ИСП РАН

ОЦЕНКА СТЕПЕНИ 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
{fedotoff, vartan, korpse,
kursh, vishnya, oleshka}@ispras.ru

План доклада

1. Зачем анализировать исполняемый (бинарный) код?
2. Как должны быть организованы базовые средства анализа?
   - Эмулятор как средство динамического анализа
   - Анализ трасс
3. Пример использования базовых средств анализа
   - Оценка степени критичности программного дефекта
4. Дальнейшее развитие …

Цели анализа бинарного кода

- Предмет анализа
  - Исполняемый (бинарный) код
  - Отдельная программа / фрагмент распределенной системы
- Классы программ
  - Прикладное / системное ПО
  - Персональные компьютеры, коммуникационное оборудование, мобильные устройства, IoT
- Практические цели
  - Контроль отсутствия НДВ
  - Аудит свойств отдельных алгоритмов
  - Выявление программных дефектов
  - Оценка влияния программного дефекта на безопасность ПО

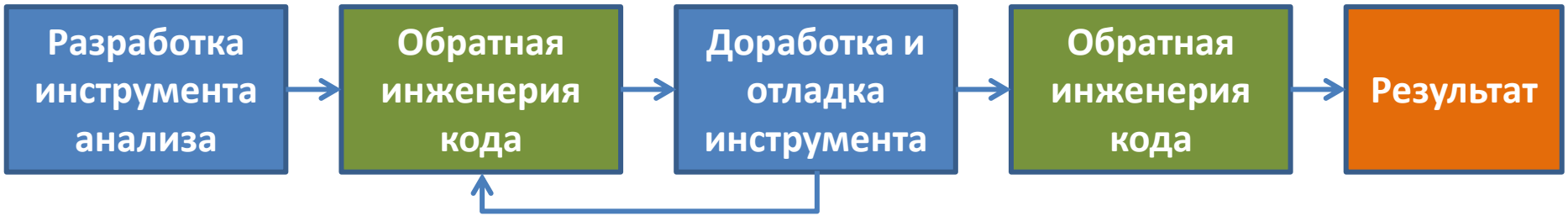
Задачи анализа бинарного кода

- Условия анализа
  - Код построен оптимизирующим компилятором
  - Отсутствует исходный код и отладочная информация
  - Документация на исследуемый код отсутствует частично или полностью
  - Код условно работоспособен
    - Отсутствует спецификация входных данных
    - Для работы требуется специфическая аппаратура
  - Код защищен от анализа
- Необходимо найти интересующий алгоритм в общей массе кода
- Представить алгоритм в «удобной» для анализа форме
  - Выявить свойства алгоритма (вручную или с помощью инструментов)
- Проверить гипотезу: анализируемая программа действительно обладает некоторым свойством (дефектом/ НДВ)

Типовой порядок работы

- Во многих случаях достаточен набор типичных инструментов
  - IDA Pro
  - Интерактивный отладчик
- Однако часто требуется доработать инструменты …
  - … или создать их «с нуля»
- Статика + динамика
  - Исследуемому коду необходима среда выполнения (эмулятор)

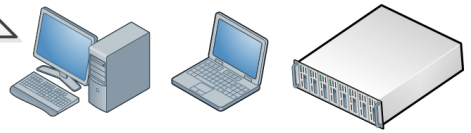
Не менее типовой порядок работы



- Достижение результата зависит не только от эффективности обратной инженерии, но и от скорости разработки инструментов анализа
- Требования к набору инструментов для анализа бинарного кода
  - Интеграция
  - Расширяемость набора
  - Глубокая автоматизация
    - Обратная инженерия бинарного кода
    - Разработка инструментов анализа
  - Метод анализа разрабатывается единожды – применяется везде

	Анализ во время выполнения (online)	Анализ трасс выполнения (offline)	
Прикладные задачи	Сервер-приманка (honeypot)	Контроль отсутствия НДВ	Исследование вредоносного ПО
Анализ потоков данных и управления	Фиксация срабатывания ошибок	Извлечение спецификации алгоритма	Восстановление формата данных
Повышение уровня представления	Интроспекция виртуальной машины	Восстановление статико-динамического представления программы	
Контролируемая среда выполнения на базе эмулятора	Интерфейс отладчика	Сбор трасс	
	Детерминированное воспроизведение	Механизм «обратной» отладки	
Разработка новых виртуальных платформ	Процессор	Периферийные устройства	Интеграция виртуальной платформы

Анализ ПО устройств различных классов



Персональные компьютеры и сервера

Мобильные устройства

Коммуникационное оборудование

Встраиваемые системы и загрузчики

Контролируемая среда выполнения на базе эмулятора Qemu (1/2)

- Эмулятор с открытым исходным кодом, непрерывно развивается, используется в индустрии
  - Поддержаны основные семейства процессоров: x86, ARM, MIPS, PowerPC
  - Поддерживает интерфейс отладчика
- Детерминированное воспроизведение
  - Запись легковесного журнала недетерминированных событий
  - Начальный снимок состояния ВМ и журнал воспроизводят выполнение с точностью уровня машинных команд
  - Разработано ИСП РАН, включено в публичную версию Qemu 2.8
- Обратная отладка
  - Reverse step, Reverse continue

Контролируемая среда выполнения на базе эмулятора Qemu (2/2)

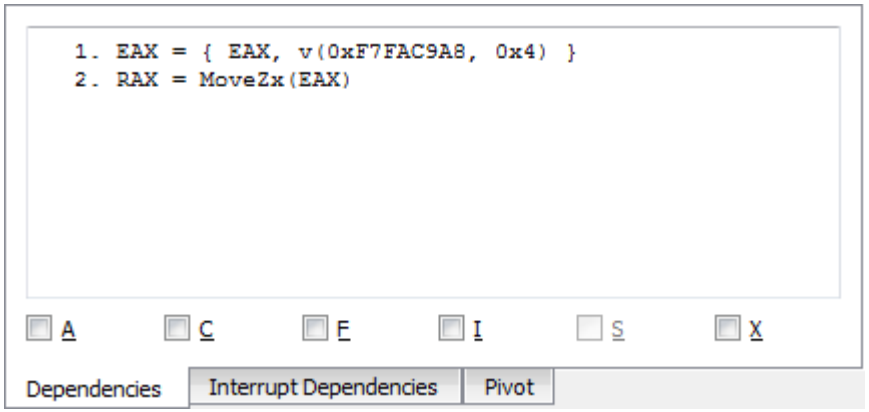
- Интроспекция ВМ
  - Механизм плагинов анализа, встроенный в эмулятор
- Полносистемный анализ помеченных данных
  - Достаточно высокая скорость работы
  - Проблемы избыточной/недостаточной помеченности
  - Недетерминизм анализа

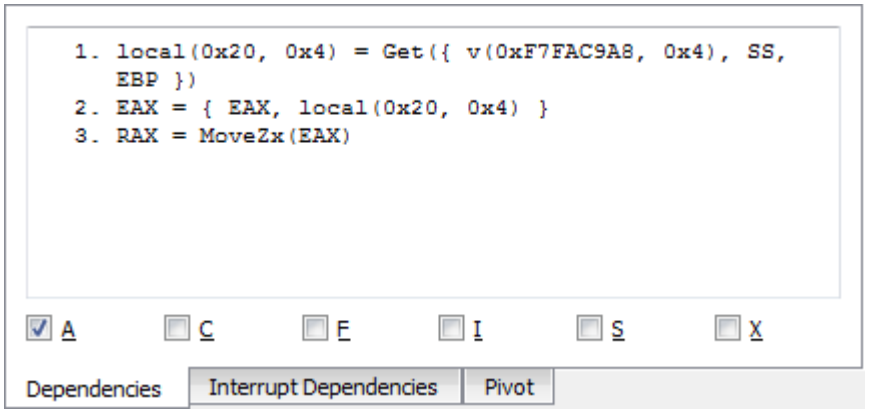
Анализ трасс выполнения

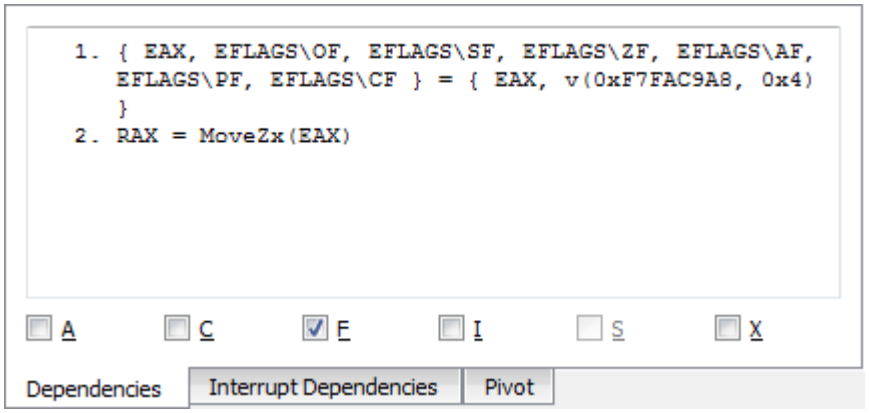
- Поддержка основных процессорных архитектур
  - x86, ARM, MIPS, PowerPC
- Модульная структура
- Большая часть алгоритмов распараллелена на общей памяти
- Основные цели
  - Поднятие уровня представления трассы
  - Восстановление «статического» представления
  - Автоматизация анализа потоков данных
- Поддержка ручного и автоматизированного анализа

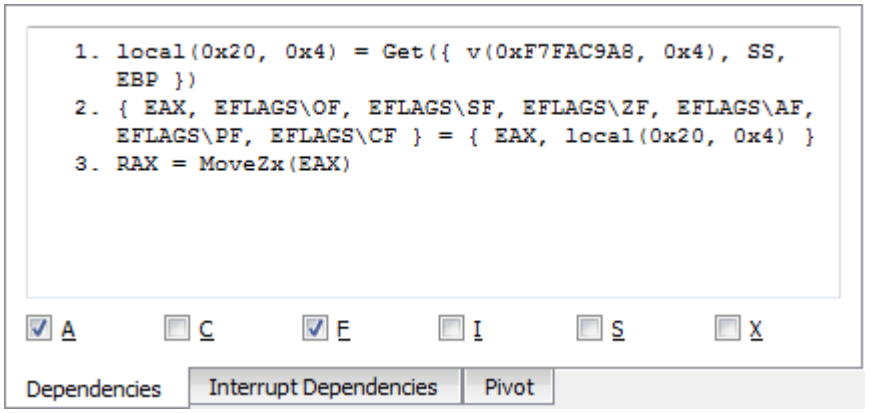
Поднятие уровня представления

Tx: Зависимости по данным
ADD EAX, DWORD SS:[EBP – 10h]

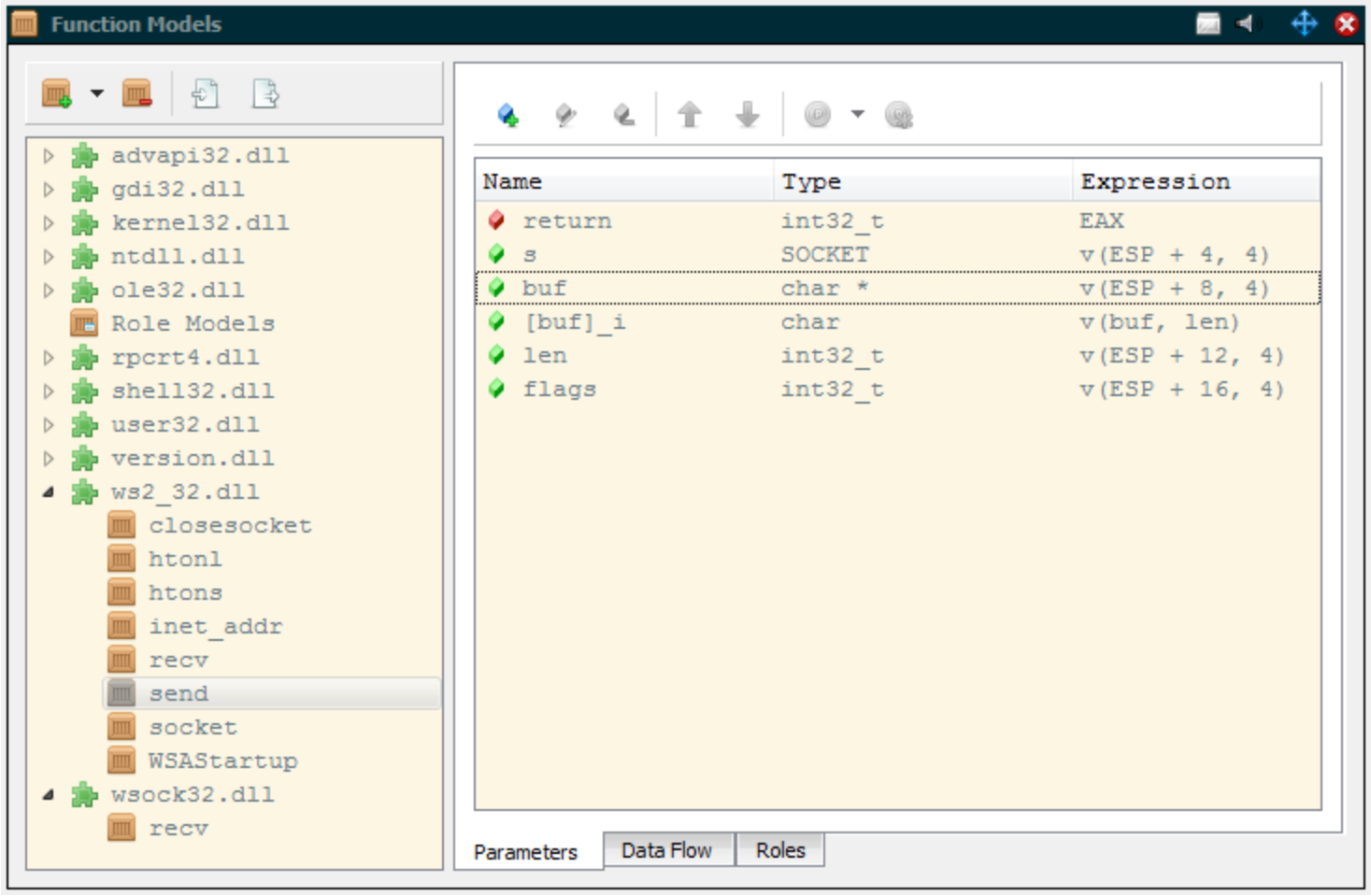


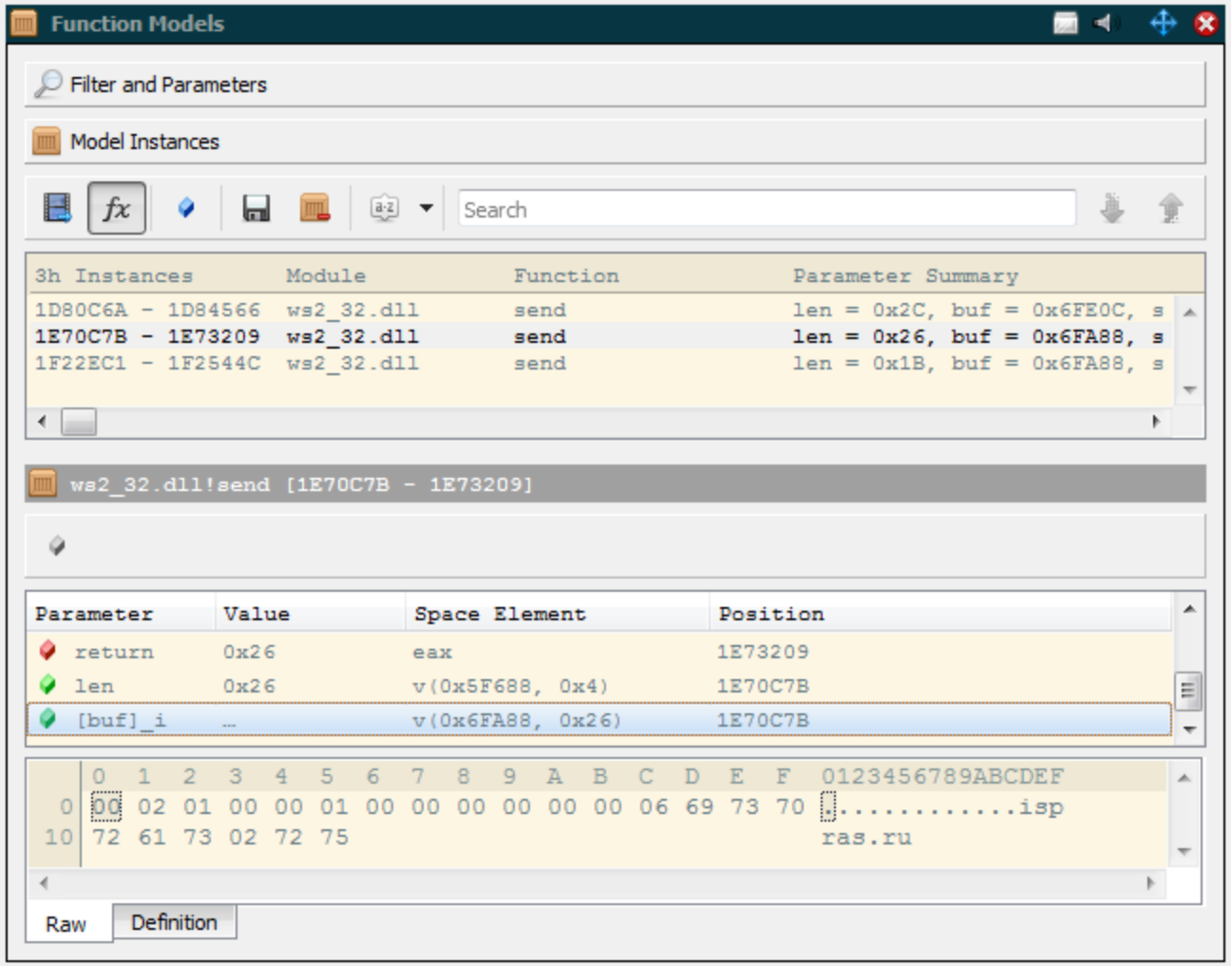


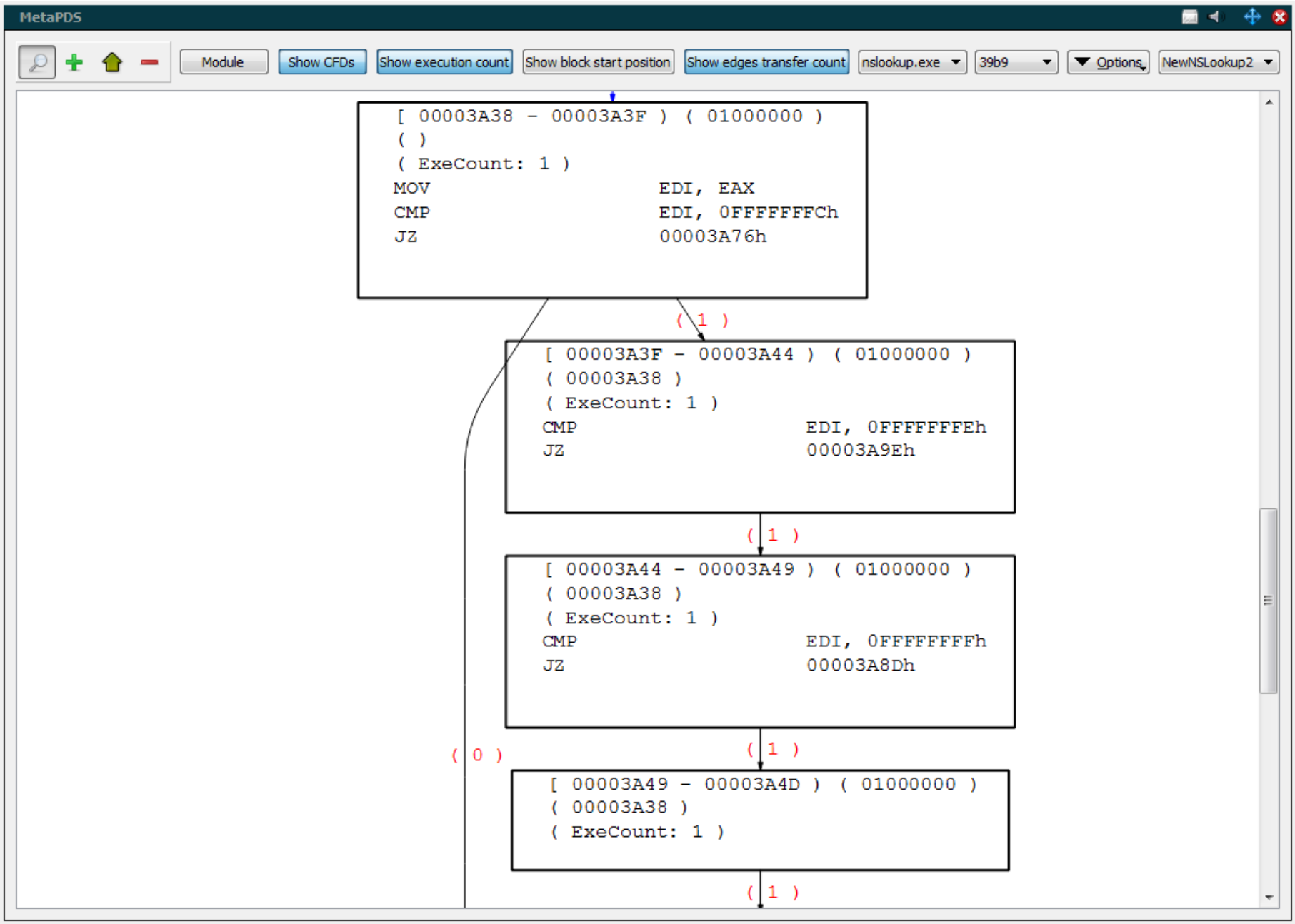




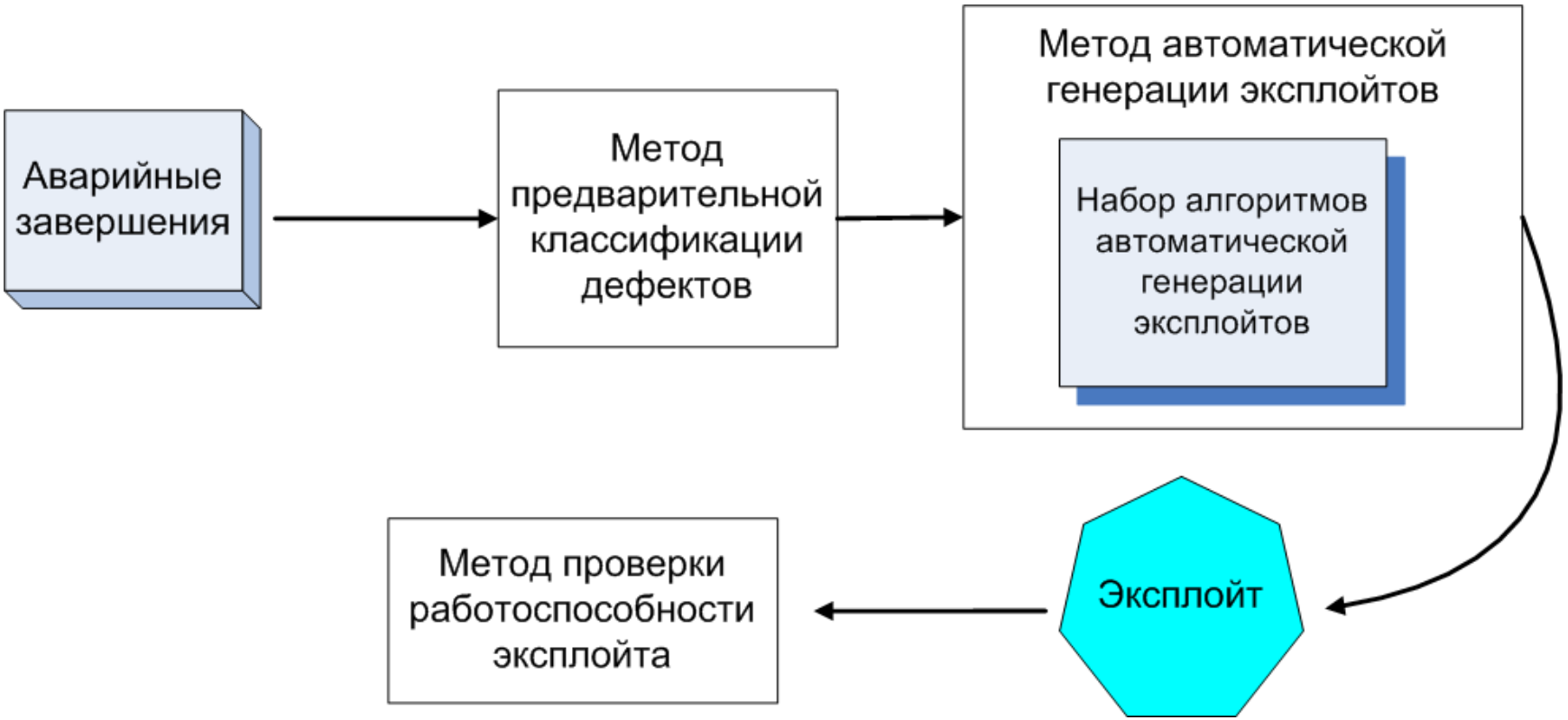
- В одной машинной команде может выполняться несколько присвоений
- «Дробление» зависимостей между входными и выходными операндами уточняет анализ потоков данных


Function Models
advapi32.dll
gdi32.dll
kernel32.dll
ntdll.dll
ole32.dll
Role Models
rpcrt4.dll
shell32.dll
user32.dll
version.dll
ws2_32.dll
closesocket
htonl
htons
inet_addr
recv
send
socket
WSAStartup
wsock32.dll
recv
Name
Type
Expression
return
int32_t
EAX
s
SOCKET
v(ESP + 4, 4)
buf
char *
v(ESP + 8, 4)
[buf]_i
char
v(buf, len)
len
int32_t
v(ESP + 12, 4)
flags
int32_t
v(ESP + 16, 4)
Parameters
Data Flow
Roles

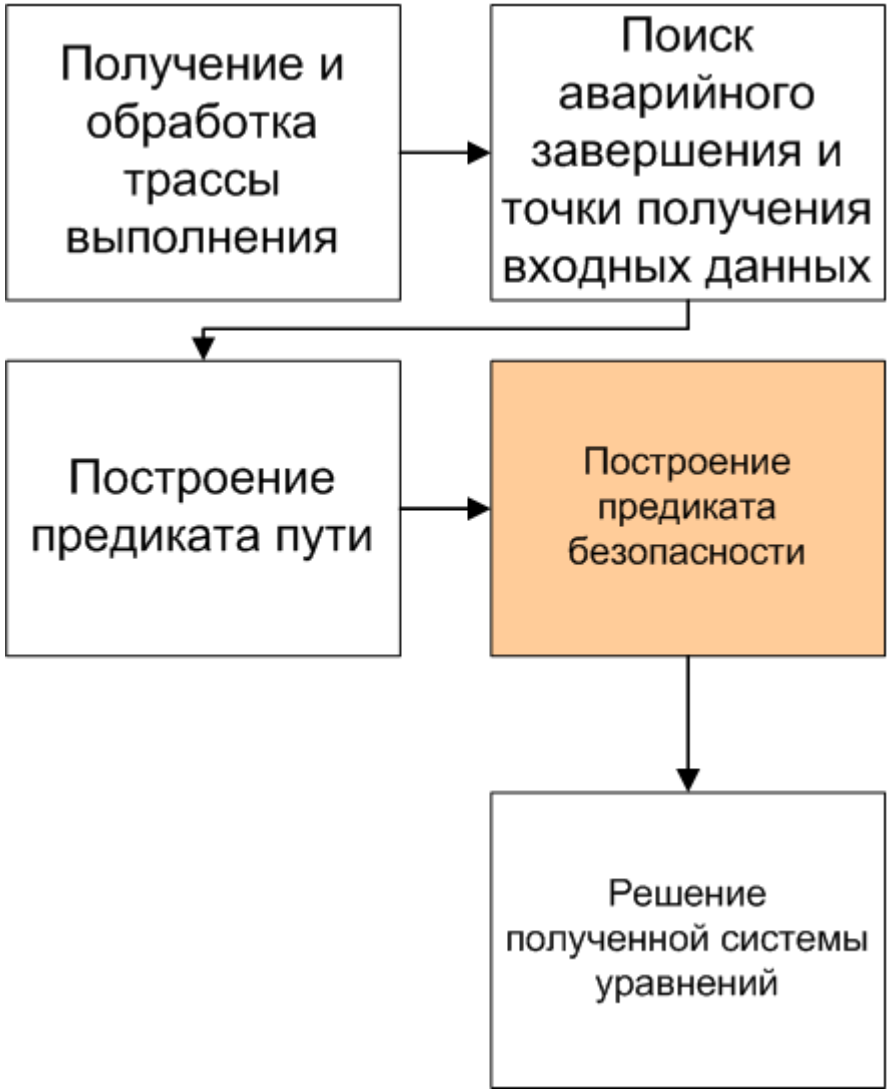

Function Models
Filter and Parameters
Model Instances
Search
3h Instances Module Function Parameter Summary
1D80C6A - 1D84566 ws2_32.dll send len = 0x2C, buf = 0x6FE0C, s
1E70C7B - 1E73209 ws2_32.dll send len = 0x26, buf = 0x6FA88, s
1F22EC1 - 1F2544C ws2_32.dll send len = 0x1B, buf = 0x6FA88, s
ws2_32.dll!send [1E70C7B - 1E73209]
Parameter Value Space Element Position
return 0x26 eax 1E73209
len 0x26 v(0x5F688, 0x4) 1E70C7B
[buf]_i … v(0x6FA88, 0x26) 1E70C7B
0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0 00 02 01 00 00 01 00 00 00 00 00 00 06 69 73 70 ....isp
10 72 61 73 02 72 75 ras.ru
Raw
Definition


MetaPDS
Module
Show CFDs
Show execution count
Show block start position
Show edges transfer count
nslookup.exe
39b9
Options
NewNSLookup2
[00003A38 - 00003A3F) (01000000)
( )
(ExeCount: 1)
MOV EDI, EAX
CMP EDI, 0FFFFFFFCh
JZ 00003A76h
(1)
[00003A3F - 00003A44) (01000000)
(00003A38)
(ExeCount: 1)
CMP EDI, 0FFFFFFFEh
JZ 00003A9Eh
(1)
[00003A44 - 00003A49) (01000000)
(00003A38)
(ExeCount: 1)
CMP EDI, 0FFFFFFFFh
JZ 00003A8Dh
(0)
(1)
[00003A49 - 00003A4D) (01000000)
(00003A38)
(ExeCount: 1)
(1)

Метод оценки степени критичности программных дефектов



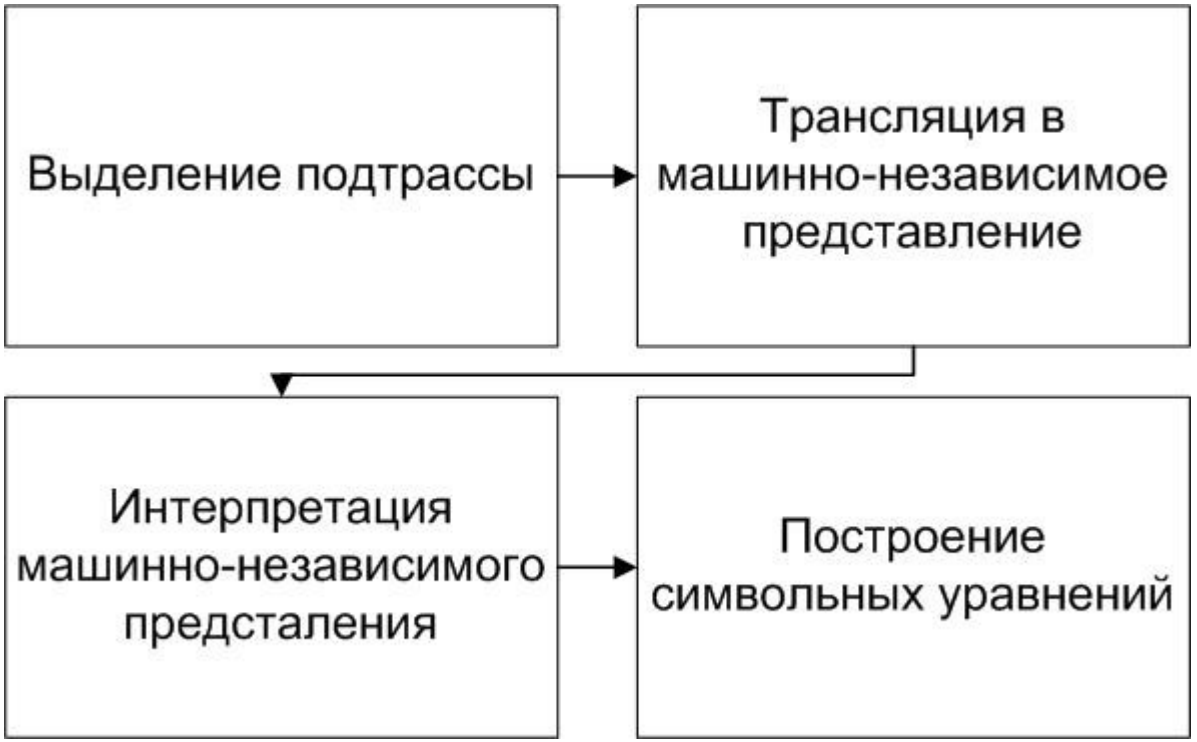
Автоматическая генерация эксплойтов



Поиск точек получения входных данных и аварийных завершений

- Анализ функций получения входных данных и их параметров. Применяется для источников входных данных: сеть, файлы.
  - Известные функции из библиотек (libc.so, kernel32.dll и т.д.)
  - Для определённой версии библиотеки достаточно сделать описание функций один раз, которое можно использовать в дальнейшем
- Поиск первой выполненной инструкции модуля или функции main(). Применяется для данных, получаемых из аргументов командной строки
- Нарушение нормального выполнения программы порождает прерывание.
  - Отбор прерываний, из которых не было возврата.
  - Обработка инструкций передачи управления или записи в память.
  - Проверка помеченных операндов у инструкции.

Построение предиката пути



Анализ современных дистрибутивов ОС Linux на предмет защищенности исполняемых файлов

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%

Часто используемые защиты в дистрибутивах ОС Linux: ASLR, DEP, «канарейка» и FORTIFY_SOURCE

Построение предиката безопасности

- Поддерживаются следующие предикаты безопасности
  - Предикат безопасности для эксплуатации уязвимости переполнения буфера на стеке.
  - Предикат безопасности для эксплуатации уязвимости переполнения буфера на стеке при работе ASLR.
  - Предикат безопасности для эксплуатации уязвимости переполнения буфера на стеке при работе DEP+ASLR.
  - Предикат безопасности для эксплуатации ошибки типа CWE-123, позволяющий обходить «канарейку».
  - Предикат безопасности для эксплуатации ошибки типа CWE-123, позволяющий обходить «канарейку» + DEP + ASLR.

Оценка эксплуатируемости полученных дефектов в результате фаззинга Debian 6.0.10

Группа аварийных завершений	Класс аварийных завершений	Количество аварийных завершений
эксплуатируемые	Исключение при доступе к памяти, адрес которой совпадает со счётчиком команд	13
Возможно эксплуатируемые	Переполнение буфера на куче	23
Не удалось установить принадлежность к группе	Нарушение доступа	238

274 аварийных завершения → 13 рабочих эксплойтов при отсутствии DEP и ASLR. Время работы ≈ 10 часов. Предварительная классификация / генерация эксплойта 1ого примера ≈ 1мин / 21 мин

Результаты применения разработанных методов и подходов

- Использование описания параметров функции v*sscanf* позволило сгенерировать работоспособный эксплойт для программы faad (32-битный Debian 8.3.0).
- Использование прекращения отслеживания параметров функции *malloc* позволило сократить эксплуатацию уязвимости переполнения буфера на стеке в программе blast2 с 6 часов до 20 минут. (32-битный Debian 6.0.10).
- Эксплуатация уязвимости переполнения буфера в программе zsnes при работе DEP и ASLR. (32-битный Debian 8.3.0).
- Эксплуатация переполнения буфера на стеке модельного примера модуля ядра Linux при работе DEP и ASLR. (32-битный Debian 8.3.0).
- Эксплуатация ошибки с типом CWE-123 при наличии канарейки, а также при работе DEP и ASLR на модельных примерах. (32-битный Debian 8.3.0).

Следующее поколение среды анализа

Предварительное исследование ПО

- Отладка в виртуальной машине — возможность предварительно оценить поведение системы.
- Сценарий работы аналитика предполагает необходимость:
  - реверсивной отладки: только на более поздний момент времени может стать понятно, какие нужны точки останова и отслеживаемые переменные на более раннем этапе;
  - интроспекции: какой выполняется процесс, поток, какие загружены модули и т.д.
- Необходимо сократить разрыв между предварительным исследованием и дальнейшим полновесным анализом, если он необходим.
- Перспектива — специализированный отладчик как часть платформы анализа.

Следующее поколение среды анализа

Внутреннее представление (1)

- Анализ потоков данных в динамике уже поддержан с максимальной детализацией.
  - Решается задача выделения алгоритма
  - Результат представлен в виде команд целевой машины
- Требуется описание семантики машинных команд
  - Упрощение представления алгоритма (в том числе – деобфускация)
- Наличие семантики позволяет также перейти к:
  - абстрактной интерпретации;
  - проверке гипотез с применением SMT-решателей;
  - символьным вычислениям;
  - более простому построению контрольных примеров.

Следующее поколение среды анализа

Внутреннее представление (2)

- Ключевые вопросы
  - выбор набора примитивных операций
  - описание побочных эффектов (обновление флагов).
- Операции — QFBV для максимально простой генерации систем SMT-уравнений.
- Побочные эффекты — две стратегии:
  - Явные: Vine, BAP, REIL, Pivot
    проще для дальнейшего автоматического анализа;
  - Неявные: Pivot, VEX
    листинг достаточно компактен и понятен пользователю (ручной анализ);

REIL (OpenREIL)
TEST EAX, EAX

```
00000000.00     STR        R_EAX:32,              ,         V_00:32
00000000.01     STR             0:1,              ,          R_CF:1
00000000.02     AND         V_00:32,          ff:8,          V_01:8
00000000.03     SHR          V_01:8,           7:8,          V_02:8
00000000.04     SHR          V_01:8,           6:8,          V_03:8
00000000.05     XOR          V_02:8,       V_03:8,          V_04:8
00000000.06     SHR          V_01:8,           5:8,          V_05:8
00000000.07     SHR          V_01:8,           4:8,          V_06:8
00000000.08     XOR          V_05:8,       V_06:8,          V_07:8
00000000.09     XOR          V_04:8,       V_07:8,          V_08:8
00000000.0a     SHR          V_01:8,           3:8,          V_09:8
00000000.0b     SHR          V_01:8,           2:8,          V_10:8
00000000.0c     XOR          V_09:8,       V_10:8,          V_11:8
00000000.0d     SHR          V_01:8,           1:8,          V_12:8
00000000.0e     XOR          V_12:8,       V_01:8,          V_13:8
00000000.0f     XOR          V_11:8,       V_13:8,          V_14:8
00000000.10     XOR          V_08:8,       V_14:8,          V_15:8
00000000.11     AND          V_15:8,           1:1,          V_16:1
00000000.12     NOT          V_16:1,             ,           R_PF:1
00000000.13     STR             0:1,             ,           R_AF:1
00000000.14      EQ         V_00:32,          0:32,          R_ZF:1
00000000.15     SHR         V_00:32,         1f:32,         V_17:32
00000000.16     AND            1:32,      V_17:32,         V_18:32
00000000.17      EQ            1:32,      V_18:32,          R_SF:1
00000000.18     STR             0:1,             ,          R_OF:1
```

Vine IL
ADD EAX, 2

```
tmp1 = EAX; EAX = EAX + 2;

//eflags calculation
CF:reg1_t = (EAX<tmp1);
tmp2 = cast(low, EAX, reg8_t);
PF = (!cast(low,
       ((((tmp2>>7)^(tmp2>>6))^((tmp2>>5)^(tmp2>>4)))^
       (((tmp2>>3)^(tmp2>>2))^((tmp2>>1)^tmp2)))), reg1_t);
AF = (1==(16&(EAX^(tmp1^2))));
ZF = (EAX==0);
SF = (1==(1&(EAX>>31)));
OF = (1==(1&(((tmp1^(2^0xFFFFFFFF))&(tmp1^EAX))>>31)));
```

BAP IL

ADD EBX, EAX / SHL EBX, CL / JC target

```
 1  addr 0x0 @asm ”add %eax, %ebx”
 2  t:u32 = R_EBX:u32
 3  R_EBX:u32 = R_EBX:u32 + R_EAX:u32
 4  R_CF:bool = R_EBX:u32 < t:u32
 5  addr 0x2 @asm ”shl %cl, %ebx”
 6  t_1:u32 = R_EBX:u32 >> 0x20:u32 – (R_ECX:u32 & 0x1f:u32)
 7  R_CF:bool =
 8      ((R_ECX:u32 & 0x1f:u32) = 0:u32) & R_CF:bool |
 9     ~ ((R_ECX:u32 & 0x1f:u32) = 0:u32) & low:bool(t1:u32)
10  addr 0x4 @asm ”jc 0x000000000000000a”
11  cjmp R_CF:bool, 0xa:u32, ”nocjmp0”
12  label nocjmp0
```

Приведен листинг после удаления избыточных вычислений.

Tx: Pivot

TEST EAX, EAX

```
1.   INIT o.0:i16  = 0000h  # EAX
2.   LOAD t.0:i32  = r[o.0]
3.   APPL t.1      = and.i32(t.0, t.0)[:ACOPSZ]
4.   INIT t.2:i16  = 0088h  # EFLAGS
5.   LOAD t.3:i16  = r[t.2]
6.   INIT t.4:i16  = 08C5h
7.   APPL t.5      = x86.uf(t.3, t.4)[ACOPSZ:]
8.   STOR r[t.2]   = t.5
```

Следующее поколение среды анализа

Текущие ограничения

Два сценария, которые представляют наибольшие сложности для современной версии среды анализа.

1. Анализ встроенного ПО (прошивок) для микропроцессоров.
   - Разработка (фрагмента) эмулятора;
   - Поддержка архитектуры (системы команд) в среде анализа
2. Анализ больших объемов кода, длительный сценарий взаимодействия с анализируемой системой.

Следующее поколение среды анализа

Анализ встроенного ПО

- Объем ПО невелик.
- Большое количество вариантов аппаратуры (на уровне наборов команд и конкретных SoC).
- Необходима модульность платформы анализа.
- Максимально возможная часть платформы должна работать над общим внутренним представлением, а не над машинным кодом как таковым.

Следующее поколение среды анализа

Анализ больших трасс

- Хорошо поддержанное целевое окружение (x86/ARM, Windows/Linux).
- Сжатая трасса большого объема (до 500 Гбайт).
- Существующие алгоритмы распараллелены для систем с общей памятью.
  - Результаты распараллеливания хорошие, но «потолок» близко.
- Перспектива — платформа анализа как сервис, работающий на кластере и состоящий из отдельных модулей для решаемых подзадач.