О методах деобфускации программ

Ш.Ф. Курмангалеев, К.Ю. Долгорукова,
В.В. Савченко, А.Р. Нурмухаметов, Р.А. Матевосян, В.П. Корчагин
kursh@ispras.ru, unerkannt@ispras.ru
sinmipt@ispras.ru, oleshka@ispras.ru, hripsime@ispras.ru, korchagin@ispras.ru

Аннотация. Целью работы является разработка программного обеспечения, проводящего деобфусцирующие преобразования. Основная область применения – это анализ запутанного кода вредоносного программного обеспечения. Потребность в подобного рода продуктах возникла в связи с ростом популярности методик запутывания кода для сокрытия алгоритмов работы. Основным инструментом аналитика является дизассемблер, осуществляющий преобразование бинарного кода в читаемый человеком текст, но не проводящий его верификацию и упрощение. Ранее для «чистки» запутанного кода хватало удаления бесполезного кода по шаблонам, но применяемые методики запутывания усложняются, и для распутывания требуются средства, использующие более прогрессивные методы анализа и упрощения кода. В связи со схожестью проблем, стоящих перед оптимизирующим компилятором и деобфускатором, было опробовано использование компиляторной инфраструктуры LLVM в качестве ядра деобфускатора. Основным различием является то, что оптимизатор компилятора работает в условиях полного знания о программе, в то время как деобфускатор обладает неполной информацией, извлеченной непосредственно из участка анализируемого кода. По результатам испытаний, в связи с высокой зашумленностью выходного кода, было принято решение разработать свою инфраструктуру, которая позволяет добиться более чистого выходного кода. Тем не менее, применение LLVM или аналогичной разработки остается одним из перспективных направлений при разработке деобфусцирующего программного обеспечения.

Ключевые слова: анализ бинарного кода, деобфускация, обфускация, LLVM

1. Введение

Запутывание (обфускация) кода широко применяется при защите программного обеспечения от обратного проектирования, причем как

легального, так и вредоносного. Поскольку запутывание вызывает существенное замедление программы, очень часто запутывается только небольшая ее часть, содержащая важный алгоритм. Распутывание (деобфускация) применяется в области минимизации бинарного кода программ, что является важным для встраиваемых систем, так и для анализа вредоносного кода.

Современные вирусы наделены определенной способностью к маскировке выполняемых действий. Вирусы пытаются избежать обнаружения и воспрепятствовать анализу своего кода за счет применения обфусцирующих преобразований, включая изменение графа потока управления, эквивалентные замены команд, перестановки команд, изменение назначения адресов. Для создания алгоритма лечения вируса, требуется сначала детально разобрать алгоритм его работы. Без использования автоматических деобфускаторов скорость анализа весьма мала. С другой стороны, полностью автоматизировать этот процесс очень сложно, т.к. вредоносный код может иметь динамическое шифрование, различные варианты самомодификации. Как правило, аналитик вынужден работать с небольшим «окном» кода, на котором произведено запутывающее преобразование. Важной особенностью является то, что, как правило, одновременно применяется небольшое количество преобразований во избежание неожиданного и неправильного поведения кода. Создание эффективных средств лечения неизбежно требует изучения методов обфускации программ и разработки, специальных деобфусцирующих алгоритмов.

В разделе 2 описываются основные подходы к анализу программ. В разделе 3 описывается реализованный вариант деобфускатора, в разделе 4 описываются полученные экспериментальные результаты.

2. Алгоритмы распутывания

В данном разделе мы рассмотрим методы, которые применяются при анализе программ. Цель таких методов – выявление зависимостей между компонентами программы, что даёт возможность применить определённые оптимизирующие преобразования.

Методы анализа программ могут быть разделены на 4 группы [1]:

- Синтаксические**.** К этой группе относятся методы, основанные только на результатах лексического, синтаксического и статического семантического анализа программы.
- Статические [1][2]**.** К этой группе относятся методы анализа потоков управления и данных и методы, основанные на результатах анализа потоков управления и данных. Статические методы анализа работают

с программой, не используя информацию о работе программы на конкретных начальных данных.

- Динамические**.** Динамические методы анализа программ используют информацию, полученную в результате "наблюдения" за работой программы на конкретных входных данных.
- Статистические**.** Статистические методы используют информацию, собранную в результате значительного количества запусков программы на большом количестве наборов входных данных.

Важным этапом при проведении анализа кода является его нормализация. Нормализация кода – это процесс преобразования участка кода в каноническую форму, более пригодную для сравнения. Большинство из используемых преобразований обфускации ведет к увеличению размера кода. Иными словами, различные мутации фрагмента программы, могут рассматриваться как неоптимизированные версии своего прототипа, поскольку они содержат некоторые вычисления, присутствие которых имеет единственную цель – затруднение анализа. Нормализация кода направлена на устранение всех изменений, внесенных в ходе процесса обфускации.

Методы оптимизации, применяемые компиляторами, могут быть использованы для уменьшения количества «мусорного» кода. Обычно оптимизацию выполняет компилятор для уменьшения времени выполнения или для уменьшения размера кода или используемых данных.

Рассмотрим технику, предложенную в работе [5]: метапредставление инструкций. Все инструкции процессора можно разделить на три категории:

- условные и безусловные переходы;
- вызовы функций и возврат из функции;
- все остальные инструкции, оказывающие влияние на регистры, память и флаги управления.

Все инструкции третьей группы можно называть присваиваниями. Инструкции сравнения могут быть представлены в виде присваиваний, т.к. они обычно выполняют арифметическую операцию над своими операндами, а результат заносят в регистр управления (например, eflags для архитектуры IA-32). Для облегчения манипуляции объектным кодом будем использовать высокоуровневое представление, отражающее семантику машинных инструкций. В результате получаем так называемые «тройки» и «четверки», анализ и упрощение которых хорошо изучен в теории компиляторов. Стоит отметить, что даже простая инструкция декремента скрывает сложную семантику: аргумент уменьшается на единицу, и в зависимости от результата устанавливаются флаги.

Пример представлен в табл. 1.

Табл.1. Метапредставление инструкций

Машинная инструкция	Метапредставление
pop eax	r10 = [r11] r11 = r11 + 4
lea edi,[ebp]	r06 = r12
dec ebx	tmp = r08 r08 = r08 - 1 NF = r08@[31:31] ZF = [r08 = 0?1:0] CF = (˜(tmp@[31:31])

Поскольку большинство выражений содержит арифметические или логические операторы, иногда они могут быть упрощены в соответствии с обычными алгебраическими правилами [5]. Когда упрощение невозможно, переменные и константы могут быть переупорядочены, чтобы обеспечить возможность дальнейшего упрощения после этапа продвижения констант. В табл. 2 приведены примеры правил, которые могут быть использованы для выполнения упрощения и переупорядочивания (C обозначает константу, а t – переменную).

Табл.2. Примеры алгебраических правил

Оригинальное выражение	**Упрощенное выражение**
c1 + c2	Рассчитанная сумма констант
t1 − c1	−c1 + t1
t1 + c1	c1 + t1
0 + t1	t1
t1 + (t2 + t3)	(t1 + t2) + t3
(t1 + t2) * c1	(c1 * t1) + (c1 * t2)

3. Архитектура деобфускатора

Основной целью работы программы является получение листинга ассемблерного кода, по виду близкого к оригинальному (до обфускации) и пригодному для визуального анализа человеком. В процессе деобфускации происходят различные оптимизирующие преобразования, позволяющие добиться поставленной цели. Задача по своему характеру достаточна близка к задачам, встающим перед компиляторами на этапе оптимизации кода [3], и большинство подходов, выработанных при конструировании компиляторов, могут с успехом применяться в задачах деобфускации. В качестве вспомогательных библиотек для реализации используются boost [7][8] и graphviz [9].

Задачу деобфускации можно свести к построению оптимизирующего компилятора для языка ассемблера. Основное отличие от трансляторов ассемблера – наличие этапа оптимизации, характерного для высокоуровневых языков программирования. Архитектура системы будет иметь вид, представленный на рис. 1.

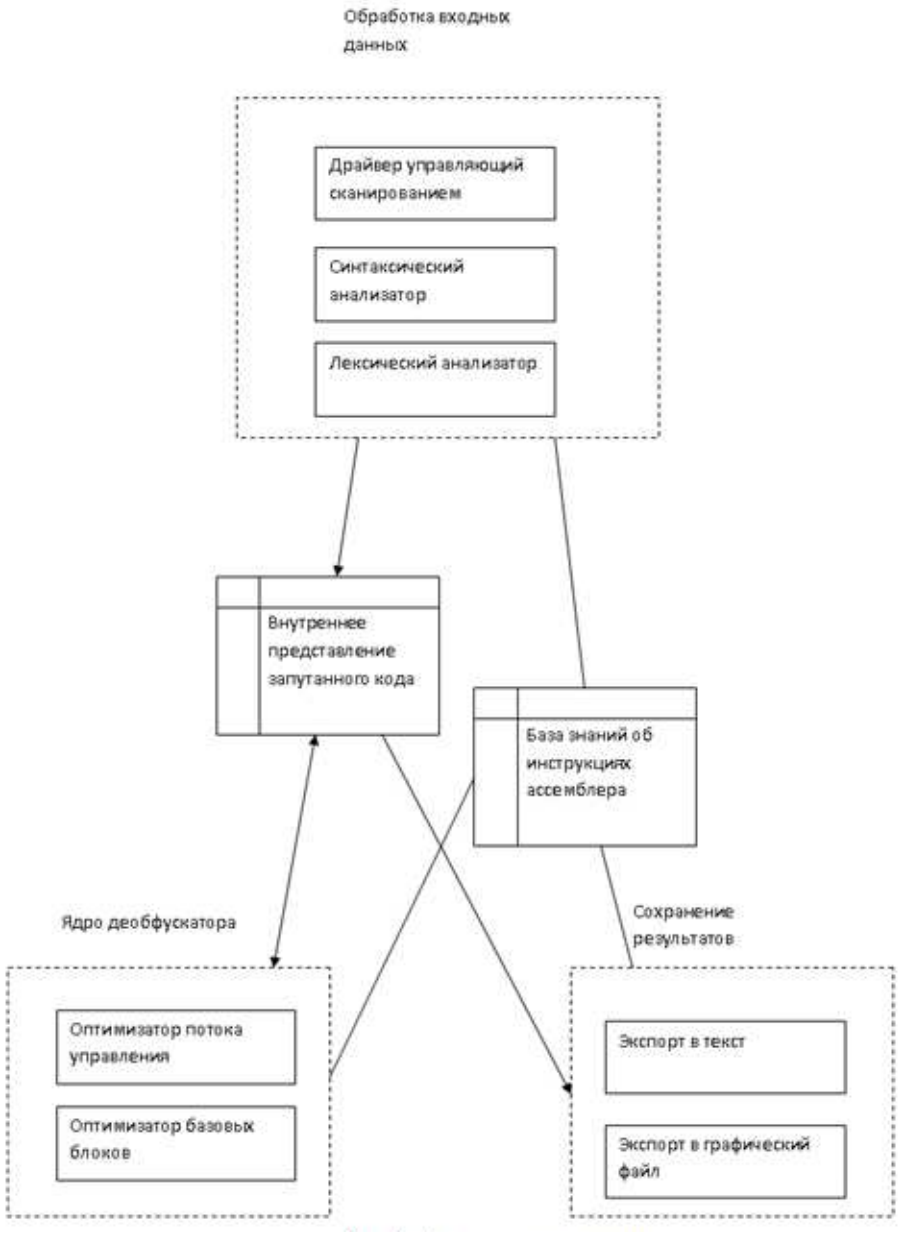

Рис.1. Архитектура системы

3.1 Используемые пакеты ПО

Graphviz — разработанный специалистами лаборатории AT&T пакет утилит для автоматической визуализации графов, заданных в виде описания на языке «dot».

Для построения лексического анализатора используется программа Flex, генерирующая лексический анализатор на основе описаний токенов, заданных с помощью регулярных выражений. В качестве генератора синтаксического анализатора используется Bison [6].

Low Level Virtual Machine (LLVM) [4] — инфраструктура для разработки компиляторов, содержащая:

- компилятор языка C++ в промежуточный байт-код, разработанный с целью обеспечения возможности оптимизации программы на всех этапах использования;
- набор RISC-подобных инструкций виртуального процессора, из которых строится байт-код;
- мощный оптимизатор байт-кода;
- окружение, предназначенное для исполнения байт-кода на различных платформах.

Первоначально в качестве ядра оптимизатора кода была использована LLVM. Но в связи с высоким уровнем «шума» в выходном коде, обусловленным потерями информации об именах регистров, было принято решение спроектировать свое ядро, проводящее оптимизации, специфические для обфусцированного кода.

3.2 Описание применяемых алгоритмов

3.2.1 Деобфускатор на базе LLVM

Для иллюстрации жизнеспособности гипотезы о применении техник оптимизации, используемых в компиляторах, был разработан транслятор некоторых машинных инструкций в язык промежуточного представления компиляторной инфраструктуры LLVM в форме SSA (форма с единственным присваиванием). Этот язык [4] компилируется в байт-код, который подвергается оптимизации с помощью набора инструментов, предоставляемого LLVM. В качестве теста использовался обфусцированный код выражений, приведенный в листинге 1.


```
EAX = EAX + 22;  ADD EAX, 22;
ECX = EDX + 1; MOV ECX, EDX; INC ECX;
```

Листинг 1. Код, использованный для обфускации

В качестве запутывающих преобразований использовались:

- внесение мертвого кода;
- внесение недостижимого кода, и, как следствие, изменение потока управления;
- внесение избыточного кода (избыточные вычисления).

После обфускации код примет вид, показанный в листинге 2.

```
ADD EAX,10
ADD EAX,10
INC EAX
MOV EAX,EAX
MOV ECX,EDX
MOV EDX,ECX
MOV EDX,ECX
ADD EAX,10
SUB EAX,15
INC EAX
INC EAX
NOP
NOP
NOP
NOP
INC EAX
INC EAX
INC EAX
JMP DEAD_CODE
NOP
NOP
NOP
NOP
MOV EAX,90909090
NOP
NOP
NOP
NOP
DEAD_CODE:
INC EAX
INC EAX
```

```
INC EAX
INC EAX
INC ECX
DEC EAX
DEC EAX
DEC EAX
```

Листинг 2. Обфусцированный код

Видно, что сложность восприятия кода заметно повысилась. После трансляции в форму SSA код примет вид, приведенный в листинге 3.

```
@eax = global i32 0;
@ebx = global i32 0;
@ecx = global i32 0;
@edx = global i32 0;
@esi = global i32 0;
@edi = global i32 0;
@eip = global i32 0;
@ebp = global i32 0;
@esp = global i32 0;
define void @main() nounwind {
 entry:
%0 = load i32* @eax
%1 = add i32 %0,10
store i32 %1, i32* @eax
%2 = load i32* @eax
%3= add i32 %2,10
store i32 %3, i32* @eax
%4 = load i32* @eax
%5 = add i32 %4,1
store i32  %5, i32* @eax
….
```



```
%38 = load i32* @eax
%39 = sub i32 %38,1
store i32  %39, i32* @eax
%40 = load i32* @eax
%41 = sub i32 %40,1
store i32  %41, i32* @eax
ret void }
```

Листинг 3. Код на промежуточном языке в форме SSA

После преобразования в форму SSA этот код передается на вход оптимизатора LLVM. В связи с особенностями архитектуры происходит потеря информации о регистрах, и единственный способ сохранить информацию – это использование имен регистров в качестве имен глобальных переменных, не подлежащих исключению. После проведения оптимизации получим код, продемонстрированный в листинге 4.

```
mov       EAX, DWORD PTR [_edx]
inc       EAX
mov       ECX, DWORD PTR [_eax]
mov       DWORD PTR [_ecx], EAX
add       ECX, 22
mov       DWORD PTR [_eax], ECX
```

Листинг 4. Код после проведения оптимизации с помощью LLVM

Код заметно упрощен, хотя и не соответствует в точности исходной форме, т.к. нет возможности повлиять на выбор регистров оптимизатором.

Поскольку для архитектуры Intel x86 существует возможность обращения к различным частям регистров, имеется регистр флагов, содержание которого может изменяться в результате исполнения команд, что приводит к дополнительному зашумлению листинга. Рассматривалась возможность внесения исправлений в архитектуру LLVM, но в связи с большим объемом работ и ограниченными сроками было принято решение отказаться от использования LLVM в качестве оптимизатора. Несмотря на указанные выше недостатки, использование LLVM в качестве ядра деобфускатора остается перспективным направлением.

3.2.2 Описание ядра деобфускатора

Для тестирования стойкости методов запутывания кода часто применяют алгоритмы оптимизации. Нами были реализованы следующие алгоритмы

оптимизации, позволяющие упрощать код, запутанный с использованием часто встречающихся методов:

- статическое построение и анализ потока управления;
- объединение базовых блоков;
- удаление недостижимого кода;
- продвижение констант;
- свертывание констант;
- удаление «мертвого» кода;
- алгебраическое упрощение, как часть алгоритма свертывания констант;
- статический слайсинг.

Дадим некоторые определения и опишем работу перечисленных алгоритмов.

Граф потока. Программа состоит из одного и более базовых блоков, являющихся узлами графа. Ребрами графа являются дуги переходов. Дуги переходов могут быть трех видов:

- нормальное движение по потоку управления Flow;
- переход по условию Cond;
- безусловный переход Uncond.

Статическое построение и анализ потока управления. Работа этого алгоритма начинается еще на стадии синтаксического анализа входного текста. Каждая распознанная инструкция передается в функцию, анализирующую переданную ей инструкцию и в зависимости от результата анализа либо добавляющую код в конец базового блока, либо создающую новый блок с достраиванием ребра типа Flow. После завершения этапа синтаксического анализадля всех блоков, на которые возможен переход, т.е. имеющих «метку», просматриваются все блоки графа и достраиваются ребра соответствующих переходов.

Объединение базовых блоков. Два следующих друг за другом базовых блока можно объединить, если они соединены ребром типа Flow или Uncond либо эквивалентом, состоящим из Flow и Cond, которые исходят из одной вершины. Обход вершин осуществляется обходом графа с помощью поиска в глубину. Обход проводится только из одной стартовой вершины. Для обхода графа используется модифицированная версия нерекурсивного алгоритма поиска в глубину с учетом приоритетов по типам ребер, где наибольший приоритет имеют ребра типа Flow и Uncond, а наименьший – Cond, и допускается удаление ребер из графа во время работы.

Удаление недостижимого код. Алгоритм основан на поиске в глубину. После завершения поиска из стартовой вершины все посещенные вершины имеют цвет, отличный от белого, и являются достижимыми из начала кода;

Вершины, цвет которых остался белым, могут быть удалены из графа, так как они не могут быть достигнуты из стартовой точки. Алгоритм состоит из двух частей – поиска в глубину и прохода по всем вершинам и удаления не достижимых.

Продвижение констант. Для всех инструкций в базовом блоке происходит сравнение с шаблоном mov reg, number. При совпадении значение регистра считается известным, и все его вхождения в качестве второго операнда заменяются на значение.

Свертывание констант. Для всех инструкций в базовом блоке происходит сравнение с шаблоном mov reg, number. При совпадении значение регистра считается известным. Далее продолжается просмотр инструкций, изменяющих значение регистра, и, если это возможно, происходит вычисление его нового значения. После вычисления нового значения происходит замена значения, инициализирующего регистр, и удаление проэмулированной инструкции. Поиск продолжается с инструкции, следующей за удаленной.

Удаление «мертвого» кода. Производится удаление бесполезных присваиваний для всех инструкций в базовом блоке. Выполняется сравнение с шаблоном mov reg, number. При совпадении поиск продолжается со следующей инструкции. Если встречается инструкция, перезаписывающая значение регистра без использования его предыдущего значения, то первоначально найденное присваивание можно удалить.

Алгебраическое упрощение, как часть алгоритма свертывания констант. Эта оптимизация заключается в попытке эмулировать выполнение арифметических инструкций, если известно значение операндов.

Статический слайсинг. Оптимизация удаляет конструкции вида push reg/pop reg, если значение регистра не используется для вычислений новых значений. Также удаляются инструкции, присваивающие новое значение этому регистру.

4. Тестирование

В качестве ключей оптимизации можно использовать следующие:

- «-JMPCLUE» – объединение базовых блоков;
- «-PROP» – распространение констант;
- «-FOLD» – свертывание констант;
- «-UNREACH» – удаление недостижимого кода;
- «-DEADELIM» – удаление мертвого кода
- «-PUSHPOP» – удаление конструкций вида push reg/pop reg и вычислений, результаты которых не используются

Запутанный код	Распутанный код

label_1:
push eax
mov eax,0ffaah
xor ecx,eax
mov ebx,15h
pop eax
mov ebx,eax
jmp loc_1
dec ecx
loc_1:
inc ebx
jmp loc_2
loc_2:
jmp loc_3
loc_3:
mov eax,12ffh
jmp loc_4
loc_4:
jnz label_2
label_2:
jz label_77
label_77:
jnz label_3
xor eax,eax
jmp label_4
jnz label_5
label_5:
inc ecx
label_3:
jmp label_1
label_4:
jns label_2
label_99:
jz label_99
	label_1:
	xor ecx,eax
	mov ebx,eax
	inc ebx
	mov eax,12ffh
	label_2:
	jnz label_1
	xor eax,eax
	jns label_2
	label_99:
	jz label_99

Табл.3. Результаты работы деобфускатора

Результаты работы деобфускатора и граф потока управления для запутанного и распутанного кода приведены в табл. 3 и на рис. 2 соответственно. В таблицах наглядно видно результаты работы алгоритмов, связанных с

анализом потока управления. Убраны все недостижимые из начальной точки вершины, проведено объединение базовых блоков, выполнен анализ условий переходов и убраны транзитные переходы. Кроме того, к базовым блокам были применены такие оптимизации, как слайсинг, свертывание и распространение констант.

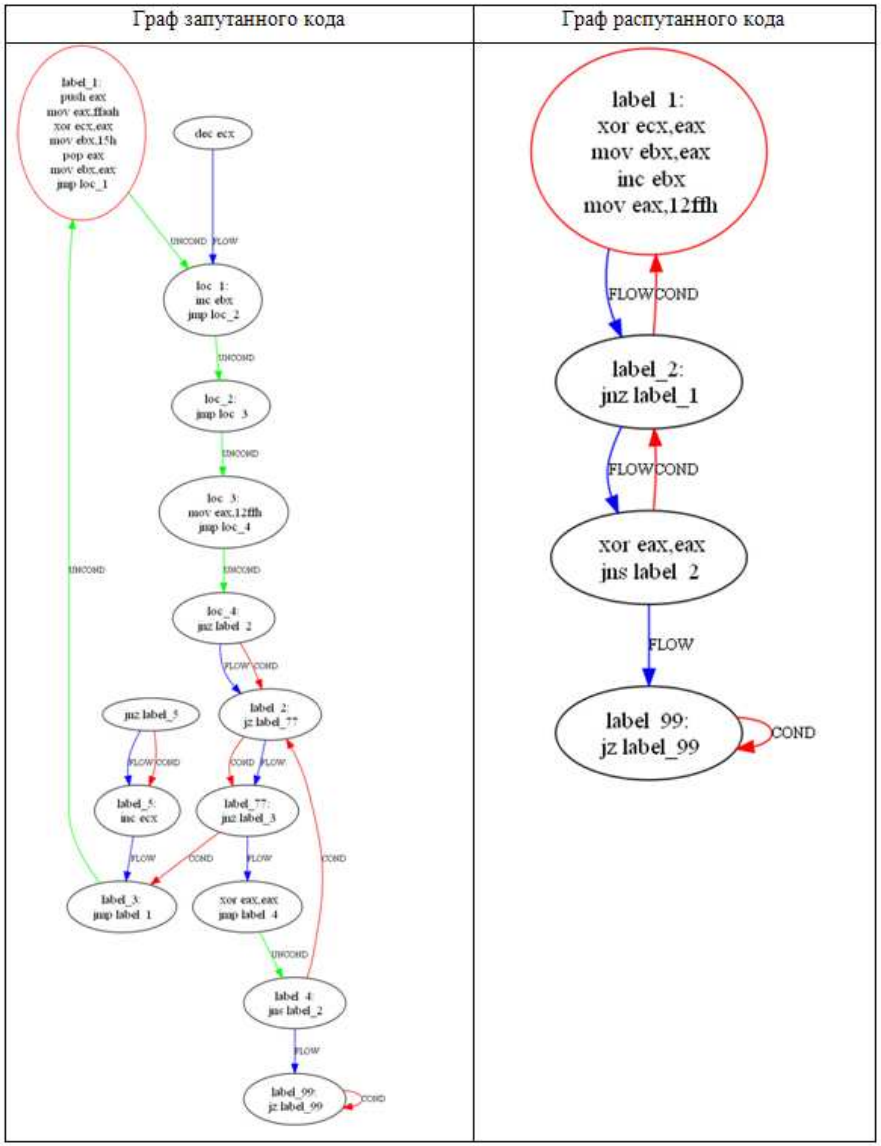

Рис. 2. Граф потока управления до и после оптимизации

В табл.4 приведены результаты последовательной оптимизации базового блока.

Табл. 4 - Результаты последовательных оптимизаций

Запутанный код	Слайсинг	Удаление мертвого кода	Распространение и свертывание констант

mov eax,10h
mov ecx,150h
mov ecx,0ffffh
mov ebx,90h
xor ebx,21h
add eax,11h
xor ecx,0ffh
mov ebx,eax
push eax
sub eax,20h
xor eax,777h
add eax,0codeh
pop eax
sub eax,5
mov ecx,eax
	mov eax,10h
	mov
	ecx,150h
	mov ecx,ffffh
	mov ebx,90h
	xor ebx,21h
	add eax,11h
	xor ecx,ffh
	mov ebx,eax
	sub eax,5
	mov ecx,eax
		mov eax,10h
		mov ecx,ffffh
		mov ebx,90h
		xor ebx,21h
		add eax,11h
		xor ecx,ffh
		mov ebx,eax
		sub eax,5
		mov ecx,eax
			mov eax,1ch
			mov ecx,1ch
			mov ebx,21h

Видно, что применение методик деобфускации дает существенное улучшение читаемости кода. Для лучшего эффекта распространение и свертывание констант применяются в одном проходе как взаимодополняющие оптимизации.

5. Заключение

В рамках данной работы разработан прототип деобфускатора программного кода, представляющий собой программное решение для оптимизации кода ассемблера процессоров intel x86.

Были реализованы следующие методики распутывающих преобразований:

- статическое построение и анализ потока управления;
- объединение базовых блоков;
- удаление недостижимого кода;
- продвижение констант;
- свертывание констант;
- удаление «мертвого» кода;
- алгебраическое упрощение, как часть алгоритма свертывания констант.

В результате это обеспечило качественную деобфускацию программного кода. Кроме того, созданная архитектура программного обеспечения позволяет легко добавлять новые алгоритмы деобфускации, т.к. предоставляет удобный доступ к данным и функции для работы с ними. Рассмотрено перспективное направление использования компиляторной инфраструктуры LLVM в качестве ядра деобфускатора, определены основные сложности такого подхода.

Список литературы

[1]. Анализ запутывающих преобразований программ. Чернов А. В., Труды Института Системного программирования РАН [электронный ресурс] http://www.citforum.ru/security/articles/analysis/, свободный.-Загл с экрана.

[2]. Reverse Compilation Techniques By Cristina Cifuentes [электронный ресурс] http://www.itee.uq.edu.au/~cristina/dcc/decompilation_thesis.ps.gz, свободный.-Загл с экрана.

[3]. Ахо А., Лам М., Сети Р., Ульман Д.Компиляторы. Принципы, технологии, инструментарий, 2-изд.: Пер с англ.- М.: ООО «И.Д. Вильямс», 2008. – 1184с. : ил.

[4]. LLVM Language Reference Manual [электронный ресурс] http://LLVM.org/docs/LangRef.html, свободный.-Загл с экрана.

[5]. Using Code Normalization for Fighting Self-Mutating Malware Danilo Bruschi, Lorenzo Martignoni, Mattia Monga [электронный ресурс] http://idea.sec.dico.unimi.it/~lorenzo/rt0806.pdf, свободный.-Загл с экрана.

[6]. John R Flex & bison. 1st edition, 304p. Levine Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472. ISBN: 978-0-596-15597-1

[7]. Библиотека BOOST C++ [электронный ресурс] http://www.solarix.ru/for_developers/cpp/boost/boost-library.shtml, свободный.-Загл с экрана.

[8]. Сик, Ли, Ламсдэйн, C++ Boost Graph Library. Библиотека программиста / Пер. с английского Сузи Р. – СПб.: Питер,2006.- 304с. ил. ISBN 5-469-00352-3.

[9]. Using Graphviz as a library [электронный ресурс] http://www.graphviz.org/pdf/libguide.pdf, свободный.-Загл с экрана.

Software deobfuscation methods: analysis and implementation

Sh.F. Kurmangaleev, K.Y. Dolgorukova,
V.V. Savchenko, A.R. Nurmukhametov, H. A Matevosyan, V.P. Korchagin
kursh@ispras.ru, unerkannt@ispras.ru
sinmipt@ispras.ru, oleshka@ispras.ru, hripsime@ispras.ru, korchagin@ispras.ru
ISP RAS, Moscow, Russia

Annotation. This paper describes the work on development of the deobfuscation software.

The main target of the developed software is the analysis of the obfuscated malware code. The need of this analysis comes from the obfuscation techniques being widely used for protecting implementations. The regular disassembly tool mostly used by an analyst transforms a binary code in a human-readable form but doesn't simplify the result or verify its correctness. Earlier for this task it was enough to apply pattern-matching cleanup of the inserted useless garbage code, but nowadays obfuscation techniques are getting more complicated thus requiring more complex methods of code analysis and simplification.

As deobfuscation methods require analysis and transformation algorithms similar to those of an optimizing compiler, we have evaluated using LLVM compiler infrastructure as a basis for deobfuscation software. The difference from the compiler is that the deobfuscation algorithms do not have the full information about the program being analyzed, but rather a small part of it. The evaluation results show that using LLVM directly does not remove all the artifacts from the obfuscated code, so to provide the cleaner output it is desirable to develop an independent tool. Nevertheless, using LLVM or similar compiler infrastructure is the feasible approach for developing deobfuscation software.

Keywords: binary code analysis, obfuscation, deobfuscation, LLVM

References

[1]. A.V. Chernov. Analiz zaputyvayushhikh preobrazovanij programm. [Analysis obfuscating program transformations] Trudy ISP RAN [The Proceedings of ISP RAS], 2002, vol.3, pp. 7-38 (in Russian).
[2]. Reverse Compilation Techniques By Cristina Cifuentes http://www.itee.uq.edu.au/~cristina/dcc/decompilation_thesis.ps.gz
[3]. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. ISBN:0321486811
[4]. LLVM Language Reference Manual http://LLVM.org/docs/LangRef.html
[5]. Using Code Normalization for Fighting Self-Mutating Malware Danilo Bruschi, Lorenzo Martignoni, Mattia Monga http://idea.sec.dico.unimi.it/~lorenzo/rt0806.pdf
[6]. John R Flex & bison. 1st edition, 304p. Levine Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472. ISBN: 978-0-596-15597-1
[7]. BOOST C++ http://www.solarix.ru/for_developers/cpp/boost/boost-library.shtmlc

[8]. Jeremy G. Siek; Lie-Quan Lee; Andrew Lumsdaine. The Boost Graph Library: User Guide and Reference Manual. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA ©2002 ISBN:0-201-72914-8
[9]. Using Graphviz as a library http://www.graphviz.org/pdf/libguide.pdf