VmProtect и VmSweeper
Автор: Vam Vamit

Создано: 25 марта 2010 09:26:19

Проведя небольшой анализ VMProtect (до точки вызова обработчиков примитивов) выявил следующие отличия в его ВМ по сравнению с Орeановскими ВМ:
1. Инициализация ВМ (конечная точка – вход в цикл интерпретатора ВМ)

Code:

	Описание	VMProtect	CodeVirtualizer
1.	Код инициализации ВМ	обфусцирован	явный
2.	Адрес таблицы пикода (дно стека)	закодирован	доступен
3.	Порядок сохранения нативных регистров в стеке	произвольный	фиксированный
4.	Расположение регистров ВМ	в стеке	в области кода
5.	Указатель на регистры ВМ	**EDI**	**EDI**
6.	Ключ кодирования инициализирован адресом пикода	**EBX**	**EBX**
7.	Указатель на стек программы	**EBP**	**ESP**

2. Интерпретатор ВМ (конечная точка – вызов обработчика примитива)

Code:

	Описание	VMProtect	CodeVirtualizer
1.	Порядок просмотра пикода	прямой/обратный	прямой
2.	Таблица обработчиков примитивов	в коде (256DW)	в коде (<256DW)
3.	Адрес обработчика примитива	закодирован	доступен

Выводы:
1. Реализовать в декомпиляторе один обработчик разных типов ВМ проблематично, но это и к лучшему. Проще под каждый тип ВМ сделать собственный обработчик на едином алгоритме, чем делать «солянку» в одном обработчике, по крайней мере, добавление обработчика нового типа ВМ не нарушит работу уже существующих.
2. Понятен общий принцип определения типа ВМ декомпилятором (сейчас он представляет из себя набор «жестких»условий, но на самом деле это не так):
- найти точку входа в цикл интерпретатора ВМ при статическом анализе точки входа в ВМ из тела функции.
- выявить анализатором моменты указанные в таблице 1
- дойти статическим анализом до точки вызова обработчика примитива
- выявить анализатором моменты указанные в таблице 2
- подтвердить моменты таблицы 1
- определить алгоритм хеширования кода примитива
- определить алгоритм хеширования адреса обработчика примитива

А далее, после определения типа ВМ, идем по уже пройденной дорожке:
- идентифицируем обработчики примитивов ВМ
- «заменяем» интерпретатор ВМ с обработчиками примитивов промежуточным кодом и т.д., что уже было описано и реализовано ранее…

Создано: 21 апреля 2010 09:27:08

VMProtect (VMP) - декомпилятором получены чистые тела всех обработчиков примитивов для двух реализаций ВМ. Приведу основные отличия обработчиков по сравнению с **CodeVirtualizer (CV)**. Тела обработчиков VMP имеют упорядоченную структуру, поэтому для их получения и распознавания пришлось разобрать вручную около десятка обработчиков, написать пару правил, применить двухуровневую регистровую деобфускацию - и всё, со всем остальным декомпилятор справился сам. В CV же для получения примитивов требовалось сначала вручную разобрать практически каждый обработчик, затем создать шаблон(ы), и только затем декомпилятор на основе шаблона(ов) мог распознать примитив. Есть несколько основных отличий от CV:

1. Использование под контекст ВМ "отдельного стека" с которым идет работа как с памятью (через ebp). Основной стек после обработчиков не меняется. Шаблон примитива в таком случае получается однозначным. В CV же под контекст ВМ используется основной стек - это определяет основную сложность в локализации примитивов, т.к. вариантов работы со стеком куда больше, чем с памятью. Для распознавания примитивов простой регистровой деобфускации здесь недостаточно, нужно отслеживать не только изменения вершины стека, но и его содержимое. Например, примитив xchg регистра со стеком имеет около десятка разных реализаций, пока для его распознавания у меня используется 6 шаблонов, но я думаю это не предел, т.к. (см. первый пост) "каждая третья реализация автоматом не определяется".
2. Кол-во примитивов. В CV - фиксированное кол-во и не зависит от содержимого защищаемой функции, в этом случае многие примитивы имеют только тела, но никогда не используются, что затрудняет их анализ. В VMP же набор примитивов в каждой реализации ВМ динамический (в пределах 255), в набор включены только нужные (используемые) примитивы и очень много их алиасов. Это упрощает их анализ для одной реализации ВМ, но в другой ВМ могут встретиться "ещё неизвестные" примитивы, что потребует доработки декомпилятора только в части добавления шаблона, но не изменения анализа.
3. "Мусорные" (незначимые, dummy) примитивы. В VMP не обнаружены. В CV таких примитивов достаточно много и их легко спутать с другими простыми примитивами (тот же xchg).
4. Обфускация обработчиков. В VMP она гораздо "круче", но только с виду, т.к. используется большое кол-во (практически весь набор инструкций i486) раритетных команд, которые легко очищаются регистровой и стековой (т.к. его вершина постоянна) деобфускацией. Наличие "двойных" маршрутов в одном обработчике с одинаковой логикой анализ не усложняет. В ручную VMP разбирать сложнее, но не декомпилятору.
5. Получение индексов регистров ВМ. В VMP индексом регистра является сам опкод примитива, т.е. для чтения и записи в разные регистры ВМ используются разные примитивы, но с одинаковым телом. В CV же индекс регистра является дополнительным хешированным байтом в пикоде (в последних версиях это применено и в VMP). С точки зрения обработки - сложностей в обоих случаях нет.

Выводы об эмуляции реальных инструкций пока делать рано, они появятся только на стадии разбора промежуточного кода, т.к. одна инструкция может быть представлена n-кол-вом примитивов.

В связи с тем, что каждая реализация VMP имеет ограниченный набор примитивов, хотелось бы заранее предусмотреть некоторые моменты, которые пока не встретились, поэтому есть несколько вопросов знатокам VMP (Примечание: Ниже вопросов приведены ответы):

1. Обработчики с хешированием констант пикода. Всегда ли одинаковые обработчики имеют одинаковый алгоритм хеширования?
   (Каждый обработчик может иметь свой алгоритм хеширования).
2. Проверка переполнения стека контекста ВМ ("наезд" на регистры ВМ). Реально ли она происходит или это сделано только на непредвиденный случай, например, как защита?
   (Происходит реально, например при выделении места в стеке под локальные переменные функции).
3. Функциональный состав обработчиков. В VMP я не встретил обработчиков sub, idiv, imul, циклических сдвигов и некоторых других, хотя, обработчики add, div, mul имеются. Вопрос или они отсутствуют вообще и эти инструкции эмулируются, или они пока не встретились в разобранных реализациях ВМ? Хотя, если есть обработчик mul, то эмулировать imul бесполезно, или я ошибаюсь?
   (Прямых обработчиков этих и других инструкций нет, т.к. практически все инструкции виртуализованы VMP и состоят из нескольких обработчиков).

Эксклюзивный материал, ранее нигде не публиковался – **Чистые тела обработчиков пикода (см. Приложение 1)**

Создано: 7 мая 2010 09:46:08

Некоторые теоретические соображения по декомпиляции промежуточного кода независимые от ВМ.
В принципе линейный (последовательный) асм код может выполнять одновременно две разные задачи, если он подготовлен для этого специальным образом.
Попробую показать это: Асм инструкции оперируют большей частью регистрами и константами (immediate). Одиночные константы в инструкциях практически не встречаются (не учитываем все команды передачи управления), а используются в совокупности с регистрами, поэтому их мы в расчет не берем. Остаются регисты и если иметь два набора регистров и "размешать" асм инструкции одного набора инструкциями другого набора, то такой код сможет выполнять одновременно две разные задачи.
Практическая реализация такого кода возможна в ВМ. Описывать как это делается я не буду, кто знаком с ВМ, тот поймет. Добавлю только, что в такой код можно добавить ещё и "мусор" (обфускировать), который никакой полезной работы не делает.
CodeVitrualizer - полезная работа кода одна (выполнение контекста защищенной функции) + "мусор".
VMProtect - полезных работ две (выполнение контекста защищенной функции + реализация защитных механизмов) + "мусор".
Наша конечная задача восстановить код контекста защищенной функции, всё остальное для этой задачи "мусор" и подлежит удалению.
Это можно сделать двумя способами:

1. Академический - первоначально восстанавливаются оба рабочих кода (контекст + защита), затем защита удаляется. Плюс - можно отследить методы защиты. Минус - более полный разбор кода.
2. Практический - сразу выделяется только код контекста, остальное удаляется. Плюсы, минусы - противоположные от первого.

Далее, весь этот код может быть виртуализован (CV нет, VMP да), девиртуализацию необходимо делать при любом способе, иначе:
а). может быть потерян смысл восстановленного контекста
б). контекст может не поместиться в отведенное ему место

Создано: 7 июня 2010 09:37:47 (эксклюзивный материал)

Для полного понимания процессов и правил на стадии декомпиляции промкода ВмПротекта буду кратко описывать по мере реализации все шаги и правила для них.
Шаг 1: Разбивка промкода на функциональные блоки.
Блоки vBLK_ENTRY, vBLK_RESTORE - вход/выход из ВМ, включают в себя не только предварительный/заключительный код входа/выхода, но и часть примитивов ВМ, которые загружают/выгружают контент программы по регистрам ВМ/регистрам CPU.
Особенности:
1. Начало vBLK_ENTRY всегда известно, как для первого входа в ВМ, так и для всех последующих - это начало условного примитива EntryVm. Конец vBLK_ENTRY будет различен для разных типов кодируемых частей функций. Если ВМ кодирует часть функции конец vBLK_ENTRY будет на конце примитива PopSvmLong, встретившегося первым, т.к. в этой точке стек ВМ выравнивается со стеком контента функции. Если ВМ кодируется вся функция startup, то где будет конец vBLK_ENTRY я пока не знаю, это предстоит ещё выяснить.
2. Конец vBLK_RESTORE всегда известен, это конец примитива ExitVm. Началом vBLK_RESTORE будет первый примитив PushRvmLong загрузки первого регистра ВМ, соответствующего регистру CPU, в стек.
3. Эти блоки не имеют дальнейшего текстового представления, т.е. изымаются из дальнейшего анализа.
4. Блок vBLK_RESTORE может делать кодирование регистров CPU, а декодировка производиться в asm вставке, в принципе это не должно стать проблемой, т.к. на присвоениях/изменениях регистров блоки создаваться не будут (это будет показано ниже) и если всё сделано правильно, свертка/развертка регистров должна пройти втихую и без отображения в дальнейшем тексте.
5. В блоке vBLK_ENTRY запоминается порядок размещения регистров CPU по регистрам ВМ, а в блоке vBLK_RESTORE запоминается порядок размещения регистров ВМ по регистрам CPU. В дальнейшем на основании этой информации принимается решение о соответствии регистров CPU-ВМ между этими блоками.

Создано: 7 июня 2010 20:12:22 (эксклюзивный материал)

Блок vBLK_CNGFLAG - изменения флагов CPU.
Особенности:
1. Конец блока ставится на инструкцию после которой происходит сохранение флагов в стеке (pushfd) с последующей загрузкой в регистр ВМ. Начало блока не определено, юстируется по концу предыдущего блока.
2. Текстовое представление - res = instr op1, op2;
где: res - место сохранения результата операции, если в регистре, то, например, __$eax = совпадает с операндом op1; если на вершине стека, то [__$esp] =; если в памяти, то [addr] =.
instr - мнемоника инструкции, например, add
op1 - операнд 1, правила как у res
op2 - операнд 2, правила как у res
конкретный пример: [__$esp] = add [414018], 0xFFFFFFDC - говорит о том, что значение из памяти сложено с константой и результат помещен на вершину стека

Создано: 8 июня 2010 13:49:15 (эксклюзивный материал)

Блок vBLK_ASSIGN - присвоение переменных.
Данный блок имеет 5 типов:
vASSIGN_MEM - присвоение ячейке памяти.
Ставится на любую инструкцию осуществляющую запись в память. Понятия память и стек разные, поэтому их нельзя путать, и данный блок работает только с памятью. Пример текстового представления - [41401C] = [__$esp] - говорит о том, что значение с вершины стека помещено в память по адресу 41401C.
vASSIGN_REGVM - присвоение регистру ВМ.
В ВмПротекте не используется, т.к. все регистры ВМ имеют "плавающее" предназначение. Вместо этого блока используется vASSIGN_MEM.
vASSIGN_REG - присвоение регистру.
В ВмПротекте используется только присвоение регистру esp, т.к. через него происходит реальное изменение размеров стека, остальные регистры CPU в обменах данными между примитивами ВМ не участвуют, поэтому присвоения на них не ставятся. Пример текстового представления - __$esp = [__$esp] - говорит о том, что значение с вершины стека помещено в регистр esp.
vASSIGN_ARG - присвоение аргументу.
Пример текстового представления - push [41401C]
vASSIGN_POP - извлечение из стека.
Пример текстового представления - pop [41401C]

Создано: 5 августа 2010 09:42:13

Виртуализация инструкций CPU VmProtect и их практическая девиртуализация.
На сегодняшний день девиртуализованы практически все основные инструкции CPU, в большинстве их вариаций как по размеру операндов, так и по типам операндов. Построены универсальные шаблоны девиртуализации инструкций, в большинстве они охватывают все вариации операндов, но некоторые инструкции требуют отдельных шаблонов на разные вариации, например, на строковые инструкции MOVS, LODS и другие используется по три шаблона в зависимости от размера операнда (BYTE, WORD, DWORD).
Для определения алгоритмов виртуализации было написано около 200 тестовых примеров с использованием конкретных инструкций, затем эти примеры были обработаны ВмПротектом с разными опциями преобразования кода - виртуализация, мутация, шифрация регистров и скрытие констант. Далее полученные файлы обрабатывались Декомпилятором и проводился анализ виртуализации, на основе этого строились шаблоны и выполнялась практическая девиртуализация инструкций.

Основные общие принципы виртуализации инструкций (VmProtect):
1. Измененение основного представления инструкции.
Большинство инструкций имеют следующий формат или схожий с ним - code dst, src; где code - мнемоника инструкции, dst - входной операнд и

одновременно операнд результата, src - входной операнд. За реальными примерами далеко ходить не нужно - add x, y; dec x и т.д. movs и аналогичные тоже попадают в эту категорию, т.к. имеют вполне реальные "скрытые" операнды.
Любая из этих инструкций, вернее её составная часть, представляется ВмПротектом следующим образом - dst = code src1, src2; Отсюда видно, что входной операнд и операнд результата разделены, таким образом осуществляется скрытие регистров CPU, т.к. на очередном присвоении они могут поменять свое месторасположение в регистрах ВМ.
2. Флаги CPU.
Флаги, где это возможно, отделены от инструкций и обрабатываются отдельно. Даже "невиртуализованные" инструкции, которые имеют отдельный примитив, состоят из нескольких блоков - сама инструкция + блок обработки флагов. Примеры - rcl, mul и другие.
3. Разбивка инструкции на составляющие части.
Простые логические инструкции виртуализованы через NOR примитив, арифметические - через ADD, некоторые инструкции сдвига - через SHLD, SHRD. Все остальные инструкции виртуализуются через эти простые составляющие.

Отдельно стоят инструкции условных переходов, т.к. здесь виртуализация затрагивает не только саму инструкцию, но и окружающий код в точке перехода и в точке метки перехода, но об этом в следующий раз...

Эксклюзивный материал, ранее нигде не публиковался – **Шаблоны виртуализации инструкций (см. Приложение 2)**

Создано: 8 августа 2010 11:39:43

Виртуализация переходов в VMProtect
Немного терминологии - каждый переход имеет две точки, назовем их так - точка ветвления и точка слияния. Условный переход в точке ветвления имеет два пути - прямой и переход. В точке ветвления происходит разрыв хеширования пикода, т.е. ключ хеширования в этой точке заново инициализируется адресом следующего пикода. В точке слияния разрыва хеширования пикода не происходит, для соблюдения этого условия имеется специальный алгоритм. Следовательно, прямого продолжения пикода в точке ветвления не происходит и выйти на прямой путь или переход можно только через примитив Cmd_Jmp (условный или безусловный).
Пример:

Code:
```
label1:
        jz      label2          // jmp way
        ....                    // direct way
label2:
        ....
```

label1 - точка ветвления, label2 - точка слияния. В точке ветвления имеем два пути - переход или прямой путь. В точке ветвления условно можно получить два адреса продолжения выполнения пикода - по переходу или по прямому пути. Условно потому, что в точке ветвления ВмПротект вычисляет только один адрес в зависимости от состояния нужных флагов, а затем следует безусловный переход на этот адрес.
Т.е. этот пример с точки зрения ВмПротекта на лету (в ходе выполнения) будет превращен в две модификации для обоих путей - перехода и прямого пути, соответственно:

Code:
```
label1:
        // if(z == 1)           // jmp way
        jmp     label2
        ....                    // direct way
label2:
        ....
```

Code:
```
label1:
        // if(z == 0)           // jmp way
        jmp     direct
direct:
        ....                    // direct way
label2:
        ....
```

Теперь о том, как это конкретно работает - перед условным переходом ВмПротект записывает последовательно в стек два кодированных адреса, которые впоследствии превратятся в адреса переходов - direct и jmp. Далее производится анализ нужных флагов, причем анализ может проводиться как на установку, так и на сброс флагов, в результате анализа берется один из кодированных адресов из стека, производится его декодирование и происходит переход на этот адрес.
Таким образом парные (дуальные) переходы, например jz и jnz и т.д., представлены 8 комбинациями (по 4 на каждый): используется адрес перехода в стеке - 1 или 2, фактические флаги - установлены или сброшены, код проверяющий флаги - на установку или сброс и фактический переход - direct или jmp. Проанализировав все эти комбинации, можно 100% точно установить условие перехода в исходном коде.
Но это ещё не всё - вокруг точек ветвления и слияния ВмПротект строит определенные зоны, до точки - зона сохранения контента ВМ (регистры), после точки зона восстановления контента ВМ (регистры), причем восстановление проводится в регистры отличные от сохраненных, таким образом обеспечивается дополнительная защита по сокрытию регистров CPU. Есть ещё одна особенность зоны сохранения контента перед точкой слияния, в ней корректируется ключ хеширования таким образом, чтобы он в точке слияния равнялся адресу пикода, таким образом устраняется разрыв хеширования пикода в точке слияния.

В заключение - пара листингов:
1. Незначительно модифицированый промкод (до девиртуализации)

Code:
```
0042B0E0: rvm_08 = 1
0042B0EB: svm_3 = and ~rvm_08, ~rvm_08
0042B105: rvm_30 = iEFL
0042B10C: svm_4 = add 2, svm_3
0042B110: rvm_1C = iEFL
0042B117: svm_5 = and ~svm_4, ~svm_4
0042B126: rvm_38 = iEFL
0042B12D: rvm_10 = svm_5
0042B133: svm_6 = and ~rvm_1C, ~rvm_1C
0042B149: rvm_2C = iEFL
0042B150: svm_7 = and ~svm_6, 0x00000815
0042B15F: rvm_24 = iEFL
0042B166: svm_8 = and ~rvm_38, ~rvm_38
0042B17C: rvm_24 = iEFL
0042B183: svm_9 = and ~svm_8, 0xFFFFF7EA
```

```
0042B192: rvm_24 = iEFL
0042B199: svm_10 = add svm_7, svm_9
0042B19D: rvm_3C = iEFL
0042B1A4: rvm_30 = svm_10
0042B1AA: svm_11 = and ~rvm_30, ~rvm_30
0042B1D2: rvm_24 = iEFL
0042B1D9: svm_12 = and ~svm_11, 0x00000080
0042B1E8: rvm_2C = iEFL
0042B1EF: rvm_3C = svm_12
0042B1F5: svm_13 = and ~rvm_30, ~rvm_30
0042B20A: rvm_1C = iEFL
0042B211: svm_14 = and ~svm_13, 0x00000800
0042B220: rvm_3C = iEFL
0042B227: rvm_38 = svm_14
0042B22D: svm_15 = and ~rvm_2C, ~rvm_2C
0042B243: rvm_1C = iEFL
0042B24A: svm_16 = and ~rvm_3C, ~rvm_3C
0042B25F: rvm_24 = iEFL
0042B266: svm_17 = and ~svm_16, ~svm_15
0042B270: rvm_10 = iEFL
0042B277: svm_18 = and ~rvm_2C, ~rvm_3C
0042B28D: rvm_1C = iEFL
0042B294: svm_19 = and ~svm_18, ~svm_17
0042B29E: rvm_1C = iEFL
0042B2A5: rvm_1C = svm_19
0042B2B0: svm_20 = and ~rvm_1C, ~svm_19
0042B2C0: rvm_10 = iEFL
0042B2C7: rvm_24 = svm_20
0042B2CD: svm_21 = and ~rvm_24, ~rvm_24
0042B2E3: rvm_10 = iEFL
0042B2EA: svm_22 = and ~svm_21, 0x00000040
0042B2F9: rvm_1C = iEFL
0042B300: svm_23 = shr svm_22, 4
0042B307: rvm_2C = iEFL
0042B30E: svm_24 = add __$esp, svm_23
0042B312: rvm_1C = iEFL
0042B319: rvm_38 = [svm_24]
0042B323: rvm_24 = 0xA810D9D2
0042B329: rvm_1C = 0xA810D631
0042B32F: rvm_10 = rvm_38
0042B340: svm_25 = and ~rvm_10, ~rvm_38
0042B350: rvm_38 = iEFL
0042B357: svm_26 = and ~svm_25, 0xA85251F7
0042B366: rvm_1C = iEFL
0042B36D: svm_27 = and ~rvm_10, 0x57ADAE08
0042B382: rvm_24 = iEFL
0042B389: svm_28 = and ~svm_27, ~svm_26
0042B393: rvm_2C = iEFL
0042B39A: rvm_38 = svm_28
0042B408: jmp 0x0042B40D
0042B40D: label_42B40D:
0042B4CF: rvm_04 = 3
0042B59B: label_42B59B:
0042B65D: svm_29 = add 0xFFFFFFE0, rvm_10
0042B672: rvm_14 = iEFL
0042B679: [svm_29] = rvm_30
```

2. Девиртуализованный декомпилятором код

Code:

```
0042B0E0: rvm_08 = 1
0042B0EB: rvm_10 = cmp rvm_08, 2
0042B1D9: jnl 0x0042B59B
0042B4CF: rvm_04 = 3
0042B59B: label_42B59B:
0042B65D: svm_29 = rvm_10 + 0xFFFFFFE0
0042B679: [svm_29] = rvm_30
```

3. А это исходный код, который был защищен

Code:

```
        mov     eax, 1
        cmp     eax, 2
        jnl     label6
        mov     eax, 3
label6:
        mov     res, eax
```

чтобы получить его, как видно, осталось совсем немного - нужно сделать три шага:
- произвести поглощение стековых переменных (svm)
- восстановить цепочку соответствий регистров CPU регистрам ВМ (rvm)
- ассемблировать этот условный код

Создано: 10 августа 2010 11:50:30 (эксклюзивный материал)

Определение соответствия регистров CPU регистрам ВМ (PBM) в VMProtect

Задача эта в общем-то нетривиальная и окончательный алгоритм её решения ещё не определен.
Регистры CPU могут менять свое положение в PBM в следующих случаях:
- На присвоении (до присвоения регистр CPU занимает один PBM, а после присвоения тот же регистр перемещается в другой PBM.
- В точках переходов и меток. В такой точке все PBM записываются в стек а затем выгружаются на новые места.
- Принудительно. Вставка в код простых инструкций присвоения, например, rvm_20 = rvm_0.

Что имеется на данный момент:
1. Разбивка промкода на зоны сеансов ВМ.
Каждая такая зона начинается блоком - ENTRY, в котором производится первоначальное размещение регистров CPU в PBM, и заканчивается блоком - RESTORE, в котором конкретные PBM выгружаются в регистры CPU. В этих точках 100% известно соответствие регистров CPU PBM.
Пример блоков

Code:

```
1.  ENTRY  :  (0042B040) Start: 0, End: 30, Adjust: DONE
2.  CNGFLAG:  (0042B0C0) Start: 31, End: 35, Adjust: NOT_START
3.  ASSIGN :  (0042B0CF) Start: 36, End: 37, Adjust: NOT_START, Type: REGVM
4.  ..............................
5.  ASSIGN :  (0042B702) Start: 564, End: 565, Adjust: NOT_START, Type: REGVM
6.  RESTORE:  (0042B70F) Start: 567, End: 587, Adjust: NOT_START
```

и соответствия регистров

Code:

```
1.  *** Zone bounds: 0042B040 - 0042B75D ***
2.  *** Regs Associations (zone 0) ***
3.  regs  edx   ebx   eax   ecx   ebp   efl   edi   esp   esi
4.  in    rvm_3C rvm_14 rvm_2C rvm_1C rvm_34 rvm_18 rvm_04 rvm_24 rvm_20
5.  out   rvm_28 rvm_0C rvm_14 rvm_24 rvm_34 rvm_2C rvm_3C rvm_1C rvm_04
6.  false -      -      -      -      -      -      -      +      -
```

2. Выявление блоков (SKIP) сохранения PBM в стеке и восстановления их из стека до и после точек перехода/метки.
В этих точках соответствие регистров CPU PBM напрямую определить невозможно, имеем только информацию о том, что PBM поменял свое место.
Кол-во PBM в этих блоках постоянно и превышает кол-во регистров CPU, часть PBM инициализируется константами.
Пример блоков

Code:

```
1.  ..........................
2.  ASSIGN :  (0042B40C) Start: 351, End: 351, Adjust: NOT_START, Type: REGVM
3.  SKIP   :  (0042B412) Start: 352, End: 372, Adjust: NOT_START
4.  JMP    :  (0042B47A) Start: 373, End: 373, Adjust: NOT_START, Jmp: 374
5.  SKIP   :  (0042B47F) Start: 374, End: 433, Adjust: NOT_START
6.  ASSIGN :  (0042B541) Start: 434, End: 435, Adjust: NOT_START, Type: REGVM
7.  ..........................
8.  SKIP   :  (0042B551) Start: 437, End: 491, Adjust: NOT_START
9.  LABEL  :  (0042B60F) Start: 492, End: 492, Adjust: DONE
10. SKIP   :  (0042B60F) Start: 492, End: 549, Adjust: NOT_START
11. CNGFLAG:  (0042B6D2) Start: 550, End: 554, Adjust: NOT_START
12. ..........................
```

и соответствия PBM

Code:

```
1.  *** Rvm zone bounds: 0042B412 - 0042B53B ***
2.  in   rvm_30 rvm_08 -      rvm_14 rvm_34 rvm_04 rvm_30 rvm_20 rvm_3C rvm_1C rvm_08 -      rvm_0C
3.  out  rvm_1C rvm_20 rvm_30 rvm_38 rvm_18 rvm_34 rvm_10 rvm_0C rvm_04 rvm_3C rvm_00 rvm_28 rvm_2C
```

здесь ещё необходимо отметить, что зоны могут строиться не прямо вокруг своей точки: SKIP - JMP - SKIP и SKIP - LABEL - SKIP, но и по пути перехода - SKIP - JMP -> LABEL - SKIP, но тогда такая зона может одновременно принадлежать двум зонам разных сеансов ВМ, т.к. между точкой перехода и меткой возможен выход/вход из/в ВМ.
3. Определение размещения предопределенных регистров CPU в виртуализованных инструкциях по PBM.
Например, инструкция LODS имеет следующие предопределенные регистры - EAX на выходе, ESI на входе и на выходе, и EFL на входе.
Следовательно, в зоне виртуализации таких инструкций известно соответствие предопределенных регистров CPU PBM.
Пример виртуализованного кода

Code:

```
1.  // LODSB
2.  // lodsb
3.
4.  /*      rvm_Eax = [rvm_esi]
5.          *svm1 = and ~rvm_efl, 0x00000400
6.          +svm2 = shr svm1, 9
7.          svm3 = svm2 + 0xFFFFFFFF
8.          rvm_Esi = svm3 + rvm_esi  */
```

и соответствия регистров

Code:

```
1.  *** Subzone bounds: 0042B0FC - 0042B15D ***
2.  *** Regs Associations (zone 0) ***
3.  regs  esi     efl     ebx   ebp   ebx   edi   eax     edx   ecx
4.  in    rvm_04  rvm_18  -     -     -     -     -       -     -
```

```
5.  out  rvm_30 -      -      -      -      -      rvm_08 -      -
6.  false -      -      +      -      -      -      -      -      -
```

Как видно из всех примеров все зоны имеют границы в промкоде и соответственно в его "текстовом" представлении.
Основная же задача состоит в том, чтобы каждый rvm_ в тексте заменить эквивалентным ему регистром CPU, на основе информации соответствия в зонах.

Создано: 18 августа 2010 16:19:59

Последние три шага сделаны, вот исходный код куска защищенной функции, полученный декомпилятором:

Code:

```
1.  ++++++++++++++++++++++++++++++++++++
2.   Section asm
3.  ++++++++++++++++++++++++++++++++++++
4.
5.  0042B0DD: lea     esp, dword ptr [esp + 4]
6.  0042B0E0: mov     eax, 1
7.  0042B0EB: cmp     eax, 2
8.  0042B408: jnl     0x004011DB
9.  0042B4CF: mov     eax, 3
10. 0042B679: mov     dword ptr [ebp + 0xFFFFFFE0], eax
```

Создано: 3 ноября 2010 16:11:00

Разобрана последняя опция ВмПротекта - **Проверка целостности кода**

Этот алгоритм запускается перед выходом из ВМ для вызова процедуры CALL. Сначала в стек помещаются три адреса (всё это делается под ВМ):
1. Адрес возврата в следующую секцию ВМ после отработки процедуры.
2. Адрес обфускированного кода в котором осуществляется вызов процедуры.
3. Адрес обфускированного кода в котором выполняется дешифрация ресистров CPU.
Делается это следующим образом (сокрытие констант не учитываю):
- адрес 1 непосредственно записывается в стек
- производится восстановление начального кода по этому адресу
Code:
01ABEDDC: rvm_xx = 0x01A1CB84 // адрес
01ABEE59: [rvm_xx] = 0x68
01ABEF49: [rvm_xx + 1] = 0x32A8E094
01ABF012: [rvm_xx + 5] = 0xE8
01ABF116: [rvm_xx + 6] = 0xFFFF9500

- адрес 2 непосредственно записывается в стек
- этот адрес модифицируется, если CRC неправильная или не модифицируется, если CRC правильная, следующим образом:
а). берется случайное число
Code:
01AB6B09: rdtsc
01AB6B8F: xor eax, 0xA221
01AB6CAE: div eax, 0x0056
01AB6D45: mul edx, 0x0009

б). берутся параметры проверки кода и считается CRC
Code:
01AB6E2E: svm_134 = 0x01A16F39 // адрес структур
01AB6E3F: rvm_0C = eax + svm_134 // адрес кодированного адреса проверяемого кода, eax из пункта а)
01AB6EFF: svm_143 = rvm_0C + 4 // адрес длины кода
01AB702E: svm_155 = xor [rvm_0C], 0x03447E8F // декодирование адреса проверяемого кода
01AB7050: svm_157 = crc svm_155, [svm_143] // параметры: адрес кода и его длина (байт)

в). к CRC прибавляется константа из структуры
Code:
01AB709B: svm_161 = rvm_0C + 5 // адрес константы
01AB70A6: svm_162 = svm_157 + [svm_161]

г). полученный результат svm_162 (при правильной CRC он равен нулю) прибавляется к адресу 2.
- повторяются все теже действия с адресом 2 для адреса 3 (значение адреса структур другое и конечно же случайное число другое)
- перед выходом из ВМ хешируются все регистры CPU
- осуществляется выход из ВМ на адрес 3
- выполнение кода по адресу 3 - расхеширование регистров
- выполнение кода по адресу 2 - вызов требуемой функции
- выполнение кода по адресу 1 - вход в новую секцию ВМ

Создано: 7 декабря 2010 10:33:50

Теперь пройдусь по стадиям обработки и анализа кода **VmSweeper**:
1. **Analyze all VM references.** Процесс должен пройти 100% весь код, восстановить весь импорт, включая ИАТ и вызовы/обращения к АПИ функциям. Должное внимание следует уделить заданию диапазонов адресов кода и вм, от их правильности зависит вся дальнейшая работа. Эти диапазоны могут охватывать более одной секции файла и могут перекрываться. По диапазону кода пояснения не требуются - это рабочий диапазон кода программы. Требования к диапазону вм - охватить им все места размещения вм. Критические части ВМ, которые обязательно должны находиться внутри этого диапазона:
- начальный адрес ленты пикода
- таблица обработчиков примитивов
А так как вмпротект занимает собой все свободные места в проге, то диапазон вм может быть от начала кода до конца файла. Но следует учитывать, чем шире этот диапазон, тем больше мусора (ложных точек входа) будет найдено, но ничего пропущено не будет, если дипазон вм задать уже (по сегментам вм), то возможен пропуск как точек входа, так и импорта и в дальнейшем можем получить сообщения о нераспознаваемости частей вм. Если во время работы этого процесса получаем сообщение об ошибке, то первым делом ищем в **последних** логах/трейсах строку с #ERROR#.
2. **Decode VM.** Этот процесс состоит из нескольких частей:
- Определение типа вм. Должно быть 100% распознавание - в случае ошибки смотрим файлы Regs_xxx и Stack_xxx, и если знаем теорию, а без неё здесь делать нечего, то понимаем причину ошибки.
- Определение цикла вм и её параметров - начального адреса ленты пикода и таблицы обработчиков. В случае ошибки - смотри выше...
- Декодирование цикла вм и всех обработчиков. ФПУ обработчики пока не реализованы (уже сделано). Любая ошибка 100% локализуется по логам и трейсам.

- Создание промкода. В статусе имеем "VMS: Creating intermediate code... " - всё должно работать и код должен создасться рабочим. За исключением некоторых случаев - хардсвичи двух типов не обрабатываются (уже сделано), если они виртуализованы вм, также возможны ошибки при определении сложных ветвлений внутри функции.
- Декомпиляция промкода. В статусе имеем "VMS: Decompiling intermediate code... ". Сложностей здесь быть не должно, если промкод построен правильно.
- Получение исходного кода. Здесь только в 5-10% случаев можно получить полностью восстановленный код. Все стадии этого процесса есть в логах - иногда небольшая ручная доработка позволяет получить исходный код.

Важное замечание: Если после поиска всех ссылок вы хотите декомпилировать функции, то нужно выполнить перезапуск Ольки, сохранив предварительно таблицу ссылок в текстовом файле, в противном случае возможны сбои в работе, т.к. под восстановленным импортом может оказаться часть вм!!!

Создано: 7 декабря 2010 19:10:59

Автор Ultras
С разрешения Vamit, публикую небольшой step-by-step tutorial по успешной декомпиляции свипером защищенных ф-ий.

Основная цель: минимизация ошибок на этапе декомпиляции. Прежде чем писать багрепорт автору, внимательно пробегитесь по описанным ниже шагам, может быть вы что-то упустили в очередной раз
Подготовка:
- Перед началом работы обновите dbghelp.dll до самой свежей версии.
- Для целей декомпиляции рекомендуется иметь отдельную сборку OllyDbg с минимальным нужным набором плагинов (например только Phantom и VMSweeper). (Примечание автора: Phantom уже не нужен, т.к. VmSweeper имеет свою защиту от обнаружения его VmProtect)
- Убедитесь что у вас установлена последняя версия свипера (версию проверяем в окне About плагина).

Итак приступим:
1. Загружаем программу в OllyDbg и добираемся до OEP или останавливаемся близко к ней, главный момент здесь – это данные программы не должны быть еще инициализированы!
2. Запускаем "Analyse all VM references". Указываем диапазон секции кода и VM и жмем OK.
3. После анализа свипер предложит восстановить импорт: вводим начальный адрес расположения переходников и предположительно конечный. Если конечный адрес задать неверно то свипер сообщит о невозможности расположения некоторых восстановленных переходников -> надо будет увеличивать конечную границу и повторять анализ.
4. После анализа и восстановления импорта сохраняем лог найденных референсов (Reference VM Window) в файл и **перезапускаем** программу, чтобы восстановленный импорт не нарушил логику ВМ! (Примечание автора: Перезапускать правильнее Ольку).
5. Ну вот мы опять на OEP или рядом. По логу референсов находим интересующие нас ф-ии (помечены как Postponed) и начинаем декомпиляцию: для этого желательно брякнуться на нужной ф-ии, либо принудительно установить на нее EIP (New origin here). Жмем F1. Запустился декомпилятор…
6. На перезапуски программы отвечаем OK. Перед продолжением декомпиляции, после перезапусков, важно находиться всегда в одной и той же точке программы, например OEP: брякнулись на OEP, нажали Shift+F1, свипер отработал и предложил перезапуск, мы опять на OEP, опять Shift+F1 и тд.
Если в процессе создания свипером промежуточного кода вылетел эксепшен, то рестартуем программу и продолжаем по Shift+F1. (Примечание автора: Уже добавлена новая опция при перезапуске – Автоматическая декомпиляция, используйте её)
7. Если всё сделали правильно то доберетесь до конца декомпиляции.

Небольшие замечания по ходу декомпиляции:
- На предложенный свипером реанализ кода – всегда жмем OK.
- В некоторых случаях может потребоваться указание в качестве начала границы VM - начало секции кода (например в Delphi программах). Для этого сначала рекомендуется провести анализ с узким диапазоном границ VM, сохранить окно референсов, затем расширить границы сегмента VM и повторить анализ. При таком анализе отложенных (Postponed) ф-ий будет много, поэтому работу по декомпиляции ведем по нашему сохраненному вначале окну референсов.
- Перед анализом ВМ ссылок рекомендуется закрыть окошко CPU, а после можно открывать. Это слегка ускорит процесс.

Удачной декомпиляции!

Создано: 1 февраля 2011 14:30:29

Обзор создаваемых декомпилятором файлов.
При декомпиляции защищенной функции по F1 создаются два основных файла с именем адреса декодируемой области, на котором нажали F1, и расширениями log и trc.

Трейс файл содержит следующую последовательную информацию:
1. Асм код с трассировкой значений от точки старта до начала цикла вм, по этому куску распознается тип вм.
2. Сегменты кода и вм заданные юзером или автоматически определенные Свипером.
3. Асм код с трассировкой значений цикла вм до вызова обработчика.
4. Трассировка обработчиков вм от **Start Virtual Machine** до **Stop Virtual Machine**. В конце этой секции находится карта шагов по реструктуризации промкода - Pimap: step 1 и т.д.
5. Созданный промкод. Это последовательность обработчиков примитивов представленная чистым асм кодом. Далее этот код декомпилируется и если декомпиляция успешна (создается файл логов), то на точке старта создается инструкция jmp промкод. Можно осуществить переход в него и походить по нему Олькой.
6. Дальнейшие секции файла от **Adjust block 000** до **Optimized text a12 final** отображают шаги декомпиляции промкода и интереса для юзера не представляют.

Лог файл содержит следующую последовательную информацию:
1. От **Section 000** до **Section 005** разбивка промкода на логические блоки и их группировка.
2. **Section a00** - представление значимых инструкций промкода
3. **Section a01 code** - аналогично, только адреса регистров вм заменены аббревиатурой rvm_xx, где xx - смещение регистра вм в блоке регистров.
4. Список зон изменения соответствий регистров вм и CPU.
5. Шаги декомпиляции зон промкода. От **Section a02 (zone x)** до **Section a07 remove rvm2 (zone x)**, где x - номер зоны промкода. Что производит Свипер с кодом на этих шагах можно понять из названия секции и сравнения кода в ней с кодом предыдущей секции.
6. **Section a08 svm** - секция объединяющая все декомпилированные зоны промкода с последующим поглощением стековых переменных вм - svm_xx
7. Список зон изменения соответствий регистров вм и CPU с учетом девиртуализации инструкций.
8. **Section a11 reg** - первоначальная замена регистров вм регистрами CPU.
9. Список зон изменения соответствий регистров вм и CPU с учетом выполненной замены.
10. **Section a11 resolve reg** - окончательная замена регистров вм регистрами CPU.
11. **Section a12 final** - текстовое представление декомпилированного промкода.
12. **Section asm** - окончательно восстановленный асм код функции.

Информация с блока 8 и до конца часто может быть недостоверной, т.к. пока 100% восстановить соответствие регистров вм и CPU удается далеко не всегда. До блока 8 информация достоверна, но может быть избыточной ввиду того, что некоторые инструкции кода не девиртуализовались по разным причинам. Поэтому для полного ручного восстановления кода брать информацию до блока 8.

Так же ещё создаются порядка десятка map файлов - карты ресурсов вм и промкода, содержимое их должно быть понятно из названия.

Создано: 26 февраля 2011 16:52:44

Привожу "правильные" опции ассемблера Ольки при которых создавался и тестировался VmSweeper (выделены опции которые у некоторых пользователей действительно приводили к ошибкам анализа/декомпиляции кода).
[Settings]
IDEAL disassembling mode=0
Disassemble in lowercase=0
Separate arguments with TAB=0
Extra space between arguments=0
Show default segments=1
Always show memory size=1
NEAR jump modifiers=0
Show local module names=1
Show symbolic addresses=0
Use short form of string commands=0
Use RET instead of RETN=0
SSE size decoding mode=0
Size sensitive mnemonics=1
Top of FPU stack=1
Decode registers for any IP=0
Automatically select register type=0
Decode SSE registers=0

Создано: 17 марта 2012 13:20:22

Фишка новой версии вмпрота - переход на новый цикл вм со всеми его атрибутами (направление чтения пикода, хеширование пикода, набор примитивов) в процессе работы вм. Появились два новых примитива, а может быть и больше, которые меняют цикл вм на лету... Теперь нужно обучить этому и декомпилятор...

Создано: 21 марта 2012 10:08:55

Теперь нужно обучить этому и декомпилятор...
Выполнено..., как оказалось при исследовании декомпилированного кода это "инлайн" вызов другой завиртуализованной функи из текущего сеанса без выхода из вм. Поясню - контекстное окружение вм остается прежним (вирт. регистры и стек), но меняется сама реализация вм (набор примитивов, направление и хеширование ленты пикода). Таким образом в вызываемую функцию точки входа в программе не найти, потому что её нет и само тело этой функции растворено в виртуализованной вызывающей функции. Возможно это начало новой технологии, которая может превратить всю программу в одну виртуализованную функцию...

Создано: 23 марта 2012 09:24:37

Не всё так печально, как написано выше, любая функция может вызываться всевозможными косвенными методами, поэтому, "тело" любой завиртуализованной функи в программе находится на своем месте и имеет точку входа в вм и может быть декомпилировано отдельно. Сложность здесь в отладке, т.к. функа явно не вызывается, и в определении соответствия реальной функции - "инлайн" виртуализованной функции, но это можно сделать простым сравнением карт пикода отдельной функи и "инлайн" комбайна.
Начало новой технологии откладывается на неопределенный срок..., но попортить жизнь многим реверсерам своими наворотами вмпрот способен...

Приложение №1
«Чистые» тела обработчиков примитивов

AddByte
```
// pop    ax
// add    byte ptr [esp], al
// pushfd
```

AddShort
```
// pop    ax
// add    word ptr [esp], ax
// pushfd
```

AddLong
```
// pop    eax
// add    dword ptr [esp], eax
// pushfd
```

ImulByte
```
// pop    ax
// pop    cx
// imul   cl
// push   ax
// pushfd
```

ImulShort
```
// pop    ax
// pop    cx
// imul   cx
// push   ax
// push   dx
// pushfd
```

ImulLong
```
// pop    eax
// pop    ecx
// imul   ecx
// push   eax
```

```
// push   edx
// pushfd
```

MulByte
```
// pop    ax
// pop    cx
// mul    cl
// push   ax
// pushfd
```

MulShort
```
// pop    ax
// pop    cx
// mul    cx
// push   ax
// push   dx
// pushfd
```

MulLong
```
// pop    eax
// pop    ecx
// mul    ecx
// push   eax
// push   edx
// pushfd
```

IdivByte
```
// pop    ax
// pop    cx
// idiv   cl
// push   ax
// pushfd
```

IdivShort
```
// pop    dx
// pop    ax
// pop    cx
// idiv   cx
// push   ax
// push   dx
// pushfd
```

IdivLong
```
// pop    edx
// pop    eax
// pop    ecx
// idiv   ecx
// push   eax
// push   edx
// pushfd
```

DivByte
```
// pop    ax
// pop    cx
// div    cl
// push   ax
// pushfd
```

DivShort
```
// pop    dx
// pop    ax
// pop    cx
// div    cx
// push   ax
// push   dx
// pushfd
```

DivLong
```
// pop    edx
// pop    eax
// pop    ecx
// div    ecx
// push   eax
// push   edx
// pushfd
```

NorByte
```
// pop    ax
// pop    dx
// not    al
// not    dl
// push   ax
// and    byte ptr [esp], dl
// pushfd
```

NorShort
```
// pop    ax
// pop    dx
// not    ax
// not    dx
// push   ax
// and    word ptr [esp], dx
// pushfd
```

NorLong
```
// pop    eax
// pop    edx
// not    eax
// not    edx
// push   eax
// and    dword ptr [esp], edx
// pushfd
```

RclByte
```
// pop    ax
// pop    cx
// push   ax
// rcl    byte ptr [esp], cl
// pushfd
```

RclShort
```
// pop    ax
// pop    cx
// push   ax
// rcl    word ptr [esp], cl
// pushfd
```

RclLong
```
// pop    eax
// pop    cx
// push   eax
// rcl    dword ptr [esp], cl
// pushfd
```

RcrByte
```
// pop    ax
// pop    cx
// push   ax
// rcr    byte ptr [esp], cl
// pushfd
```

RcrShort
```
// pop    ax
// pop    cx
// push   ax
// rcr    word ptr [esp], cl
// pushfd
```

RcrLong
```
// pop    eax
// pop    cx
// push   eax
// rcr    dword ptr [esp], cl
// pushfd
```

ShlByte
```
// pop    ax
// pop    cx
// push   ax
// shl    byte ptr [esp], cl
// pushfd
```

ShlShort
```
// pop    ax
// pop    cx
// push   ax
// shl    word ptr [esp], cl
// pushfd
```

ShlLong
```
// pop    eax
// pop    cx
// push   eax
// shl    dword ptr [esp], cl
// pushfd
```

ShrByte
```
// pop    ax
// pop    cx
// push   ax
```

```
// shr    byte ptr [esp], cl
// pushfd
```

ShrShort
```
// pop    ax
// pop    cx
// push   ax
// shr    word ptr [esp], cl
// pushfd
```

ShrLong
```
// pop    eax
// pop    cx
// push   eax
// shr    dword ptr [esp], cl
// pushfd
```

Shld
```
// pop    eax
// pop    edx
// pop    cx
// push   eax
// shld   dword ptr [esp], edx, cl
// pushfd
```

Shrd
```
// pop    eax
// pop    edx
// pop    cx
// push   eax
// shrd   dword ptr [esp], edx, cl
// pushfd
```

PushImmByteShort
```
// push   const byte
```

PushImmByteLong
```
// mov    al, const byte
// movsx  eax, al
// push   eax
```

PushImmShortShort
```
// push   const word
```

PushImmShortLong
```
// mov    ax, const word
// movsx  eax, ax
// push   eax
```

PushImmLong
```
// push   const dword
```

PushDsMemByte
```
// pop    edx
// mov    al, byte ptr ds:[edx]
// push   ax
```

PushSsMemByte
```
// pop    edx
// mov    al, byte ptr ss:[edx]
// push   ax
```

PushEsMemByte
```
// pop    edx
// mov    al, byte ptr es:[edx]
// push   ax
```

PushDsMemShort
```
// pop    edx
// push   word ptr ds:[edx]
```

PushSsMemShort
```
// pop    edx
// push   word ptr ss:[edx]
```

PushEsMemShort
```
// pop    edx
// push   word ptr es:[edx]
```

PushDsMemLong
```
// pop    eax
// push   dword ptr ds:[eax]
```

PushSsMemLong
```
// pop    eax
```

```
// push   dword ptr ss:[eax]
```

PushEsMemLong
```
// pop    eax
// push   dword ptr es:[eax]
```

PushFsMemLong
```
// pop    eax
// push   dword ptr fs:[eax]
```

PushSs
```
// mov     ax, ss
// push    ax
```

PushRvmByte
```
// mov    al, byte ptr [reg_const byte]
// push   ax
```

PushRvmShort
```
// push   word ptr [reg_const byte]
```

PushRvmLong
```
// push   dword ptr [reg_const byte || reg_cmd]
```

PushSvmShort
```
// push   sp
```

PushSvmLong
```
// push   esp
```

PopDsMemByte
```
// pop    eax
// pop    dx
// mov    byte ptr ds:[eax], dl
```

PopSsMemByte
```
// pop    eax
// pop    dx
// mov    byte ptr ss:[eax], dl
```

PopEsMemByte
```
// pop    eax
// pop    dx
// mov    byte ptr es:[eax], dl
```

PopDsMemShort
```
// pop    eax
// pop    dx
// mov    word ptr ds:[eax], dx
```

PopSsMemShort
```
// pop    eax
// pop    dx
// mov    word ptr ss:[eax], dx
```

PopEsMemShort
```
// pop    eax
// pop    dx
// mov    word ptr es:[eax], dx
```

PopDsMemLong
```
// pop    eax
// pop    edx
// mov    dword ptr ds:[eax], edx
```

PopSsMemLong
```
// pop    eax
// pop    edx
// mov    dword ptr ss:[eax], edx
```

PopEsMemLong
```
// pop    eax
// pop    edx
// mov    dword ptr es:[eax], edx
```

PopFsMemLong
```
// pop    eax
// pop    edx
// mov    dword ptr fs:[eax], edx
```

PopRvmByte
```
// pop    dx
// mov    byte ptr [reg_const byte], dl
```

PopRvmShort

```
// pop    word ptr [reg_const byte]
```

PopRvmLong
```
// pop    dword ptr [reg_const byte || reg_cmd]
```

PopSvmShort
```
// mov    sp, word ptr [esp]
```

PopSvmLong
```
// mov    esp, dword ptr [esp]
```

Popfd
```
// popfd
```

Jmp
```
// pop    eax
// add    eax, dword ptr [esp]
// jmp    eax
```

Exit
```
// clear stack (m_nContentSize - m_nLoadIndex)
// load registers (m_StoreReg[m_nLoadIndex])
// call   [esp]
```

Crc1
```
// pop    edx                                          (адрес памяти)
// pop    ecx                                          (длина проверки)
// calculate CRC in eax
// push   eax                                          (CRC)
```

Crc2
```
// pop    edx                                          (адрес памяти)
// pop    ecx                                          (длина проверки)
// calculate CRC in eax
// push   eax                                          (CRC)
```

Rdtsc
```
// rdtsc
// push   eax
// push   edx
```

Cpuid
```
// pop     eax                                         (входной параметр)
// cpuid
// process eax
// push    eax
// push    ebx
// push    ecx
// push    edx
```

Call
```
// т.к. аргументы находятся на своих местах в стеке, то их не трогаем
// countArg = byte const
// pop     eax        - адрес перехода на функцию
// call    eax
// аргументы из стека удаляет вызываемая функция, но это действие эмулируется, т.к. реально функция не вызывается,
// push    eax        - возвращаемое значение
```

SimpleCall
```
// простой вызов функции без аргументов и возвращаемого значения
// pop     eax        - адрес перехода на функцию
// call    eax
```

Приложение №2

Шаблоны виртуализации ассемблерных инструкций

```
// Условные обозначения:
// svm.. - стековая переменная
// rvm.. - регистровая переменная
// rvm_fl.. - флаги промежуточных выражений
// rvm_eax - предопределенный регистр
// rvm_Reg - регистр результата инструкции
// rvm_Efl - флаги результата инструкции
// * - первая значимая инструкция
// + - вторая значимая инструкция
```

// Арифметические инструкции

ADD

```
// add     rvm1, rvm2

*svm1 = add rvm1, rvm2
+rvm_Efl = iEFL
rvm_Reg = svm1
```

ADD ESP

```
// add     esp, rvm1

svm1 = __$esp + 4
*svm2 = add rvm1, svm1
+rvm_Efl = iEFL
rvm_Esp = svm2
```

XADD

```
// xadd rvm1, rvm2

*svm1 = add rvm1, rvm2
+rvm_Efl = iEFL
rvm_Reg1 = svm1
rvm_Reg2 = rvm1

*svm1 = add rvm1, rvm2
+rvm_Efl = iEFL
rvm_Reg1 = svm1
rvm_Reg2 = rvm2
```

SUB

```
// sub     rvm1, rvm2

svm1 = not  rvm1, rvm1
*svm2 = add rvm2, svm1
+rvm_fl1(3) = iEFL
svm3 = not  svm2, svm2
rvm_fl2(4) = iEFL
rvm_Reg = svm3
svm4 = and rvm_fl1(3), 0x00000815
svm5 = and rvm_fl2(4), 0xFFFFF7EA
rvm_Efl = svm4 + svm5
```

SUB without flags

```
// sub     rvm1, rvm2

svm1 = not  rvm1, rvm1
*svm2 = svm1 + rvm2
svm_Reg = not  svm2, svm2
```

SUB ESP

```
// sub     esp, rvm1

svm1 = __$esp + 8
svm2 = __$esp + C
svm3 = and  ~svm2, ~svm1
*svm4 = add rvm1, svm3
+rvm_fl1(2) = iEFL
svm5 = not  svm4, svm4
rvm_fl2(3) = iEFL
rvm_Esp = svm5
svm6 = and rvm_fl1(2), 0x00000815
svm7 = and rvm_fl2(3), 0xFFFFF7EA
rvm_Efl = svm6 + svm7
```

SUB ESP without flags

```
// sub     esp, rvm1

svm1 = __$esp + 8
svm2 = __$esp + C
svm3 = and  ~svm2, ~svm1
svm4 = svm3 + rvm1
*rvm_Esp = not  svm4, svm4
```

CMP

```
// cmp     rvm1, rvm2

svm1 = not  rvm1, rvm1
*svm2 = add rvm2, svm1
+rvm_fl1(3) = iEFL
svm3 = not  svm2, svm2
rvm_fl2(4) = iEFL
rvm3 = svm3                 // изъятие ненужного промежуточного результата вычитания
svm4 = and rvm_fl1(3), 0x00000815
svm5 = and rvm_fl2(4), 0xFFFFF7EA
rvm_Efl = svm4 + svm5
```

// Шаблон отсутствует (эквивалентен шаблону SUB), определение инструкции CMP производится следующим шагом при дальнейшем анализе результирующего регистра

ADC

```
// adc     rvm1, rvm2

svm1 = and rvm_efl, 0x0001
*svm2 = add rvm2, svm1
+rvm_fl1(3) = iEFL
svm3 = add svm2, rvm1
rvm_fl2(4) = iEFL
rvm_Reg = svm3
svm4 = and rvm_fl1(3), 1
rvm_Efl = or  svm4, rvm_fl2(4)
```

SBB

```
// sbb     rvm1, rvm2

svm1 = and rvm_efl, 0x0001
svm2 = rvm2 + svm1
svm3 = not  svm2, svm2
svm4 = svm3 + 1
*svm5 = add svm4, rvm1
+rvm_fl(3) = iEFL
rvm_Reg = svm5
rvm_Efl = xor  rvm_fl(3), 0x00000011
```

INC

```
// inc     rvm1

*svm1 = add rvm1, 1
+rvm_fl(2) = iEFL
rvm_Reg = svm1
svm2 = and rvm_efl, 1
svm3 = and rvm_fl(2), 0xFFFFFFFE
rvm_Efl = svm2 + svm3
```

DEC

```
// dec rvm1

*svm1 = add rvm1, 0xFFFFFFFF
+rvm_fl(2) = iEFL
rvm_Reg = svm1
svm2 = and rvm_elf, 1
svm3 = and rvm_fl(2), 0xFFFFFFFE
rvm3 = svm2 + svm3
rvm_Efl = xor  rvm3, 0x00000010
```

NEG

```
// neg     rvm1

*svm1 = add rvm1, 0xFFFFFFFF
+rvm_fl1(2) = iEFL
svm2 = not  svm1, svm1
rvm_fl2(3) = iEFL
rvm_Reg = svm2
svm3 = and rvm_fl1(2), 0x00000815
svm4 = and rvm_fl2(3), 0xFFFFF7EA
rvm_Efl = svm3 + svm4
```

// Инструкции умножения/деления

DIV
// div rvm1 (DWORD or WORD)

*iEAX = div rvm1, rvm_eax, rvm_edx
rvm_Efl = iEFL
rvm_Edx = iEDX
rvm_Eax = iEAX

DIV
// div rvm1 (DWORD or WORD) without flags

*iEAX = div rvm1, rvm_eax, rvm_edx
rvm_Edx = iEDX
rvm_Eax = iEAX

DIV
// div rvm1 (BYTE)

*iEAX = div rvm1, rvm_ax
rvm_Efl = iEFL
rvm_Eax = iEAX

DIV
// div rvm1 (BYTE) without flags

*iEAX = div rvm1, rvm_ax
rvm_Eax = iEAX

IDIV
// idiv rvm1 (DWORD or WORD)

*iEAX = idiv rvm1, rvm_eax, rvm_edx
rvm_Efl = iEFL
rvm_Edx = iEDX
rvm_Eax = iEAX

IDIV
// idiv rvm1 (DWORD or WORD) without flags

*iEAX = idiv rvm1, rvm_eax, rvm_edx
rvm_Edx = iEDX
rvm_Eax = iEAX

IDIV
// idiv rvm1 (BYTE)

*iEAX = idiv rvm1, rvm_ax
rvm_Efl = iEFL
rvm_Eax = iEAX

IDIV
// idiv rvm1 (BYTE) without flags

*iEAX = idiv rvm1, rvm_ax
rvm_Eax = iEAX

MUL
// mul rvm1 (DWORD or WORD)

*iEAX = mul rvm1, rvm_eax
rvm_fl(2) = iEFL
rvm_Edx = iEDX
rvm_Eax = iEAX
svm1 = and rvm_efl, 0x000000D4
svm2 = and rvm_fl(2), 0xFFFFFF2B
rvm_Efl = svm1 + svm2

MUL
// mul rvm1 (DWORD or WORD) without flags

*iEAX = mul rvm1, rvm_eax
rvm_Edx = iEDX
rvm_Eax = iEAX

MUL
// mul rvm1 (BYTE)

*iEAX = mul rvm1, rvm_al
rvm_fl(2) = iEFL
rvm_Eax = iEAX
svm1 = and rvm_efl, 0x000000D4
svm2 = and rvm_fl(2), 0xFFFFFF2B

```
rvm_Efl = svm1 + svm2
```

MUL
```
// mul     rvm1                         (BYTE) without flags

*iEAX = mul rvm1, rvm_al
rvm_Eax = iEAX
```

IMUL
```
// imul    rvm2, rvm1                           (DWORD or WORD) 2 operands

*iEAX = imul rvm1, rvm2
rvm_fl(3) = iEFL
rvm_R1 = iEDX
rvm_R2 = iEAX
svm1 = and rvm_efl, 0x000000D4
svm2 = and rvm_fl(3), 0xFFFFFF2B
rvm_Efl = svm1 + svm2
```

IMUL
```
// imul    rvm2, rvm1                           (DWORD or WORD) 2 operands without flags

*iEAX = imul rvm1, rvm2
rvm_R1 = iEDX
rvm_R2 = iEAX
```
// IMUL
```
// imul    rvm5, rvm2, rvm1  (DWORD or WORD) 3 operands

*iEAX = imul rvm1, rvm2
rvm_fl(3) = iEFL
rvm_R1 = iEDX
rvm_R2 = iEAX
svm1 = and rvm_efl, 0x000000D4
svm2 = and rvm_fl(3), 0xFFFFFF2B
rvm_Efl = svm1 + svm2
```

IMUL
```
// imul    rvm5, rvm2, rvm1  (DWORD or WORD) 3 operands without flags

*iEAX = imul rvm1, rvm2
rvm_R1 = iEDX
rvm_R2 = iEAX
```

IMUL
```
// imul    rvm1                         (BYTE)

*iEAX = imul rvm1, rvm_al
rvm_fl(2) = iEFL
rvm_Eax = iEAX
svm1 = and rvm_efl, 0x000000D4
svm2 = and rvm_fl(2), 0xFFFFFF2B
rvm_Efl = svm1 + svm2
```

IMUL
```
// imul    rvm1                         (BYTE) without flags

*iEAX = imul rvm1, rvm_al
rvm_Eax = iEAX
```

// Логические инструкции

AND
```
// and     rvm1, rvm2

*svm1 = and rvm1, rvm2
+rvm_Efl = iEFL
rvm_Reg = svm1
```

TEST
```
// test rvm1, rvm2

*svm1 = and rvm1, rvm2
+rvm_Efl = iEFL
rvm3 = svm1                          // изъятие ненужного промежуточного результата AND
```
// Шаблон отсутствует (эквивалентен шаблону AND), определение инструкции TEST производится следующим шагом при дальнейшем анализе результирующего регистра

NOT
```
// not     rvm1

rvm_Reg = not  rvm1, rvm1            // Обрабатывается без шаблона
```

OR
```
// or      rvm1, rvm2
```

```
*svm1 = or  rvm1, rvm2
+rvm_Efl = iEFL
rvm_Reg = svm1
```

XOR

```
// xor      rvm1, rvm2

*svm1 = xor  rvm2, rvm1
+rvm_Efl = iEFL
rvm_Reg = svm1
```

// Инструкции сдвига

RCL

```
// rcl      rvm1, rvm2

svm1 = and rvm_efl, 0x0001
svm2 = shl svm1, 8
svm3 = svm2 + rvm2
*svm4 = rcl rvm1, svm3
+rvm_fl(3) = iEFL
rvm_Reg = svm4
svm5 = and rvm_efl, 0x000000D4
svm6 = and rvm_fl(3), 0xFFFFFF2B
rvm_Efl = svm5 + svm6
```

RCR

```
// rcr      rvm1, rvm2

svm1 = and rvm_efl, 0x0001
svm2 = shl svm1, 8
svm3 = svm2 + rvm2
*svm4 = rcr rvm1, svm3
+rvm_fl(3) = iEFL
rvm_Reg = svm4
svm5 = and rvm_efl, 0x000000D4
svm6 = and rvm_fl(3), 0xFFFFFF2B
rvm_Efl = svm5 + svm6
```

ROL

```
// rol      rvm1, rvm2                  (DWORD)

*svm1 = shld rvm1, rvm1, rvm2
+rvm_fl(3) = iEFL
rvm_Reg = svm1
svm2 = and rvm_efl, 0x000000D4
svm3 = and rvm_fl(3), 0xFFFFFF2B
rvm_Efl = svm2 + svm3
```

ROL

```
// rol      rvm1, rvm2                  (BYTE)

svm1 = __$esp + 6
svm2 = shl [svm1], 8
svm3 = rvm1 + svm2
*svm4 = shld svm3, svm3, rvm2
+rvm_fl(3) = iEFL
rvm_Reg = svm4
rvm4 = svm4
svm5 = and rvm_efl, 0x000000D4
svm6 = and rvm_fl(3), 0xFFFFFF2B
rvm_Efl = svm5 + svm6
```

ROL

```
// rol      rvm1, rvm2                  (WORD)

*svm1 = shld rvm1, rvm1, rvm2
+rvm_fl(3) = iEFL
rvm_Reg = svm1
rvm4 = svm1
svm2 = and rvm_efl, 0x000000D4
svm3 = and rvm_fl(3), 0xFFFFFF2B
rvm_Efl = svm2 + svm3
```

ROR

```
// ror      rvm1, rvm2                  (DWORD)

*svm1 = shrd rvm1, rvm1, rvm2
+rvm_fl(3) = iEFL
rvm_Reg = svm1
svm2 = and rvm_efl, 0x000000D4
svm3 = and rvm_fl(3), 0xFFFFFF2B
rvm_Efl = svm2 + svm3
```

ROR
```
// ror      rvm1, rvm2                  (BYTE)

svm1 = __$esp + 2
svm2 = shl [svm1], 8
svm3 = rvm1 + svm2
*svm4 = shrd svm3, svm3, rvm2
+rvm_fl(3) = iEFL
rvm_Reg = svm4
rvm4 = svm4
svm5 = and rvm_efl, 0x000000D4
svm6 = and rvm_fl(3), 0xFFFFFF2B
rvm_Efl = svm5 + svm6
```

ROR
```
// ror      rvm1, rvm2                  (WORD)

*svm1 = shrd rvm1, rvm1, rvm2
+rvm_fl(3) = iEFL
rvm_Reg = svm1
rvm4 = svm1
svm2 = and rvm_efl, 0x000000D4
svm3 = and rvm_fl(3), 0xFFFFFF2B
rvm_Efl = svm2 + svm3
```

SAR
```
// sar      rvm1, rvm2                  (DWORD)

svm1 = shr rvm1, 0x1F
svm2 = not  svm1, svm1
svm3 = svm2 + 1
*svm4 = shrd rvm1, svm3, rvm2
+rvm_Efl = iEFL
rvm_Reg = svm4
```

SAR
```
// sar      rvm1, rvm2                  (BYTE)

svm1 = shr rvm1, 7
svm2 = not svm1, svm1
svm3 = svm2 + 1
svm4 = __$esp + 2
svm5 = shl [svm4], 8
svm6 = svm5 + rvm1
*svm7 = shrd svm3, svm3, rvm2
+rvm_Efl = iEFL
rvm_Reg = svm7
rvm3 = svm7
```
SAR
```
// sar      rvm1, rvm2                  (WORD)

svm1 = shr rvm1, 0xF
svm2 = not  svm1, svm1
svm3 = svm2 + 1
*svm4 = shrd svm3, svm3, rvm2
+rvm_Efl = iEFL
rvm_Reg = svm4
rvm3 = svm4
```

SHL
```
// shl      rvm1, rvm2

*svm1 = shl rvm1, rvm2
+rvm_Efl = iEFL
rvm_Reg = svm1
```

SHR
```
// shr      rvm1, rvm2

*svm1 = shr rvm1, rvm2
+rvm_Efl = iEFL
rvm_Reg = svm1
```

SHLD
```
// shld     rvm1, rvm2, rvm3

*svm1 = shld rvm1, rvm2, rvm3
+rvm_Efl = iEFL
rvm_Reg = svm1
```

SHRD
```
// shrd rvm1, rvm2, rvm3

*svm1 = shrd rvm1, rvm2, rvm3
+rvm_Efl = iEFL
```

```
rvm_Reg = svm1
```

// Инструкции преобразования размера операнда

CBW
```
// cbw

*svm1 = shr rvm_al, 7
svm2 = not  svm1, svm1
rvm_al+1 = svm2 + 1
```

CWDE
```
// cwde

*svm1 = shr rvm_ax, 0x0F
svm2 = not  svm1, svm1
svm3 = svm2 + 1
rvm_eax = rvm_ax | svm3
```

CWD
```
// cwd

*svm1 = shr rvm_ax, 0x0F
svm2 = not  svm1, svm1
rvm_dx = svm2 + 1
```

CDQ
```
// cdq

*svm1 = shr rvm_eax, 0x1F
svm2 = not  svm1, svm1
rvm_edx = svm2 + 1
```

MOVSX
```
// movsx rvm1, rvm2                    (BYTE -> WORD)

*svm1 = shr rvm2, 7
svm2 = not  svm1, svm1
svm3 = svm2 + 1
svm4 = shl svm3, 8
rvm1 = svm4 + rvm2
```

MOVSX
```
// movsx rvm1, rvm2                    (BYTE -> DWORD)

*svm1 = shr rvm2, 7
svm2 = not  svm1, svm1
svm3 = svm2 + 1
svm4 = __$esp + n * 2
svm5 = shl [svm4], 8
svm6 = svm5 + rvm2
rvm1 = svm6 | svm3
```

MOVSX
```
// movsx rvm1, rvm2                    (WORD -> DWORD)

*svm1 = shr rvm2, 0x0F
svm2 = not  svm1, svm1
svm3 = svm2 + 1
rvm1 = rvm2 | svm3

// Обрабатываются без шаблонов по размерам операндов
```

MOVZX
```
// movzx rvm1, rvm2                    (BYTE -> WORD)
// rvm1 = rvm2
```

MOVZX
```
// movzx rvm1, rvm2                    (BYTE -> DWORD)
// rvm1 = rvm2
```

MOVZX
```
// movzx rvm1, rvm2                    (WORD -> DWORD)
// rvm1 = rvm2
```

// Инструкции перестановок операндов

BSWAP

```
// bswap rvm1

rvm2 = rvm1                                  // LWORD rvm1
*svm1 = shl rvm1 | rvm1, 8  // сдвиг двух HWORD
svm2 = shl rvm2 | rvm2, 8    // сдвиг двух LWORD
rvm3 = svm2                                  // изъятие незначимого LWORD
svm3 = __$esp + 8                    // адрес значимого HWORD
rvm_Reg = [svm3] | svm2
rvm4 = svm1                                  // изъятие использованного промежуточного результата
```

BSWAP (вариант без обфускации)

```
// bswap rvm1

rvm2 = rvm1                                  // LWORD rvm1
*svm1 = shl rvm2 | rvm2, 8  // сдвиг двух HWORD
svm2 = __$esp + 8                    // адрес значимого HWORD
rvm_Reg = [svm2] | svm1

// Обрабатывается без шаблона
```

XCHG

```
// xchg rvm1, rvm2
// rvm_Reg2 = rvm1
// rvm_Reg1 = rvm2
```

// Битовые инструкции

BT

```
// bt       rvm1, rvm2                  (DWORD)

*svm1 = shr rvm1, rvm2
svm2 = and svm1, 0x0001
svm3 = and rvm_efl, 0xFFFE
rvm_Efl = svm2 + svm3
rvm3 = svm1                          // изъятие использованного промежуточного результата
```

BT

```
// bt       rvm1, rvm2                  (WORD)

*svm1 = shr rvm1, rvm2
svm2 = and svm1, 0x0001
svm3 = and rvm_efl, 0xFFFE
rvm_Efl = svm2 + svm3
```

BTC

```
// btc      rvm1, rvm2                  (DWORD)

*svm1 = shr rvm1, rvm2
svm2 = and svm1, 0x0001
svm3 = and rvm_efl, 0xFFFE
rvm_Efl = svm2 + svm3
rvm3 = svm1                          // изъятие использованного промежуточного результата
svm4 = shl 1, rvm2
svm5 = and ~rvm1, ~svm4
svm6 = not rvm1, rvm1
svm7 = shl 1, rvm2
svm8 = not svm7, svm7
svm9 = and ~svm8, ~svm6
rvm_Reg = and ~svm9, ~svm5          // должно быть rvm_Reg = xor rvm1, svm4
```

BTC

```
// btc      rvm1, rvm2                  (WORD)

*svm1 = shr rvm1, rvm2
svm2 = and svm1, 0x0001
svm3 = and rvm_efl, 0xFFFE
rvm_Efl = svm2 + svm3
svm4 = shl 1, rvm2
svm5 = and ~rvm1, ~svm4
svm6 = not rvm1, rvm1
svm7 = shl 1, rvm2
svm8 = not svm7, svm7
svm9 = and ~svm8, ~svm6
rvm_Reg = and ~svm9, ~svm5          // должно быть rvm_Reg = xor rvm1, svm4
```

BTR

```
// btr      rvm1, rvm2                  (DWORD)

*svm1 = shr rvm1, rvm2
svm2 = and svm1, 0x0001
svm3 = and rvm_efl, 0xFFFE
rvm_Efl = svm2 + svm3
```

```
rvm3 = svm1                         // изъятие использованного промежуточного результата
svm4 = shl 1, rvm2
svm5 = and ~rvm1, ~rvm1
rvm_Reg = and ~svm5, ~svm4          // должно быть rvm_Reg = and rvm1, ~svm4
```

BTR

```
// btr      rvm1, rvm2                 (WORD)

*svm1 = shr rvm1, rvm2
svm2 = and svm1, 0x0001
svm3 = and rvm_efl, 0xFFFE
rvm_Efl = svm2 + svm3
svm4 = shl 1, rvm2
svm5 = and ~rvm1, ~rvm1
rvm_Reg = and ~svm5, ~svm4          // должно быть rvm_Reg = and rvm1, ~svm4
```

BTS

```
// bts      rvm1, rvm2                 (DWORD)

*svm1 = shr rvm1, rvm2
svm2 = and svm1, 0x0001
svm3 = and rvm_efl, 0xFFFE
rvm_Efl = svm2 + svm3
rvm3 = svm1                         // изъятие использованного промежуточного результата
svm4 = shl 1, rvm2
rvm_Res = or  svm4, rvm1
```

BTS

```
// bts      rvm1, rvm2                 (WORD)

*svm1 = shr rvm1, rvm2
svm2 = and svm1, 0x0001
svm3 = and rvm_efl, 0xFFFE
rvm_Efl = svm2 + svm3
svm4 = shl 1, rvm2
rvm_Res = or  svm4, rvm1
```

// Инструкции установки/сброса флагов

CLC

```
// clc
*rvm_Efl = and rvm_efl, 0xFFFFFFFE
```

CLD

```
// cld

*rvm_Efl = and rvm_efl, 0xFFFFFBFF
```

CMC

```
// cmc

*rvm_Efl = xor  rvm_efl, 1
```

STC

```
// stc

*rvm_Efl = or  rvm_efl, 1
```

STD

```
// std

*rvm_Efl = or  rvm_efl, 0x00000400
```

SAHF

```
// sahf

*svm1 = and rvm_al+1, 0xD5
svm2 = and rvm_efl, 0xFF2A
rvm_Efl = svm1 + svm2
```

// Строковые инструкции

LODSB

```
// lodsb

rvm_Eax = [rvm_esi]
*svm1 = and ~rvm_efl, 0x00000400
+svm2 = shr svm1, 9
svm3 = svm2 + 0xFFFFFFFF
rvm_Esi = svm3 + rvm_esi
```

LODSW

```
// lodsw

rvm_Eax = [rvm_esi]
*svm1 = and ~rvm_efl, 0x00000400
```

```
+svm2 = shr svm1, 8
svm3 = svm2 + 0xFFFFFFFE
rvm_Esi = svm3 + rvm_esi
```

LODSD

```
// lodsd

rvm_Eax = [rvm_esi]
*svm1 = and ~rvm_efl, 0x00000400
+svm2 = shr svm1, 7
svm3 = svm2 + 0xFFFFFFFC
rvm_Esi = svm3 + rvm_esi
```

STOSB

```
// stosb

[rvm_edi] = rvm_eax
*svm1 = and ~rvm_efl, 0x00000400
+svm2 = shr svm1, 9
svm3 = svm2 + 0xFFFFFFFF
rvm_Edi = svm3 + rvm_edi
```

STOSW

```
// stosw

[rvm_edi] = rvm_eax
*svm1 = and ~rvm_efl, 0x00000400
+svm2 = shr svm1, 8
svm3 = svm2 + 0xFFFFFFFE
rvm_Edi = svm3 + rvm_edi
```

STOSD

```
// stosd

[rvm_edi] = rvm_eax
*svm1 = and ~rvm_efl, 0x00000400
+svm2 = shr svm1, 7
svm3 = svm2 + 0xFFFFFFFC
rvm_Edi = svm3 + rvm_edi
```

MOVSB

```
// movsb

[rvm_edi] = [rvm_esi]
*svm1 = and ~rvm_efl, 0x00000400
+svm2 = shr svm1, 9
svm3 = svm2 + 0xFFFFFFFF
rvm_Esi = svm3 + rvm_esi
rvm_Edi = svm3 + rvm_edi
```

MOVSW

```
// movsw

[rvm_edi] = [rvm_esi]
*svm1 = and ~rvm_efl, 0x00000400
+svm2 = shr svm1, 8
svm3 = svm2 + 0xFFFFFFFE
rvm_Esi = svm3 + rvm_esi
rvm_Edi = svm3 + rvm_edi
```

MOVSD

```
// movsd

[rvm_edi] = [rvm_esi]
*svm1 = and ~rvm_efl, 0x00000400
+svm2 = shr svm1, 7
svm3 = svm2 + 0xFFFFFFFC
rvm_Esi = svm3 + rvm_esi
rvm_Edi = svm3 + rvm_edi
```

CMPSB

```
// cmpsb

svm1 = not  [rvm_esi], [rvm_esi]
svm2 = add svm1, [rvm_edi]
rvm_fl1(1) = iEFL
svm3 = not  svm2, svm2
rvm_fl2(2) = iEFL
rvm3 = svm3                  // изъятие ненужного промежуточного результата вычитания
svm4 = and rvm_fl1(1), 0x00000815
svm5 = and rvm_fl2(2), 0xFFFFF7EA
rvm_Efl = svm4 + svm5
*svm6 = and ~rvm_Efl, 0x00000400
+svm7 = shr svm6, 9
svm8 = svm7 + 0xFFFFFFFF
```

```
rvm_Esi = svm8 + rvm_esi
rvm_Edi = svm8 + rvm_edi
```

CMPSW

```
// cmpsw

svm1 = not  [rvm_esi], [rvm_esi]
svm2 = add svm1, [rvm_edi]
rvm_fl1(1) = iEFL
svm3 = not  svm2, svm2
rvm_fl2(2) = iEFL
rvm3 = svm3                 // изъятие ненужного промежуточного результата вычитания
svm4 = and rvm_fl1(1), 0x00000815
svm5 = and rvm_fl2(2), 0xFFFFF7EA
rvm_Efl = svm4 + svm5
*svm6 = and ~rvm_Efl, 0x00000400
+svm7 = shr svm6, 8
svm8 = svm7 + 0xFFFFFFFE
rvm_Esi = svm8 + rvm_esi
rvm_Edi = svm8 + rvm_edi
```

CMPSD

```
// cmpsd

svm1 = not  [rvm_esi], [rvm_esi]
svm2 = add svm1, [rvm_edi]
rvm_fl1(1) = iEFL
svm3 = not  svm2, svm2
rvm_fl2(2) = iEFL
rvm3 = svm3                 // изъятие ненужного промежуточного результата вычитания
svm4 = and rvm_fl1(1), 0x00000815
svm5 = and rvm_fl2(2), 0xFFFFF7EA
rvm_Efl = svm4 + svm5
*svm6 = and ~rvm_Efl, 0x00000400
+svm7 = shr svm6, 7
svm8 = svm7 + 0xFFFFFFFC
rvm_Esi = svm8 + rvm_esi
rvm_Edi = svm8 + rvm_edi
```

SCASB

```
// scasb

rvm1 = cmp rvm_eax, [rvm_edi]
*svm1 = and ~rvm_Efl, 0x00000400
+svm2 = shr svm1, 9
svm3 = svm2 + 0xFFFFFFFF
rvm_Edi = svm3 + rvm_edi
```

SCASW

```
// scasw

rvm1 = cmp rvm_eax, [rvm_edi]
*svm1 = and ~rvm_Efl, 0x00000400
+svm2 = shr svm1, 8
svm3 = svm2 + 0xFFFFFFFE
rvm_Edi = svm3 + rvm_edi
```

SCASD

```
// scasd

rvm1 = cmp rvm_eax, [rvm_edi]
*svm1 = and ~rvm_Efl, 0x00000400
+svm2 = shr svm1, 7
svm3 = svm2 + 0xFFFFFFFC
rvm_Edi = svm3 + rvm_edi
```

// Инструкции тестирования флагов

SETNG or SETLE

```
// setle rvm10

rvm1 = test rvm_efl, 0x00000080
rvm2 = test rvm_efl, 0x00000800
rvm5 = xor  rvm3, rvm4
rvm6 = test rvm_efl, 0x00000040
svm1 = not  rvm7, rvm7
rvm8 = or svm1, rvm5
*svm2 = and rvm8, 0x00000040
+rvm9 = shr svm2, 6
```

SETG or SETNLE

```
// setg     rvm11

rvm1 = test rvm_efl, 0x00000080
rvm2 = test rvm_efl, 0x00000800
rvm5 = xor  rvm3, rvm4
```

```
rvm6 = not  rvm5, rvm5
rvm7 = test rvm_efl, 0x00000040
rvm10 = and rvm8, rvm9
*svm1 = and rvm10, 0x00000040
+rvm11 = shr svm1, 6
```

SETNL or SETGE

```
// setge rvm7

rvm1 = test rvm_efl, 0x00000080
rvm2 = test rvm_efl, 0x00000800
rvm5 = xor  rvm3, rvm4
rvm6 = not  rvm5, rvm5
*svm1 = and rvm6, 0x00000040
+rvm7 = shr svm1, 6
```

SETL or SETNGE

```
// setl     rvm6

rvm1 = test rvm_efl, 0x00000080
rvm2 = test rvm_efl, 0x00000800
rvm5 = xor  rvm3, rvm4
*svm1 = and rvm5, 0x00000040
+rvm6 = shr svm1, 6
```

SETNA or SETBE

```
// setbe rvm4

rvm1 = test rvm_efl, 0x00000041
rvm3 = not  rvm2, rvm2
*svm1 = and rvm3, 0x00000040
+rvm4 = shr svm1, 6
```

SETA or SETNBE

```
// seta     rvm3

rvm1 = test rvm_efl, 0x00000041
*svm1 = and rvm2, 0x00000040
+rvm3 = shr svm1, 6
```

SETNE or SETNZ

```
// setne rvm1

*svm1 = and ~rvm_efl, 0x00000040
+rvm1 = shr svm1, 6
```

SETE or SETZ

```
// sete     rvm1

*svm1 = and rvm_efl, 0x00000040
+rvm1 = shr svm1, 6
```

SETNC or SETAE or SETNB

```
// setae rvm1

*rvm1 = and ~rvm_efl, 1
```

SETC or SETB or SETNAE

```
// setb     rvm1

*rvm1 = and rvm_efl, 1
```

SETNO

```
// setno rvm1

*svm1 = and ~rvm_efl, 0x00000800
+rvm1 = shr svm1, 0x0B
```

SETO

```
// seto rvm1

*svm1 = and rvm_efl, 0x00000800
+rvm1 = shr svm1, 0x0B
```

SETNP or SETPO

```
// setnp rvm1

*svm1 = and ~rvm_efl, 4
+rvm1 = shr svm1, 2
```

SETP or SETPE

```
// setp rvm1

*svm1 = and rvm_efl, 4
```

```
+rvm1 = shr svm1, 2
```

SETNS
```
// setns rvm1

*svm1 = and ~rvm_efl, 0x00000080
+rvm1 = shr svm1, 7
```

SETS
```
// sets rvm1

*svm1 = and rvm_efl, 0x00000080
+rvm1 = shr svm1, 7
```

// Инструкции условных переходов

JNG or JLE
```
// jng      xx

rvm1 = test rvm_efl, 0x00000080       // jmp to 1 address or dir to 2 address
rvm2 = test rvm_efl, 0x00000800
rvm5 = xor  rvm3, rvm4
rvm6 = test rvm_efl, 0x00000040
svm1 = not  rvm7, rvm7
rvm8 = or svm1, rvm5
*svm2 = and rvm8, 0x00000040
+svm3 = shr svm2, 4
svm4 = __$esp + svm3
rvm9 = [svm4]

rvm1 = test rvm_efl, 0x00000080       // jmp to 2 address or dir to 1 address
rvm3 = test rvm_efl, 0x00000800
rvm5 = xor rvm3, rvm4
rvm6 = not rvm5, rvm5
rvm7 = test rvm_efl, 0x00000040
rvm10 = and rvm8, rvm9
*svm1 = and rvm10, 0x00000040
+svm2 = shr svm1, 4
svm3 = __$esp + svm2
rvm11 = [svm3]
```

JG or JNLE
```
// jg       xx

rvm1 = test rvm_efl, 0x00000080       // jmp to 1 address or dir to 2 address
rvm3 = test rvm_efl, 0x00000800
rvm5 = xor rvm3, rvm4
rvm6 = not rvm5, rvm5
rvm7 = test rvm_efl, 0x00000040
rvm10 = and rvm8, rvm9
*svm1 = and rvm10, 0x00000040
+svm2 = shr svm1, 4
svm3 = __$esp + svm2
rvm11 = [svm3]

rvm1 = test rvm_efl, 0x00000080       // jmp to 2 address or dir to 1 address
rvm2 = test rvm_efl, 0x00000800
rvm5 = xor  rvm3, rvm4
rvm6 = test rvm_efl, 0x00000040
svm1 = not  rvm7, rvm7
rvm8 = or svm1, rvm5
*svm2 = and rvm8, 0x00000040
+svm3 = shr svm2, 4
svm4 = __$esp + svm3
rvm9 = [svm4]
```

JNL or JGE
```
// jnl      xx

rvm1 = test rvm_efl, 0x00000080       // jmp to 1 address or dir to 2 address
rvm2 = test rvm_efl, 0x00000800
rvm5 = xor  rvm3, rvm4
rvm6 = not  rvm5, rvm5
*svm1 = and rvm6, 0x00000040
+svm2 = shr svm1, 4
svm3 = __$esp + svm2
rvm7 = [svm3]

rvm1 = test rvm_efl, 0x00000080       // jmp to 2 address or dir to 1 address
rvm2 = test rvm_efl, 0x00000800
rvm5 = xor  rvm3, rvm4
*svm1 = and rvm5, 0x00000040
+svm2 = shr svm1, 4
svm3 = __$esp + svm2
rvm6 = [svm3]
```

JL or JNGE

```
// jl       xx

rvm1 = test rvm_efl, 0x00000080      // jmp to 1 address or dir to 2 address
rvm2 = test rvm_efl, 0x00000800
rvm5 = xor  rvm3, rvm4
*svm1 = and rvm5, 0x00000040
+svm2 = shr svm1, 4
svm3 = __$esp + svm2
rvm6 = [svm3]

rvm1 = test rvm_efl, 0x00000080      // jmp to 2 address or dir to 1 address
rvm2 = test rvm_efl, 0x00000800
rvm5 = xor  rvm3, rvm4
rvm6 = not  rvm5, rvm5
*svm1 = and rvm6, 0x00000040
+svm2 = shr svm1, 4
svm3 = __$esp + svm2
rvm7 = [svm3]
```

/ / JNA or JBE

```
// jna     xx

rvm1 = test rvm_efl, 0x00000041      // jmp to 1 address or dir to 2 address
rvm3 = not  rvm2, rvm2
*svm1 = and rvm3, 0x00000040
+svm2 = shr svm1, 4
svm3 = __$esp + svm2
rvm4 = [svm3]

rvm1 = test rvm_efl, 0x00000041      // jmp to 2 address or dir to 1 address
*svm1 = and rvm2, 0x00000040
+svm2 = shr svm1, 4
svm3 = __$esp + svm2
rvm3 = [svm3]
```

JA or JNBE

```
// ja      xx

rvm1 = test rvm_efl, 0x00000041      // jmp to 1 address or dir to 2 address
*svm1 = and rvm2, 0x00000040
+svm2 = shr svm1, 4
svm3 = __$esp + svm2
rvm3 = [svm3]

rvm1 = test rvm_efl, 0x00000041      // jmp to 2 address or dir to 1 address
rvm3 = not  rvm2, rvm2
*svm1 = and rvm3, 0x00000040
+svm2 = shr svm1, 4
svm3 = __$esp + svm2
rvm4 = [svm3]
```

JNE or JNZ

```
// jne     xx

*svm1 = and ~rvm_efl, 0x00000040   // jmp to 1 address or dir to 2 address
+svm2 = shr svm1, 4
svm3 = __$esp + svm2
rvm1 = [svm3]

*svm1 = and rvm_efl, 0x00000040    // jmp to 2 address or dir to 1 address
+svm2 = shr svm1, 4
svm3 = __$esp + svm2
rvm1 = [svm3]
```

JE or JZ

```
// je      xx

*svm1 = and rvm_efl, 0x00000040    // jmp to 1 address or dir to 2 address
+svm2 = shr svm1, 4
svm3 = __$esp + svm2
rvm1 = [svm3]

*svm1 = and ~rvm_efl, 0x00000040   // jmp to 2 address or dir to 1 address
+svm2 = shr svm1, 4
svm3 = __$esp + svm2
rvm1 = [svm3]
```

JNC or JAE or JNB

```
// jnc     xx

*svm1 = and ~rvm_efl, 1                      // jmp to 1 address or dir to 2 address
+svm2 = shl svm1, 2
```

```
svm3 = __$esp + svm2
rvm1 = [svm3]

*svm1 = and rvm_efl, 1                          // jmp to 2 address or dir to 1 address
+svm2 = shl svm1, 2
svm3 = __$esp + svm2
rvm1 = [svm3]
```

JC or JB or JNAE

```
// jc       xx

*svm1 = and rvm_efl, 1                          // jmp to 1 address or dir to 2 address
+svm2 = shl svm1, 2
svm3 = __$esp + svm2
rvm1 = [svm3]

*svm1 = and ~rvm_efl, 1                         // jmp to 2 address or dir to 1 address
+svm2 = shl svm1, 2
svm3 = __$esp + svm2
rvm1 = [svm3]
```

JNO

```
// jno      xx

*svm1 = and ~rvm_efl, 0x00000800    // jmp to 1 address or dir to 2 address
+svm2 = shr svm1, 9
svm3 = __$esp + svm2
rvm1 = [svm3]

*svm1 = and rvm_efl, 0x00000800     // jmp to 2 address or dir to 1 address
+svm2 = shr svm1, 9
svm3 = __$esp + svm2
rvm1 = [svm3]
```

JO

```
// jo       xx

*svm1 = and rvm_efl, 0x00000800     // jmp to 1 address or dir to 2 address
+svm2 = shr svm1, 9
svm3 = __$esp + svm2
rvm1 = [svm3]

*svm1 = and ~rvm_efl, 0x00000800    // jmp to 2 address or dir to 1 address
+svm2 = shr svm1, 9
svm3 = __$esp + svm2
rvm1 = [svm3]
```

JNP or JPO

```
// jnp      xx

*svm1 = and ~rvm_efl, 4                         // jmp to 1 address or dir to 2 address
svm2 = __$esp + svm1
rvm1 = [svm2]

*svm1 = and rvm_efl, 4                          // jmp to 2 address or dir to 1 address
svm2 = __$esp + svm1
rvm1 = [svm2]
```

JP or JPE

```
// jp       xx

*svm1 = and rvm_efl, 4                          // jmp to 1 address or dir to 2 address
svm2 = __$esp + svm1
rvm1 = [svm2]

*svm1 = and ~rvm_efl, 4                         // jmp to 2 address or dir to 1 address
svm2 = __$esp + svm1
rvm1 = [svm2]
```

JNS

```
// jns      xx

*svm1 = and ~rvm_efl, 0x00000080    // jmp to 1 address or dir to 2 address
+svm2 = shr svm1, 5
svm3 = __$esp + svm2
rvm1 = [svm3]

*svm1 = and rvm_efl, 0x00000080     // jmp to 2 address or dir to 1 address
+svm2 = shr svm1, 5
svm3 = __$esp + svm2
rvm1 = [svm3]
```

JS
```
// js       xx

*svm1 = and rvm_efl, 0x00000080      // jmp to 1 address or dir to 2 address
+svm2 = shr svm1, 5
svm3 = __$esp + svm2
rvm1 = [svm3]

*svm1 = and ~rvm_efl, 0x00000080    // jmp to 2 address or dir to 1 address
+svm2 = shr svm1, 5
svm3 = __$esp + svm2
rvm1 = [svm3]
```

// Прочие инструкции

LAHF
```
// lahf

*rvm_al+1 = rvm_efl
```

XLAT
```
// xlat

svm1 = rvm_al + rvm_ebx
rvm_Al = [svm1]
```

RDTSC
```
// rdtsc

*rvm_Edx = rdtscH
+rvm_Eax = rdtscL
```

CPUID
```
// cpuid

*cpuidA = cpuid rvm_eax
rvm_Edx = cpuidD
rvm_Ecx = cpuidC
rvm_Ebx = cpuidB
rvm_Eax = cpuidA
```
// Restore entry VM code

```
rvm1 = const
*[rvm1] = 0x68
svm1 = rvm1 + 1
[svm1] = const
svm2 = rvm1 + 5
[svm2] = 0xE8
svm3 = rvm1 + 6
[svm3] = const
```

// Get randomize offset to CRC struct

```
rvm_Edx = rdtscH
rvm_Eax = rdtscL
svm1 = xor rvm1, const
rvm2 = svm1 | rdtscL
*iEAX = div 0x0056, rvm2, 0x0000
iEAX = mul iEDX, 0x009
rvm2 = iEAX
```