl- на ic_ и ic- соответственно

Свой инструмент в Valgrind

4. Добавляем директорию `icounter/` в сборку, изменив переменную TOOLS в корневом `valgrind/Makefile.am`
5. В `valgrind/configure.ac` добавляем в лист AC_CONFIG_FILES `icounter/Makefile`, `icounter/tests/Makefile`
6. Запускаем скрипты:

```
bash autogen.sh
./configure --prefix=$PWD/inst
make
make install -j <nproc>
```

Конфигурация и запуск сборки
проекта Valgrind

Свой инструмент в Valgrind

- Новый инструмент должен определять хотя бы четыре следующие функции:
  - `pre_clo_init()` – функция, отвечающая за инициализацию до обработки параметров из CLI
  - `post_clo_init()` – функция, отвечающая за инициализацию после обработки параметров из CLI
  - `instrument()` – функция, отвечающая за инструментацию VEX IR
  - `fini()` – функция, отвечающая за вывод результатов DBA

Свой инструмент в Valgrind

- Возвращаемся к реализации Valgrind/Icounter:
  - Логику анализа приложения реализуем в функции инструментации, `instrument()`

Свой инструмент в Valgrind

- Возвращаемся к реализации Valgrind/Icounter:
  - Логику анализа приложения реализуем в функции инструментации, `instrument()`
  - Алгоритм работы инструмента – линейный проход по всем операторам IR блока с последующей проверкой (см. `IRStmtTag`)
  - Если встречаем `Ist_IMark`, инкрементируем счетчик инструкций

Свой инструмент в Valgrind

- Возвращаемся к реализации Valgrind/Icounter:
  - Логику анализа приложения реализуем в функции инструментации, `instrument()`
  - Алгоритм работы инструмента – линейный проход по всем операторам IR блока с последующей проверкой (см. `IRStmtTag`)
  - Если встречаем `Ist_IMark`, инкрементируем счетчик инструкций

```
static IRSB* ic_instrument(VgCallbackClosure*     closure,
                           IRSB*                  bb,
                           const VexGuestLayout*  layout,
                           const VexGuestExtents* vge,
                           const VexArchInfo*     archinfo_host,
                           IRType                 gWordTy,
                           IRType                 hWordTy)
{
   IRSB* out_bb = deepCopyIRSBExceptStmts(bb);

   for (Int i = 0; i < bb->stmts_used; i++) {
      IRStmt* st = bb->stmts[i];
      switch (st->tag) {
      case Ist_IMark: {
         IRDirty* di = unsafeIRDirty_0_N(0, "incr_instr_num",
                                         VG_(fnptr_to_fnentry)(&incr_instr_num),
                                         mkIRExprVec_0());
         addStmtToIRSB(out_bb, IRStmt_Dirty(di));
      }
      case Ist_NoOp:
      case Ist_AbiHint:
      case Ist_Put:
      case Ist_PutI:
      case Ist_MBE:
      case Ist_WrTmp:
      case Ist_Store:
      case Ist_StoreG:
      case Ist_LoadG:
      case Ist_Dirty:
      case Ist_CAS:
      case Ist_LLSC:
      case Ist_Exit:
         addStmtToIRSB(out_bb, st);
         break;
      default: {
         ppIRStmt(st);
      }
      }
   }
   return out_bb;
}
```

Пример реализации подсчета числа инструкций

Свой инструмент в Valgrind

- Давайте протестируем?
- Корректность работы инструмента Icounter можно сверить, например, с Cachegrind:

```
(anton| ~/code/valgrind(git*≠master)) ./inst/bin/valgrind --tool=cachegrind date
==31568== Cachegrind, a high-precision tracing profiler
==31568== Copyright (C) 2002-2024, and GNU GPL'd, by Nicholas Nethercote et al.
==31568== Using Valgrind-3.25.0.GIT and LibVEX; rerun with -h for copyright info
==31568== Command: date
==31568==
Thu Dec  5 04:07:25 MSK 2024
==31568==
==31568== I refs:        396,878
```

```
(anton| ~/code/valgrind(git*≠master)) ./inst/bin/valgrind --tool=icounter date
==31788== Instruction counter Tool, Instruction counter Tool Valgrind tool
==31788== Copyright (C) 2002-2024, and GNU GPL'd, by Nicholas Nethercote et al.
==31788== Using Valgrind-3.25.0.GIT and LibVEX; rerun with -h for copyright info
==31788== Command: date
==31788==
Thu Dec  5 04:07:45 MSK 2024
==31788==
==31788==   Instructions number:  396,878
```

Сравнение результатов работы инструментов Icounter и Cachegrind
(выводит число инструкций по умолчанию)

Valgrind

- Несмотря на почтенный возраст Valgrind все еще актуален
- Применение Valgrind можно найти и в образовательных целях, и при разработке на относительно современных языках программирования.
- Например, для языка Rust (казалось бы?), в случае если вы:
  - Используете ключевое слово `unsafe` при разработке программ
  - Используете профилирующие инструменты из пакета Valgrind

Литература

1. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation
2. How to Shadow Every Byte of Memory Used by a Program
3. Dynamic Binary Analysis and Instrumentation
4. Rust and Valgrind
5. Twenty years of Valgrind