Анализ кода и информационная безопасность

Лекция 06

Отладка и инструментирование

МГУ / ВМК / СП

0601

ОТЛАДКА ПОЛЬЗОВАТЕЛЬСКОГО КОДА

Тестирование и отладка

- Тестирование и отладка — динамические подходы к анализу программ.
- Тестирование отвечает на вопрос «что необходимо сделать, чтобы сломать программу».
- Отладка отвечает на вопрос «что необходимо сделать, чтобы починить программу».

«Борбаги» и «гейзенбаги»

«Борбаг» (“bohr bug”) — ошибка, которая не исчезает и не меняет своих свойств при попытке её обнаружения.

«Гейзенбаг» (“heisenbug”) — ошибка, которая исчезает или меняет свои свойства при попытке её обнаружения.

Распространённые причины «гейзенбагов»:

- использование неинициализированной переменной;
- состояние гонки.

Отладка пользовательского кода

- “Print-debugging” — добавление в программу вывода значений переменных в окрестности ошибки.
- Бисекция (“wolf fence” algorithm) — способ локализации ошибки:
  - временное отключение частей кода с поиском того момента, когда ошибка перестаёт воспроизводиться;
  - либо, наоборот, последовательное включение частей кода с поиском того момента, когда ошибка начнёт воспроизводиться.
- Применение отладчика:
  - локальная отладка (на той же самой машине);
  - удалённая отладка (через сетевое соединение).
- Отладчик подключается к выполняющемуся процессу либо к дампу памяти (core dump), полученному при аварийном завершении программы.

Отладочная информация

- Компилируемая программа отлаживается на уровне машинных команд (или команд виртуальной машины соответствующего языка, например, Java VM).
- Требуется сопоставление объектов низкого уровня (адресов, неструктурированных данных) с объектами высокого уровня (функциями, строками кода, типизированными данными).
- Компилятор может выдавать отладочную информацию, описывающую соответствие между сгенерированным бинарным кодом и исходным кодом, из которого он был получен.
- Форматы отладочной информации:
  - DWARF — отладочная информация в секциях ELF-файлов;
  - Microsoft PDB — отладочная информация в отдельном PDB-файле, поддерживает загрузку по сети («серверы символов»).

DWARF

- Типы данных, используемые в программе (поддержка основных компилируемых языков программирования — Си, Си++, Fortran, Ada и родственных).
- Точки входа — адреса начал функций, их имена и сигнатуры.
- Отображение адресов в бинарном коде на имена файлов и номера строк исходного кода.
- Описание переменных: имена, типы, расположение. Поскольку расположение переменных в общем случае не является фиксированным во время выполнения программы (переменные на стеке, области с относительной адресацией), DWARF включает язык выражений, позволяющий описывать способ вычисления адреса объекта в общем виде.

Базовый функционал отладчика

Точка останова (breakpoint) — адрес команды, при достижении которого выполнение программы должно быть приостановлено.

Точка отслеживания (watchpoint) — адрес ячейки памяти, при обращении к которой на чтение или запись выполнение программы должно быть приостановлено.

Пошаговое выполнение программы — выполнение одной машинной команды или одной строки исходного кода (с использованием отладочной информации).

В момент, когда выполнение программы приостановлено, отладчик предоставляет возможность просмотра содержимого регистров и памяти.

Аппаратная поддержка отладки

Точки останова

Точки останова реализуются через внедрение команды INT 3 в тело программы по требуемому адресу.

- Команда INT 3 кодируется единственным байтом 0xCC.
- Отладчик сохраняет содержимое первого байта по адресу точки останова и заменяет его в загруженном в память образе программы на команду INT 3.
- При достижении потоком управления команды INT 3 процессор генерирует отладочное прерывание, которое при поддержке со стороны операционной системы перехватывается отладчиком.
- Операционная система приостанавливает выполнение потока, в котором возникло прерывание, до дальнейших указаний от отладчика.

Аппаратная поддержка отладки

Пошаговое выполнение

Установка бита TF в регистре RFLAGS приводит к тому, что после выполнения каждой очередной команды генерируется отладочное исключение.

Таким образом может быть обеспечен пошаговый просмотр состояния программы на границах команд.

Аппаратная поддержка отладки

Точки отслеживания

Архитектура x86 (и x86-64) включает отладочные регистры DR0..DR7, позволяющие, в том числе, задать четыре точки отслеживания.

Для каждой точки отслеживания указывается виртуальный адрес, размер (1, 2, 4 или 8 байтов) и один из четырёх режимов:

- отслеживать только выборку команд;
- отслеживать только обращения на запись;
- отслеживать любые обращения;
- отслеживать обращения на чтение и запись данных, но не выборку команд.

Таким образом, этот механизм позволяет также задавать и точки останова без модификации образа программы в памяти.

Обратимая отладка

Обратимая отладка (reversible debugging) — технология, позволяющая отладчику двигаться не только от текущего состояния программы к последующим («вперёд во времени»), но и в обратную сторону («назад во времени»).

Когда может быть полезна обратимая отладка?

Реализация обратимой отладки предполагает сохранение отладчиком достаточного количества информации для обращения выполнения команд.

Какая информация должна быть сохранена?

Обратимая отладка в GDB

GDB начиная с версии 7.0 (2009 г.) поддерживает обратимую отладку, добавляя новые команды.

Команды обратимой отладки:

- reverse-continue — выполнение программы назад до достижения состояния останова (breakpoint, watchpoint или точки входа в программу);
- reverse-finish — выполнение программы назад до входа в текущий фрейм;
- reverse-next, reverse-nexti — шаг назад на одну строку;
- reverse-step, reverse-stepi — шаг назад на одну машинную команду.

Антиотладочные приёмы

- Обнаружение точек останова и слежения:
  - сканирование образа программы во время выполнения: поиск байтов 0xCC, вычисление контрольных сумм;
  - запрос информации об аппаратных точках останова и слежения.
- Обнаружение наличия отладчика через API-вызовы и флаги в управляющих структурах операционной системы.
- Приёмы, основанные на замерах времени.
- «Самоотладка».
- Эксплуатация известных уязвимостей в распространённых отладчиках.
- Обфускация кода.
- Выполнение кода в виртуальной машине.
- Как бороться с антиотладочными приёмами?

0602

ПОЛНОСИСТЕМНАЯ ОТЛАДКА

Полносистемная отладка

- Локальная отладка с помощью ядерного отладчика:
  - Windows — KD, WinDBG.
- Удалённая отладка с подключением по сети:
  - требуется явная поддержка со стороны отлаживаемой операционной системы.
- Удалённая отладка с подключением через аппаратный отладочный интерфейс (ICE, JTAG):
  - требуется отдельная машина с интерфейсом;
  - требуется дополнительная аппаратура для подключения к интерфейсу отладки.
- Удалённая отладка в виртуальной машине.

Отладка в виртуальной машине

- Отладка может выполняться в полносистемном эмуляторе (виртуальной машине), например, в QEMU.
- В этом случае отладчик (GDB) подключается не к выполняющемуся на той же машине процессу, а через сетевое соединение к ответной части (GDB debugging stub) в эмуляторе.

Основное преимущество отладки в виртуальной машине — исключение влияния наличия отладчика на поведение отлаживаемой системы.

Какое исключение относится к вышеуказанному преимуществу?

Основные недостатки — относительная сложность подготовки окружения и семантический разрыв.

Семантический разрыв

- Полносистемная отладка предоставляет полную информацию о состоянии машины.
- Имеется возможность отладки драйверов и ядра ОС: можно обращаться к структурам данных, которые недоступны обычным отладчикам.
- Вместе с тем, отладка осуществляется на низком уровне (значения регистров, неинтерпретируемое содержимое памяти и т.д.).
- Существует семантический разрыв — состояние системы как его «понимает» отладчик и как его понимает оператор, находятся на разных уровнях.
- Пример: получение списка работающих потоков выполнения.
- Пример: получение информации о загруженных в текущий процесс динамических библиотеках.

Сужение семантического разрыва

- Специальная поддержка гостевого окружения в полносистемном отладчике позволяет упростить отладку.
- Например, отладчик WinDbg позволяет получить информацию о состоянии ОС в понятном оператору виде.

```
0:000>0: kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS ffffe000002287c0
    SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase: 001aa000  ObjectTable: ffffc00000003000
    Image: System

PROCESS ffffe00001e5a900
    SessionId: none  Cid: 0124    Peb: 7ff7809df000  ParentCid: 0004
    DirBase: 100595000  ObjectTable: ffffc000002c5680
    Image: smss.exe
```

Как это реализовано?

Детерминированное воспроизведение

- Некоторые ошибки, связанные с временными характеристиками выполения, перестают воспроизводиться при отладке программы в виртуальной машине.
- Пример: состояние гонки.
- Пример: взаимодействие с удалённым агентом по сети.
- Решение — детерминированное воспроизведение:
  - первый проход (record) — сбор журнала недетерминированных событий;
  - второй проход (replay) — воспроизведение журнала событий с подключением отладчика.
- Реализовано в основной ветви эмулятора QEMU.

0603

ИНСТРУМЕНТИРОВАНИЕ КОДА

Статическое инструментирование

Инструментирование — добавление в существующую программу блоков кода, осуществляющих отслеживание каких-либо её характеристик во время выполнения.

Статическое инструментирование выполняется однократно перед запуском программы.

Статическое инструментирование может выполняться:

- на уровне исходного кода — вручную или автоматически;
- на уровне бинарного кода — вручную или автоматически.

“Print-debugging” — пример выполненного вручную статического инструментирования на уровне исходного кода с целью отладки программы.

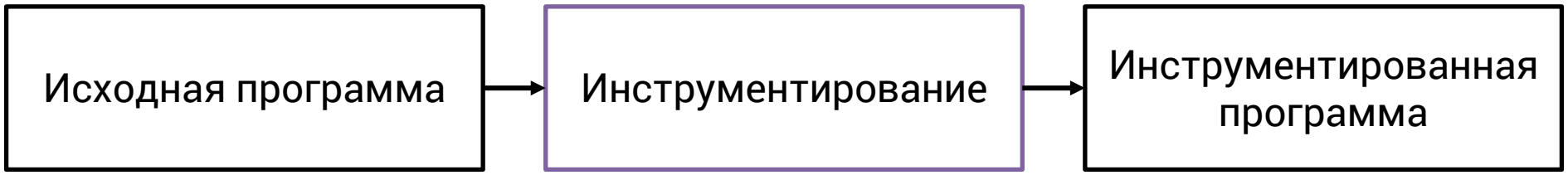
Цели инструментирования

- Трассировка выполнения программы:
  - на уровне функций;
  - на уровне базовых блоков;
  - на уровне отдельных операторов или команд.
- Профилирование — выявление наиболее «горячих» и «холодных»:
  - функций;
  - базовых блоков;
  - отдельных операторов или команд.
- Отладка:
  - некорректная работа с указателями;
  - состояния гонки (race conditions).
- Анализ и оптимизация обращений к кеш-памяти.

Статическое инструментирование

Бинарный код

- Требуется получить исчерпывающий набор базовых блоков, т.е. решить задачу дизассемблирования программы.
- Подзадачей в дизассемблировании является разделение кода и данных программы:
  - код — это те и только те байты образа программы, которых может достичь управление;
  - определение кода сводится к проблеме останова, т.е. является алгоритмически неразрешимой задачей.
- Статическое инструментирование нужно выполнять отдельно для самой программы, и отдельно — для её библиотек.



DBI

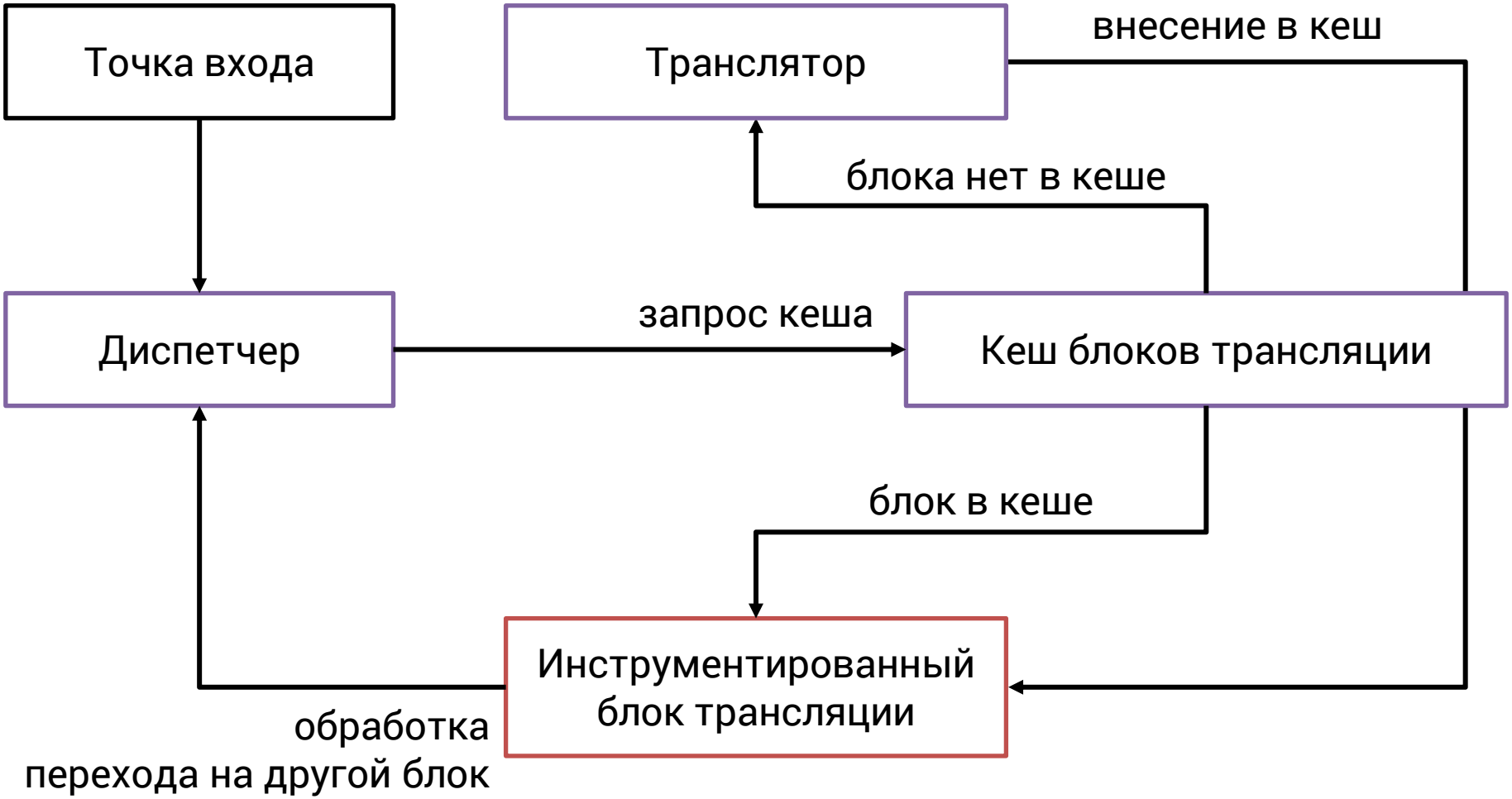
Динамическое инструментирование бинарного кода (DBI) — разновидность инструментирования бинарного кода, выполняемого во время работы программы.

Динамическое инструментирование бинарного кода родственно JIT-компиляции, т.к. предполагает построение выполняющегося кода «на лету».

Подходы к выполнению инструментирования блока трансляции:

- копирование и аннотирование (C&A — copy-and-annotate);
- «дизассемблирование» и ресинтезирование (D&R: disassemble-and-resynthesize).

DBI: общая схема работы



DBI: оптимизации

- Блок трансляции — не обязательно базовый блок, реальные DBI-системы используют более крупные единицы с несколькими выходами (расширенные базовые блоки, суперблоки).
- Сцепление (chaining, linking) блоков трансляции — если целевой адрес перехода постоянен и содержится в кеше трансляции, управление соответствующий инструментированный блок передаётся в обход диспетчера.

DBI: C&A

Добавление новых машинных команд,
передающих управление
на функции-анализаторы.

Pin

C&A DBI-система Pin — https://software.intel.com/en-us/articles/pintool.

- Разработан и поддерживается корпорацией Intel.
- Система бесплатна для некоммерческого использования.
- ISA: x86, x86-64, Intel MIC.
- Пользовательские программы для Android, Linux, Windows и macOS.

Pin: пример инструмента 1

```
FILE *trace;

VOID RecordIp(VOID *ip) {
    fprintf(trace, "%p\n", ip);
}

VOID Instruction(INS ins, VOID *v) {
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) RecordIp,
            IARG_INST_PTR, IARG_END);
}

int main(int argc, char *argv[]) {
    PIN_Init(argc, argv);
    trace = fopen("itrace.out", "w");
    INS_AddInstrumentFunction(Instruction, 0);
    PIN_StartProgram();
    return 0;
}
```

Pin: пример инструмента 2

```
FILE *trace;

VOID RecordMemWrite(VOID *ip, VOID *addr, UINT32 size) {
    fprintf(trace, "%p: W %p %d\n", ip, addr, size);
}

VOID Instruction(INS ins, VOID *v) {
    if (INS_IsMemoryWrite(ins))
        INS_InsertPredicatedCall(ins,
            IPOINT_BEFORE, (AFUNPTR) RecordMemWrite,
            IARG_INST_PTR, IARG_MEMORYWRITE_EA,
            IARG_MEMORYWRITE_SIZE, IARG_END);
}

int main(int argc, char *argv[]) {
    PIN_Init(argc, argv);
    trace = fopen("atrace.out", "w");
    INS_AddInstrumentFunction(Instruction, 0);
    PIN_StartProgram();
    return 0;
}
```

Pin: обрабатываемые события

- Помимо обработки выполнения отдельных инструкций, Pin позволяет работать с другими событиями.
- События загрузки и выгрузки динамических библиотек.
- События вызова функций.
- События входа в системные вызовы и выхода из системных вызовов.

- Поддерживается также подмена одних функций другими (например, для перехвата всех выделений и освобождений памяти, работы с файлами, сетевыми соединениями и т.п.).

DBI: D&R

Valgrind

D&R DBI-система Valgrind — http://www.valgrind.org/.

- Open source, GNU GPL 2.
- ISA: x86, x86-64, ARM, ARMv8, PPC32, PPC64, S390X, MIPS32, MIPS64, TILEGX.
- Пользовательские программы для Linux, Solaris, Android, macOS.

- IR: RISC-образная SSA-форма с явными загрузками и выгрузками памяти и регистров гостевого состояния.
- Для каждого потока выполнения (нити) поддерживается состояние гостевых регистров в отдельной области памяти.
- Теневая память и теневые регистры: аннотации для каждой ячейки памяти и гостевого регистра.

Valgrind: инструменты (tools)

- Memcheck — поиск некорректных обращений к памяти:
  - обращения по неверным адресам;
  - использование неинициализированных значений;
  - утечки памяти;
  - двойные освобождения, освобождения по неверному адресу;
  - неверное использование функций работы с памятью (например, пересекающиеся блоки памяти в memcpy).
- Cachegrind — профилирование обращений к CPU-кешу.
- Callgrind — надстройка над Cachegrind, позволяющая также строить графы вызовов.
- Massif — профилирование использования динамической памяти.
- Helgrind — поиск состояний гонки (data races).

Valgrind: этапы трансляции
1/2

1. «Дизассемблирование» — машинный код переводится в tree IR.
2. Оптимизация 1 — tree IR переводится в flat IR и оптимизируется:
   - удаление избыточных обращений к гостевому состоянию;
   - продвижение констант и копий;
   - свёртка констант;
   - удаление мёртвого кода;
   - удаление общих подвыражений;
   - частичное разворачивание циклов.
3. Инструментирование — трансформация flat IR конкретным инструментом Valgrind.
4. Оптимизация 2 — инструментированный flat IR оптимизируется вновь: выполняется свёртка констант и удаление мёртвого кода.

Valgrind: этапы трансляции
2/2

5. Построение дерева — flat IR переводится в tree IR.
6. Планирование команд — покрытие дерева tree IR машинными командами с виртуальными регистрами.
7. Распределение регистров — замена виртуальных регистров на физические, по необходимости вносятся операции сохранения и загрузки регистров из временной области (register spills).
8. Ассемблирование — перевод последовательности машинных команд с физическими регистрами в бинарный код.

Valgrind: пример трансляции

Tree IR

```
0024F275 MOV EAX, DWORD [EBX + EAX * 4 - 0x16180]
      1: ------ IMark(0x0024F275, 7) ------
      2: t0 = Add32(Add32(GET:I32(12), Shl32(GET:I32(0), 0x02:I8)), 0xFFFFC0CC:I32)
      3: PUT(0) = LDle:I32(t0)
0024F27C ADD EAX, EBX
      4: ------ IMark(0x0024F27C, 2) ------
      5: PUT(60) = 0x0024F27C:I32
      6: t3 = GET:I32(0)
      7: t2 = GET:I32(12)
      8: t1 = Add32(t3, t2)
      9: PUT(32) = 0x00000003:I32
     10: PUT(36) = t3
     11: PUT(40) = t2
     12: PUT(44) = 0x00000000:I32
     13: PUT(0) = t1
0024F27E JMP EAX
     14: ------ IMark(0x0024F27E, 2) ------
     15: PUT(60) = 0x0024F27E:I32
     16: t4 = GET:I32(0)
     17: goto {Boring} t4
```

Valgrind: пример трансляции

Flat IR, инструментированный Memcheck

```
 1: ------ IMark(0x0024F275, 7) ------
 2: t11 = GET:I32(320)
 3: t8 = GET:I32(0)
 4: t14 = Shl32(t11, 0x02:I8)
 5: t7 = Shl32(t8, 0x02:I8)
 6: t18 = GET:I32(332)
 7: t9 = GET:I32(12)
 8: t19 = Or32(t18, t14)
 9: t20 = Neg32(t19)
10: t21 = Or32(t19, t20)
11: t6 = Add32(t9, t7)
12: t24 = Neg32(t21)
13: t25 = Or32(t21, t24)
14: t5 = Add32(t6, 0xFFFFC0CC:I32)
15: t27 = CmpNEZ32(t25)
16: DIRTY t27 RdFX-gst(16, 4) RdFX-gst(60, 4)
        ::: helperc_value_check4_fail{0x380035F4}()
17: t29 = DIRTY 1:I1 RdFX-gst(16, 4) RdFX-gst(60, 4)
        ::: helperc_LOADV32le{0x38006504}(t5)
18: t10 = Ldle:I32(t5)
```

Valgrind: пример трансляции

Распределение регистров

```
                                t21 = Or32(t19, Neg32(t19))

; До распределения регистров
MOV      VR41, VR19
NEG      VR41
MOV      VR40, VR19
OR       VR40, VR41
MOV      VR21, VR40

; После распределения регистров
MOV      EDI, EDX
NEG      EDI
OR       EDX, EDI
```

Получение данных для оффлайн-анализа

- Динамическое бинарное инструментирование приводит к замедлению работы программы, особенно в случае подхода D&R.
- Чем более сложен проводимый онлайн-анализ, тем более медленно работает инструментированный код.
- Чем медленнее работает инструментированная программа, тем сложней реализовать в ней необходимый сценарий поведения. Особенно это верно для интерактивных программ и программ, работающих с сетью.
- Во многих случаях целесообразно разбить анализ на два этапа:
  - легковесный онлайн-анализ, собирающий данные в ходе работы программы, и записывающий эти данные в журнал;
  - тяжеловесный оффлайн-анализ, обрабатывающий данные из журнала и делающий окончательные выводы.

Трассировка на уровне машинных команд

- Частный пример использования DBI-систем для подготовки данных для последующего оффлайн-анализа — трассировка на уровне машинных команд.
- Под трассой выполнения программы на уровне машинных команд понимается последовательность снимков состояний гостевой программы перед каждой выполненной командой.
- Состояние должно включать в себя выбранную для выполнения машинную команду, состояние некоторого подмножества гостевых регистров и регионов гостевой памяти.
- Полная трасса позволяет анализировать все аспекты выполнения программы уже после её завершения.
- Возможны оптимизации: сохранение состояния только на входе в блоки трансляции, инкрементальное сохранение состояния.

Литература к лекции

Основные источники

1. Eli Bendersky. Серия статей “How debuggers work”.
2. Anti-debugging Techniques Cheat Sheet в блоге “0xAA — Random notes on security”.
3. Chow J., Garfinkel T., Chen P. M. Decoupling dynamic program analysis from execution in virtual environments // USENIX 2008 Annual Technical Conference on Annual Technical Conference. – 2008. – С. 1-14.
4. Luk C. K. et al. Pin: building customized program analysis tools with dynamic instrumentation // ACM Sigplan Notices. – ACM, 2005. – Т. 40. – №. 6. – С. 190-200.
5. Nethercote N., Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation // ACM Sigplan notices. – ACM, 2007. – Т. 42. – №. 6. – С. 89-100.

Литература к лекции

Дополнительные источники

1. The DWARF Debugging Standard.
2. Dovgalyuk P. Deterministic Replay of System's Execution with Multi-target QEMU Simulator for Dynamic Analysis and Reverse Debugging // CSMR. – 2012. – С. 553-556.
3. Bungale P. P., Luk C. K. PinOS: a programmable framework for whole-system dynamic instrumentation // Proceedings of the 3rd international conference on Virtual execution environments. – ACM, 2007. – С. 137-147.