**Вставка и удаление описателей.**

Для добавления блока(инструкции) в граф используется вставка описателя. При этом в других описателях формируются ссылки на вставляемый описатель. При удалении процедурных ветвлений должна удаляться вся процедура. Это второстепенная задача, далее рассмотрим удаление одного описателя из графа. При удалении описателя все ссылки на него модифицируются(разлинкование). Все прямые ссылки на удаляемый описатель(**Flink** и **BranchLink**) заменяются на прямую ссылку удаляемого описателя(**Flink**). Это например ветвления на удаляемый описатель и прямая ссылка предыдущего описателя(который расположен по обратной ссылке удаляемого описателя). Ветвления на удаляемый описатель находятся посредством трассировки графа(перечисление описателей линейного графа).

Удаляемый описатель помечается установкой флага **BRANCH\_IDLE\_FLAG**, который используется при вычислении размера ветвлений и некоторых других операциях. Это позволяет не смещать часть графа при вычислении размера ветвлений, таким образом избавится от необходимости повторной конвертации графа в линейный.

Для большей наглядности присвоим имена описателям: **J** – текущий удаляемый описатель, **I** – предыдущий описатель(**J.Blink**), **K** – следующий описатель(**J.Flink**). Они связаны следующим образом(**I**, **J**, **K**): **I = J.Blink**, **J = I.Flink**, **K = J.Flink**, **J = K.Blink**.

Операция удаления описателя **J**:

* Прямая ссылка предыдущего описателя **I.Flink = J.Flink**, где **I = J.Blink**. Таким образом **J.Blink.Flink = J.Flink**
* Обратная ссылка следующего описателя **K.Blink = J.Blink**, где **K = J.Flink**. Таким образом **J.Flink.Blink = J.Blink**

При этом **I.Blink** и **K.Flink** не изменяются. Если прямой или обратной ссылок нет, то операция пропускается.

Вначале опишем код, перенаправляющий ссылки в ветвлениях на новый описатель. Допустим имеется описатель **I**, перед которым был вставлен описатель **K**. Во всех описателях ветвлений на **I** ссылка **BranchLink** должна быть изменена с **I** на **K**:

**CsRedirectAllBranchLinks**(I, K):

Do *; Перечисление всех описателей, не трассировка. Граф должен быть линейным.*

Type = Gp.Type[J]

if J = TYPE\_LINE *; Описатель линейного блока пропускаем.*

> Next

fi

if Type <> TYPE\_JCC *; Если не безусловное ветвление, то выполняем проверки наличия ссылки на ветвление. Безусловное ветвление всегда её содержит.*

if !Gp.BRANCH\_DEFINED\_FLAG[J] *; Ветвление должно быть определено.*

> Next

fi

if Type = TYPE\_CALL *; Процедурное ветвление должно быть раскрыто.*

if !Gp.DISCLOSURE\_CALL\_FLAG[J]

> Next

fi

fi

if Gp.BranchLink[J] = I *; Если ссылка совпадает, то заменяем её.*

Gp.BranchLink[J] = K

fi

Next:

J + sizeof(Gp) *; Следующий описатель.*

Loop J < Limit *; Выполняем цикл, пока конец графа не достигнут.*

В **GCBE** эта манипуляция выглядит следующим образом(константа с префиксом **Eh\*** это смещение соответствующего поля в описателе):

mov ebx,Gp *;* ***J*** *– удаляемый описатель.*

mov ecx,dword ptr [ebx + EhEntryType] *;* ***Link.Flink & TYPE\_MASK*** *это тип описателя.*

and cl,TYPE\_MASK *; Налаживаем маску, очищая ссылку.*

cmp cl,HEADER\_TYPE\_JMP *; Безусловное ветвление завершает ветвь. Его нельзя удалить из графа, в силу(*[*r2*](#r2)*).*

je Error

mov edx,dword ptr [ebx + EhBlink] *;* ***I*** *– предыдущий описатель.*

mov eax,dword ptr [ebx + EhFlink] *;* ***K*** *– следующий описатель.*

*; На ссылки налаживается маска, очищаем младшие биты.*

and edx,NOT(TYPE\_MASK)

and eax,NOT(TYPE\_MASK)

.if !Zero? *; Ссылка на* ***K*** *имеется(соответственно* ***K*** *присутствует).*

and dword ptr [eax + EhBlink],TYPE\_MASK *; Младшие биты сохраняем, обнуляем только ссылку.*

or dword ptr [eax + EhBlink],edx *; Загружаем новую* ***Blink*** *без изменения флагов(младших бит).*

.endif

test edx,edx

.if !Zero? *; Ссылка на* ***I*** *имеется. Формируем новую* ***Flink****.*

and dword ptr [edx + EhFlink],TYPE\_MASK

or dword ptr [edx + EhFlink],eax

.endif

*; Если удаляется условное ветвление, то ветвь не удаляем. Это потребовало бы анализ всей ветви(поиск ссылок на описатели ветви, такие не могут быть удалены). Помечаем ветвление как удалённое, это позволит не удалять описатель.*

.if cl == HEADER\_TYPE\_JCC

or dword ptr [ebx + EhIdleBranch],BRANCH\_IDLE\_FLAG

.endif

invoke CsRedirectAllBranchLinks, Ebx, Eax *; Выполняем редирект ссылок всех описателей ветвлений на удаляемый.* ***Ebx*** *– ссылка на удаляемый описатель,* ***Eax(K)*** *– новая устанавливаемая ссылка.*

Вставка описателя может быть двух типов. Это вставка описателя перед текущим либо после текущего. Обычно необходима вставка перед текущим. Рассмотрим обе.

Вставка перед текущим описателем(**K** – текущий, **J** – вставляемый, **RwInsertHeadEntry()**):

* **J.Flink = K**
* **J.Blink = I**, где **I = K.Blink**. Тоесть **J.Blink = K.Blink**
* **I.Flink = J**, Где **I = K.Blink**. Тоесть **K.Blink.Flink = J**
* **K.Blink = J**

В **GCBE** вставка реализована следующим образом:

mov ebx,NewGp *;* ***J*** *– вставляемый описатель.*

mov eax,Gp *;* ***K*** *– текущий описатель, перед которым выполняется вставка* ***J****.*

*; Очищаем ссылки в* ***J****, оставляя неизменными флажки.*

and dword ptr [ebx + EhFlink],TYPE\_MASK *;* ***J.Flink***

and dword ptr [ebx + EhBlink],TYPE\_MASK *;* ***J.Blink***

or dword ptr [ebx + EhFlink],eax *;* ***J.Flink = K***

mov ecx,dword ptr [eax + EhBlink] *;* ***K.Blink***

and ecx,NOT(TYPE\_MASK) *; Очищаем флажки.*

or dword ptr [ebx + EhBlink],ecx *;* ***J.Blink = K.Blink***

and dword ptr [eax + EhBlink],TYPE\_MASK

or dword ptr [eax + EhBlink],ebx *;* ***K.Blink = J***

.if Ecx ; Если обратная ссылка присутствует, то загружаем **Flink**.

and dword ptr [ecx + EhFlink],TYPE\_MASK

or dword ptr [ecx + EhFlink],ebx *;* ***K.Blink.Flink = J***

.endif

invoke CsRedirectAllBranchLinks, Eax, Ebx *; Редирект* ***BranchLink****’s с* ***K*** *на* ***J****.*

Вставка после текущего описателя**(I** – текущий, **J** – вставляемый, **RwInsertTailEntry()**):

* **J.Blink = I**
* **J.Flink = K**, где **K = I.Flink**. Тоесть **J.Flink = I.Flink**
* **K.Blink = J**, где **K = I.Flink**. Тоесть **I.Flink.Blink = J**
* **I.Flink = J**

Так как все описатели имеют одинаковый размер, то возможна замена описателя без изменения его положения в адресном пространстве(и следовательно в графе), которая эквивалентна удалению и вставке. Частным случаем является замена описателя на некоторую часть графа. В таком случае вставка описателя приведёт к объединению графов. Так как положение описателя в графе, в который выполняется вставка, не меняется, то редирект ссылок в этом графе не нужен, что повышает быстродействие. При этом положение первого описателя вставляемой ветви изменяется, что требует редирект ссылок во вставляемом графе. Рассмотрим эту манипуляцию подробно.

Пусть **J** заменяемый описатель в графе [**I][J][K]**. Пусть **[I'][J'][K']** граф, на который заменяется описатель **J**. Описателя связаны между собой(аналогично и для вставляемого графа):

**I***;* ***Flink = J***

**[J]: J***;* ***Flink = K****,* ***Blink = I***

**K***;* ***Blink = J***

Обычно при вставках описатели добавляются в конец графа, при этом его размер увеличивается. После вставки граф будет следующим(пример в главе ”**Морфинг Cx-ветвлений**”):

**I***;* ***Flink = J***

**[J]: I’***;* ***Flink = J’****,* ***Blink = I***

**K***;* ***Blink = K’***

...

GpLimit: *; Лимит графа.*

**J’** *;* ***Flink = K’****,* ***Blink = J***

**K’** *;* ***Flink = K****,* ***Blink = J’***

Или в более краткой форме **[I][I’][J’][K’][K]**. Сравнив граф до и после вставки можно описать операцию замены описателя(**RwChangeEntry()** не включена в **GCBE**):

* **J = I'**
* **J.Blink = I *;*** *В редиректе ссылок нет необходимости. Эта операция добавлена изза затирания обратной ссылки* ***J.Blink*** *при копировании описателя* ***J = I'****.*
* **J.Flink = J’**
* **J’.Blink = J** *; Для вставляемого графа [****I'][J'][K']*** *необходим редирект ссылок.[[1]](#footnote-1)*
* **J.Flink.Blink = K'***; Так как* ***K = J.Flink****.*
* **K’.Flink = K**

После внесения необходимых изменений в граф необходимо выполнить конвертацию его в линейный. Конвертация выполняется посредством трассировки графа, таким образом описатели на которые отсутствуют ссылки будут удалены из линейного графа. При билдинге конвертация выполняется автоматически.

В **GCBE** есть следующие сервисы для вставки и удаления описателей.

* GP\_REDIRECT\_ALL\_BRANCH\_LINKS equ 12

Данный сервис выполняет редирект **BranchLink**'s всех описателей на указанный.

typedef NTSTATUS (\*PGCBE)(

IN PVOID GpBase, *// База линейного графа(адрес первого описателя в графе).*

IN PVOID GpLimit, *// Лимит графа.*

IN PVOID Gp, *// Описатель, ссылка на который заменяется на Link. Выполняется поиск описателей ветвлений в которых* ***BranchLink = Gp*** *и* ***BranchLink*** *заменяется на* ***Link****.*

IN PVOID Link *// Новая ссылка, загружаемая вместо* ***Gp****.*

);

* GP\_UNLINK\_ENTRY equ 13

Сервис удаляет описатель из линейного графа.

typedef NTSTATUS (\*PGCBE)(

IN PVOID GpBase,// *База линейного графа.*

IN PVOID GpLimit,// *Лимит графа.*

IN PVOID Gp // *Удаляемый описатель(адрес его в графе).*

);

* GP\_INSERT\_HEAD\_ENTRY equ 14

Сервис выполняет вставку описателя перед текущим.

typedef NTSTATUS (\*PGCBE)(

IN PVOID GpBase, *// База линейного графа.*

IN PVOID GpLimit, *// Лимит графа.*

IN PVOID CurrentGp, *// Адрес текущего описателя, перед которым вставляется* ***NewGp****.*

IN PVOID NewGp *// Адрес вставляемого описателя. Копирование описателя не выполняется, только связывание. При необходимости он может быть перенесён в конец графа, тогда конвертация графа в линейный далее не потребуется.*

);

**Удаление ветвей.**

Выше описан способ удаления описателя без удаления связанных с ним описателей. Допустим имеется конструкция(**Jcc** – условное ветвление):

Jcc L1 *; Удаляемый описатель.*

Jcc L2

...

L1: Jcc L3 *; Начало ветви* ***L1****,* ***Flink = Gp(Line)****.*

Line *; Группа линейных инструкций,* ***Blink = Gp(L1), Flink = Gp(L2)****.*

L2: … *; Продолжение ветви* ***L1*** *и начало ветви* ***L2****,* ***Blink = Gp(Line)****.*

При удалении первого ветвления должна удалиться вся связанная с ним часть графа:

Jcc L2

...

L2: ... *; Начало ветви* ***L2****.*

Часть ветви **L1** была удалена из-за отсутствия ссылок на удаляемые описатели, тоесть после удаления ветвления **Jcc L1** эта часть ветви уже не принадлежит графу(также и ветвь **L3**). Очевидно, что для подобной манипуляции необходимо выполнить трассировку ветви **L1**, раскрыв ветвление **Jcc L1** и удалить описатели, на которые нет ссылок(за исключением **Jcc L1**). Но так как трассировка это операция последовательная, то не раскрыв ветвление **Jcc L2**(оно раскрывается после **Jcc L1**) нельзя определить что на метку **L2** есть ссылки. Также имея ссылку нельзя определить, что она не принадлежит ветвлению из удаляемой части ветви. Так как трассировка по сути это проход по всем ссылкам, то удалив описатель **Jcc L1** из графа, часть ветви подлежащая удалению не будет описана. Таким образом, описатели которые окажутся пропущенными(в сравнении с исходным графом, это например маркировка описателей при трассировке исходного графа и очистка маркеров при трассировке графа с удалённым описателем, далее помеченные описатели должны быть удалены) при трассировке будут принадлежать удаляемой части ветви. Также можно протрассировать только часть графа(ветвь **L1**), раскрыв ветвление.

Так как механизм трассировки **GCBE** использует проход по обратным ссылкам([1.19](#r4)), то использование его невозможно для решения данной задачи. Необходимо изменить алгоритм трассировки:

**RwSlipTrace(J, CallbackRoutine, CallbackContext)**:

AccessFlag = Gp.AccessFlag[J] *; Определяем маркер, которым будут помечаться перечисленные блоки.*

PUSH(Null) *; Маркер конца стека.*

Do

If Gp.ACCESSED\_MASK\_FLAG[J] <> AccessFlag

> PopEntry *; Описатель уже перечислен.*

fi

Gp.AccessFlag[J] xor ACCESSED\_MASK\_FLAG *; Помечаем описатель как перечисленный инверсией маркера.*

If !CallbackRoutine(J, CallbackContext) *; Вызываем трассировочный калбек.*

> End *; Из калбека возвращена ошибка, завершаем трассировку.*

fi

Type = Gp.EntryType[J]

if Type = TYPE\_CALL *; Описатель процедурного ветвления(****Call****).*

if Gp.BRANCH\_DEFINED\_FLAG[J] & Gp.DISCLOSURE\_CALL\_FLAG[J] *; Ветвление раскрыто и адрес ветвления определён.*

PUSH(J) *; Сохраняем описатель в стеке.*

J = Gp.BranchLink[J] *; Раскрываем ветвление.*

fi

elseif Type = TYPE\_JMP *; Описатель безусловного ветвления(****Jmp****).*

if Gp.BRANCH\_DEFINED\_FLAG[J] *; Адрес ветвления определён, иначе оно завершает ветвь.*

J = Gp.BranchLink[J]  *; Начинаем новую ветвь(раскрываем ветвление).*

else

> PopEntry *; Адрес ветвления не определён. Ветвь закончена.*

fi

elseif Type = TYPE\_JCC *; Описатель условного ветвления(****Jcc****).*

PUSH(J) *; Сохраняем описатель.*

J = Gp. BranchLink[J]  *; Начинаем новую ветвь(раскрываем ветвление).*

elseif Type = TYPE\_LINE *; Описатель линейного блока.*

J = Gp.Flink[J]

if !J *; Следующий описатель отсутствует, ветвь закончена.*

PopEntry:

J = POP()

if !J *; В стеке описателей больше нет, завершаем.*

End

fi

fi

fi

Loop

Это простое раскрытие всех подряд ветвей. Таким образом, обратные ссылки(**Blink**) не используются. При применении этого алгоритма к коду([r3](#r3)) последовательность описываемых инструкций будет следующей: **40292C**, **40292E**, **402935, 402930, 402938, 402939**. Теперь можем написать алгоритм удаления ветви(код находится в модуле **GrpFlow.asm**).

*; Трассировочный калбек, помечающий описатели. Изначально в графе все описатели содержат одинаковое значение маркера.*

TraceCallback1(J):

!Gp.MARK[J] *; Помечаем описатель посредством инверсии.*

Для удаления найденных описателей необходимо удалить обратные ссылки на них(например в описателе **L2** останется обратная ссылка на **Line**). Прямые ссылки на удаляемые описатели будут отсутствовать, вследствие этого не нужно связывать описатели в которых остались обратные ссылки(они будут началом ветвей).

*; Трассировочный калбек, удаляющий обратные ссылки на помеченные описатели.*

TraceCallback2(J, Mark):

K = Gp.Blink[J]

if K & (Gp.MARK[K] = Mark) *; Обратная ссылка присутствует и предыдущий описатель помечен.*

!Gp.Blink[J] *; Удаляем ссылку.*

fi

Непосредственно код удаляющий ветвь(**J** – описатель ветвления на удаляемую ветвь):

**RwUnlinkFlow**(J):

if Gp.Type[J] = TYPE\_LINE *; Линейный блок.*

End

fi

UnlinkEntry(J) *; Удаляем ветвление.*

Mark = Gp.MARK[GpBase] *; Определяем маркер, которым будут помечаться удалённые описатели.*

RwSlipTrace(GpBase, @TraceCallback1()) *; Помечаем удалённые описатели(происходит инверсия маркера* ***Mark****).*

RwSlipTrace(GpBase, @TraceCallback2(), Mark) *; Удаляем обратные ссылки на помеченные описатели.*

Для удаления оставшихся описателей(они разлинкованы и следовательно не принадлежат графу) из графа можно выполнить конверсию графа в линейный. **GCBE** предоставляет следующие сервисы для удаления ветвей:

* GP\_SLIP\_TRACE equ 15

Трассировка без приоритетов, псевдокод и описание соответствует **RwSlipTrace()**.

typedef NTSTATUS (\*PGCBE)(

IN PVOID Graph,

IN PTRACE\_CALLBACK\_ROUTINE CallbackRoutine,

IN PVOID CallbackParameter

);

* GP\_UNLINK\_FLOW equ 16

Сервис удаляет ветвь. Описатель ветвления на ветвь должен быть удалён заранее. Для помечания описателей используется инверсия флага **FLOW\_UNLINK\_MARK** в **KitFlags**. Свойства этого флага аналогичны трассировочному(**ACCESSED\_MASK\_FLAG)**, в частности он должен быть одинаков у всех описателей(важно при слиянии графов).

typedef NTSTATUS (\*PGCBE)(

IN PVOID GpBase

);

**Оптимизация ветвлений.**

Для упрощения структуры графа и уменьшения размера собираемого кода необходимо выполнить оптимизацию ветвлений. Оптимизация состоит из нескольких этапов:

- Удаление **Cx**-ветвлений.

- Удаление **Idle**-ветвлений.

- Вычисление размера ветвлений(**Short**/**Near**).

Рассмотрим подробно все этапы.

**Сx-ветвления.**

Так как при сборке кода граф изменяется, а **Cx**-ветвления(**Jecxz**, **Loopd**, **Loopde**, **Loopdne**) могут быть только короткими(**Short**), то необходимо использовать средства позволяющие выполнять длинные(**Near**) ветвления. Это можно выполнить двумя способами:

* Отморфить ветвление, заменив его на более простые аналоги. Например инструкция **Jecxz Flow** выполняет ветвление на ветвь **Flow**, если регистр **Ecx** обнулён. Разложив эту инструкцию на примитивы, можно написать её аналог:

pushfd  *; Сохраняем флажки.*

test ecx,ecx *; Проверяем регистр на ноль.*

jnz @f  *; Если не ноль, то восстанавливаем флажки и переходим к следующей инструкции.*

popfd *; Иначе восстанавливаем флажки.*

jmp near Flow *; И выполняем дальнее ветвление на новую ветвь.*

@@:

popfd

... *; Блок, следующий за* ***Jecxz****.*

* Выполнить вставку дальнего ветвления по метке. Например использовать следующую конструкцию:

Jecxz short Stub

...

Stub:

Jmp near Flow

При вставке ветвления в виде стаба необходимо чтобы вставляемое ветвление находилось в пределах **127** байт от ветвления. Для этого нужно выполнить вставку стаба(инструкции **Jmp near Flow**) в другую ветвь. Такая вставка потребует связку блоков между стабом безусловным ветвлением. Например имеется следующий код:

Jecxz short Flow

Ip:

push eax

push edx

После вставки код будет выглядеть следующим образом:

Jecxz short Stub

push eax

Jmp Link

Stub:

Jmp near Flow

Link:

push edx

Либо совместив ветвление на **Link** с условным ветвлением:

Jecxz short Stub

jmp Ip

Stub:

Jmp near Flow

Ip:

push eax

push edx

Такая вставка выполняется непосредственно перед сборкой кода. После вставки граф не должен изменяться, иначе понадобится вставка заново. Это главный недостаток использования вставок, именно по этой причине не применяется в **GCBE**, а используется морфинг(разложение на примитивы). Хотя билдер может собирать **Cx**-ветвления.

Вставки в граф рассмотрены, и мы можем написать код для изменения **Cx**-ветвлений посредством вставки ветвления. Пусти **I** – описатель **Cx**-ветвления. Тогда адрес следующей за ветвлением инструкции **Gp.Flink[I]** и соответственно ветвь, на которую происходит ветвление **Gp.BranchLink[I]**. Вставка ветвления должна выполняться в конец графа, поэтому описатель **J** ветвления **jmp Ip** будет располагаться по адресу [**GpLimit]**. Описатель **K** ветвления **Jmp near Flow** будет располагаться по адресу **[GpLimit + sizeof(Gp)]**:

Jecxz short Stub *; Описатель для данной инструкции –* ***I****.*

jmp Ip *;* ***J***

Stub:

Jmp near Flow *;* ***K***

Ip:

Тогда псевдокод, удлиняющий одно **Cx**-ветвление будет следующим:

Global Op = OPCODE(“Jmp”) *; Глобальная переменная, в которую загружается опкод инструкции* ***Jmp****. Это может быть метка в коде, по адресу которой находится инструкция* ***Jmp****. Не обязательно указывать в описателе адрес этой инструкции(билдер не использует опкод если адрес ветвления определён), но может понадобится если будет использоваться* ***LDE****(сторонним кодом). Поэтому желательно формировать опкод.*

*; Формируем описатель* ***J****(****jmp Ip****)****: BRANCH\_HEADER****.*

@J = GpLimit *; Добавляем описатель в конец графа.*

GpLimit + sizeof(Gp) *; Лимит графа увеличивается.*

Gp.Type[J] = TYPE\_JMP *; Тип описателя – безусловное ветвление.*

!Gp.Flink[J] *; Безусловные ветвления завершают текущую ветвь и начинают новую, поэтому* ***Flink = Null****.*

Gp.BRANCH\_DEFINED\_FLAG[J] *; Адрес ветвления определён.*

Gp.ACCESSED\_MASK\_FLAG[J] = Gp.ACCESSED\_MASK\_FLAG[I] *; Трассировочный флаг должен быть одинаков во всех описателях.*

Gp.Blink[J] = @I *; Предыдущий описатель -* ***I****(****Cx****-ветвление).*

Gp.BranchLink[J] = Gp.Flink[I] *; Адрес ветвления на следующую инструкцию за ветвлением.*

Gp.BranchAddress[J] = Gp.Flink.Address[I] *; Это следующий описатель за* ***I****(адрес* ***Ip****).*

Gp.Address[J] = @Op *; Адрес опкода. Не обязательно его указывать.*

*; Формируем описатель* ***K****(****Jmp near Flow****)****: BRANCH\_HEADER****.*

@K = GpLimit

GpLimit + sizeof(Gp)

Gp.Type[K] = TYPE\_JMP

!Gp.Flink[K]

Gp.BRANCH\_DEFINED\_FLAG[K]

Gp.ACCESSED\_MASK\_FLAG[K] = Gp.ACCESSED\_MASK\_FLAG[I]

!Gp.Blink[K] *; Предыдущий описатель отсутствует, так как это новая ветвь.*

Gp.BranchLink[K] = Gp.BranchLink[I] *; Описатель ветви* ***Flow****.*

Gp.BranchAddress[K] = Gp.BranchAddress[I] *; Адрес ветви* ***Flow****.*

Gp.Address[K] = @Op

*; Дополнительные поправки.*

!Gp.Flink.Blink[I] *; Предыдущей ссылки нет, так как новая ветвь, образованная ветвлением.*

CsRedirectAllBranchLinks(Gp.Flink[I], J) *; Редирект ссылок в описателях ветвлений с* ***Gp.Flink[I]*** *на* ***J****.*

CsRedirectAllBranchLinks(Gp.BranchLink[I], K) *; Редирект ссылок в описателях ветвлений с* ***Gp.BranchLink[I]*** *на* ***K****.*

Для подобной модификации всех описателей **Cx**-ветвлений необходимо перечислить все описатели. Модификация графа из трассировочного калбека недопустима, поэтому граф должен быть линейным и использоваться последовательное перечисление описателей.

**Морфинг Cx-ветвлений.**

При морфинге **Cx**-ветвлений замена на примитивы выполняется автоматически. Создаётся граф для кода, который содержит примитивы для всех ветвлений, подлежащих замене. Затем из этого графа выделяется соответствующий ветвлению блок примитивов и выполняется вставка его в граф вместо заменяемого ветвления. Рассмотрим морфинг **Cx**-ветвления подробно. Опишем все примитивы в таблице. Для примера рассмотрим только один:

L1: ...  *;* ***Flink = L0***

L0: Loopdne Flow  *;* ***Flink = L2, Blink = L1***

L2: ...  *;* ***Blink = L0***

Разложенный на примитивы аналог инструкции **Loopdne Flow** будет следующим:

pushfd *; Ветвление не изменяет флажки, сохраняем их.*

lea ecx,[ecx - 1] *; Декремент регистра* ***Ecx*** *без изменения флагов.*

jz @f *; Ветвление если* ***!ZF & Ecx****.*

test ecx,ecx

jz @f

popfd *; Извлекаем флажки из стека.*

jmp Flow *; И выполняем дальнее ветвление на* ***Flow****.*

@@: popfd *; Восстанавливаем флажки. Регистр* ***Ecx*** *уменьшен.*

После морфинга код будет следующим:

L1: ...  *;* ***Flink = L0***

L0: pushfd  *;* ***Flink = L3, Blink = L1***

L2: ...  *;* ***Blink = L4***

L3: lea ecx,[ecx - 1] *;* ***Blink = L0***

jz L4

test ecx,ecx

jz L4 *;* ***BranchLink' = BranchLink - Src(L3) + Dst(L3)****, так как расположение описателей неизменно относительно начала графа.*

popfd

jmp Flow

L4: popfd *;* ***Flink = L2***

Описатель заменяемого ветвления будет заменён на описатель первого примитива из блока. При этом нет необходимости корректировать ссылки(см. **CsRedirectAllBranchLinks()** и **Blink**), так как положение описателя в памяти не изменилось. Остальные примитивы будут добавлены в конец графа, таким образом расширяя его.

Каждый блок примитивов будет описан в графе. Очевидно, что для предотвращения выхода парсера за пределы блока необходимо завершить ветвь и закрыть ветвление. Пусть инструкция **Ret** будет закрывать всю ветвь, а инструкция **Jmp eax** будет закрывать ветвление[[2]](#footnote-2). Определим макро для закрывающих описателей:

%HALT macro

ret *; Завершает ветвь.*

endm

%LINK macro

jmp eax *; Закрывает ветвление.*

endm

Блоки, описывающие примитивы для всех **Cx**-ветвлений объединим вместе, связав их условными ветвлениями:

CxReplaceTable:

*; Условие ветвления* ***JCC\_LOOPNE*** *соответствует инструкции* ***Loopne****[[3]](#footnote-3).*

je $Loopdne *;* ***Loopdne***

je $Loopwne *;* ***Loopwne***

; **JCC\_LOOPE**

je $Loopde *;* ***Loopde***

je $Loopwe *;* ***Loopwe***

*;* ***JCC\_LOOP***

je $Loopd *;* ***Loopd***

je $Loopw *;* ***Loopw***

*;* ***JCC\_ECXZ***

je $Jecxz *;* ***Jecxz***

je $Jcxz  *;* ***Jcxz***

%HALT  *; Макро завершающее ветвь(не обязательно), это инструкция* ***Ret****.*

$Loopdne: *; Блок примитивов для инструкции* ***Loopdne***.

pushfd

lea ecx,[ecx - 1]

jz @f

test ecx,ecx

jz @f

popfd

%LINK  *; Макро закрывающее ветвление, инструкция* ***Jmp eax****.*

@@: popfd

%HALT *; Макро завершающее ветвь.*

$Loopwne:  *; Блок примитивов для инструкции* ***Loopwne****.*

pushfd

...

Теперь напишем код, выполняющий морфинг одного **Cx**-ветвления:

*;* ***Src*** *– ссылка на блок примитивов.*

*;* ***Dst*** *– ссылка на текущий формируемый описатель.*

**CsMorphJcx**(Jcx:PCC\_BRANCH\_HEADER):

JcxHeader:CC\_BRANCH\_HEADER

if !Jcx.BRANCH\_CX\_FLAG *;* ***Cx****-ветвление.*

End

fi

Copy(Jcx, @JcxHeader, sizeof(Gp)) *; Копируем заменяемый описатель в буфер. Далее понадобятся ссылки из него.*

*; Вычисляем смещение в таблице описателя ветвления на блок примитивов соответствующего типа(****je $Loopdne****).*

J = Gp(CxReplaceTable) + 2\*Jcx.Condition\*sizeof(Gp)

if Jcx.JCC\_X16\_MASK *; Префикс переопределения размера.*

J + sizeof(Gp) *; Следующий описатель для* ***16****-битных ветвлений(****je $Loopwne****).*

fi

Src = Gp.BranchLink[J] *; Первый описатель, на который заменяется ветвление(****$Loopdne****).*

SrcEntry = Src *; Сохраняем ссылку на первый описатель в блоке.*

Dst = \*GpLimit *;* ***L3****, будем добавлять описатели в конец графа.*

AccessFlag = JcxHeader.ACCESSED\_MASK\_FLAG *; Трассировочный флаг должен быть одинаков во всех описателях.*

*; Описатель первой инструкции в блоке(****pushfd****).*

Copy(Src, Jcx, sizeof(Gp)) *; Копируем его на место заменяемого ветвления.*

Gp.ACCESSED\_MASK\_FLAG[J] = AccessFlag *; Устанавливаем трассировочный флаг.*

Src + sizeof(Gp) *;* ***L3***

*; Связываем описатели.*

Gp.Flink[Jcx] = Dst *; Прямая ссылка* ***L0.Flink = L3****.*

Gp.Blink[Jcx] = JcxHeader.Blink *; Обратная ссылка* ***L0.Blink = L1****(она не изменяется, но затирается при копировании описателя).*

Delta = Dst – Src *; Разница смещений описателей[[4]](#footnote-4). На это значение будет выполняться поправка ссылок.*

Do

Type = Gp.Type[Src] *; Тип описателя.*

if Type = TYPE\_LINE *; Линейный блок.*

if !Gp.Flink[Src] *; Прямая ссылка отсутствует, ветвь завершена(макро* ***%HALT****).*

Dst - sizeof(Gp) *;* ***L4***

J = JcxHeader.Flink *;* ***L2***

*; Связываем описатели.*

Gp.Flink[Dst] = J *; Прямая ссылка* ***L4.Flink = L2****.*

Gp.Blink[J] = Dst *; Обратная ссылка* ***L2.Blink = L4****.*

\*GpLimit = Dst *; Возвращаем новый лимит графа.*

End

else *; Прямая ссылка присутствует.*

IsFlink: Copy(Src, Dst, sizeof(Gp)) *; Копируем описатель.*

Gp.Flink[Dst] + Delta *; Связываем их.*

IsBlink:

if Gp.Blink[Dst]

if Gp.Blink[Dst] = SrcEntry *; Если обратная ссылка на* ***L0****, то загружаем её.*

Gp.Blink[Dst] = Jcx*;* ***L3.Blink = L0****.*

else

Gp.Blink[Dst] + Delta *; Иначе корректируем.*

fi

fi

Gp.ACCESSED\_MASK\_FLAG[Dst] = AccessFlag

Src + sizeof(Gp)

Dst + sizeof(Gp) *; Указатель на следующий формируемый описатель.*

fi

Loop *; Цикл по достижению макроса* ***%HALT****.*

elseif Type = TYPE\_JMP *; Безусловное ветвление. Это макро* ***%LINK****.*

Copy(Src, Dst, sizeof(Gp))

Gp.BranchLink[Dst] = JcxHeader.BranchLink *; Корректируем ссылку на ветвь и адрес.*

Gp.BranchAddress[Dst] = JcxHeader.BranchAddress

Gp.BRANCH\_DEFINED\_FLAG[Dst] *; Адрес ветвления определён.*

> IsBlink

elseif Type = TYPE\_JCC

Gp.BranchLink[Dst] + Delta *; Корректируем ссылку на ветвь.*

> IsFlink

fi

Для морфинга всех **Cx**-ветвлений необходимо выполнить проход по всем описателям графа. При этом **CsMorphJcx()** будет расширять граф из-за вставок блоков в конец графа.

**CsMorphJcxGraph**(GpBase, GpLimit):

J = GpBase

Do

Type = Gp.Type[J]

if Type = TYPE\_JCC or TYPE\_JMP

!Gp.BRANCH\_SIZE\_MASK[J] *; Изначально ветвления короткие(****Short****). Далее этот флажок будет устанавливать оптимизатор. Флаг* ***BRANCH\_IDLE\_FLAG*** *не очищаем, так как некоторые описатели могут быть удалены из графа(см. ).*

fi

if Type = TYPE\_JCC

if Gp.BRANCH\_CX\_FLAG[J] *;* ***Cx****-ветвление.*

**CsMorphJcx**(J)

fi

fi

J + sizeof(Gp) *; Следующий описатель.*

Loop J < GpLimit

Код можно увидеть в модуле ***GrpJcx.asm*.** При таком морфинге сохраняется регистр флагов **EFlags**(**Cx**-ветвление не изменяют флаги). В большинстве случаев он не используется или изменяется после **Cx**-ветвлений. В таком случае нет необходимости его сохранять, например разложенный на примитивы аналог инструкции **Loopdne Flow** будет следующим:

lea ecx,[ecx - 1] *; Декремент* ***Ecx****,* ***ZF*** *сохраняется.*

jz @f *; Ветвление если* ***!ZF & Ecx****.*

test ecx,ecx *;* ***Loopdne***

jnz Flow

@@:  *; Следующая инструкция.*

Для инструкции **Loopd** **Flow** аналог будет следующим:

dec ecx *; Для* ***Loopw*** *будет использоваться регистр* ***Cx****, тоесть* ***dec cx****.*

jnz Flow *;* ***Near****-ветвление.*

Подобный морфинг оптимизирует размер и уменьшает вероятность создания сигнатур для детекта[[5]](#footnote-5), но увеличивает размер движка, выполняющего морфинг. Необходимо определить используют ли инструкции, следующие за **Cx**-ветвлением и расположенные в ветви значение регистра флагов[[6]](#footnote-6). При этом необходимо раскрыть все ветви, проверив что отсутствуют инструкции, использующие флажки(в таких инструкциях значение регистра флагов влияет на результат исполнения инструкции). Раскрываются все ветви, каждая обходится с целью найти инструкцию, использующую флажки. Обход заканчивается на инструкции, изменяющей флажки. Нельзя допустить вероятность ошибочного анализа, например игнорируя процедурные ветвления (так как они обычно не используют значение регистра флагов).

При трассировке должен использоваться маркер, отличный от **AccessFlag**, так как маркером будет помечен небольшой участок графа, а использование **AccessFlag** потребует цикл его очистки. Введём новый флаг **AccessFlag2** и маску для него **ACCESSED\_MASK\_FLAG2**.

Трассировка последовательная, аналогичная **RwSlipTrace()**, но в зависимости от возвращаемого из колбека значения должна завершаться трассировка текущей ветви. Изменим проверку возвращаемого значения и введём в трассировщик маску для маркера:

**RwSlaveTrace**(J, CallbackRoutine, CallbackContext, ACCESSED\_MASK\_FLAG2):

AccessFlag = Gp[J] & ACCESSED\_MASK\_FLAG2 *; Определяем маркер, которым будут помечаться перечисленные блоки.*

PUSH(Null) *; Маркер конца стека.*

Do

If (Gp[J] & ACCESSED\_MASK\_FLAG2) <> AccessFlag

> PopEntry *; Описатель уже перечислен.*

fi

Gp[J] xor ACCESSED\_MASK\_FLAG2 *; Помечаем описатель инверсией маркера как перечисленный.*

Status = CallbackRoutine(J, CallbackContext) *; Вызываем трассировочный калбек.*

If Status = STATUS\_END\_FLOW

> PopEntry *; Завершаем ветвь.*

elseif Status = STATUS\_ END\_TRACE

End *; Завершаем трассировку.*

fi

Type = Gp.EntryType[J]

if Type = TYPE\_CALL *; Описатель процедурного ветвления(****Call****).*

if Gp.BRANCH\_DEFINED\_FLAG[J] & Gp.DISCLOSURE\_CALL\_FLAG[J] *; Ветвление раскрыто и адрес ветвления определён.*

PUSH(J) *; Сохраняем описатель в стеке.*

J = Gp.BranchLink[J] *; Раскрываем ветвление.*

fi

elseif Type = TYPE\_JMP *; Описатель безусловного ветвления(****Jmp****).*

if Gp.BRANCH\_DEFINED\_FLAG[J] *; Адрес ветвления определён, иначе оно завершает ветвь.*

J = Gp.BranchLink[J]  *; Начинаем новую ветвь(раскрываем ветвление).*

else

> PopEntry *; Адрес ветвления не определён. Ветвь закончена.*

fi

elseif Type = TYPE\_JCC *; Описатель условного ветвления(****Jcc****).*

PUSH(J) *; Сохраняем описатель.*

J = Gp. BranchLink[J]  *; Начинаем новую ветвь(раскрываем ветвление).*

elseif Type = TYPE\_LINE *; Описатель линейного блока.*

J = Gp.Flink[J]

if !J *; Следующий описатель отсутствует, ветвь закончена.*

PopEntry:

J = POP()

if !J *; В стеке описателей больше нет, завершаем.*

End

fi

fi

fi

Loop

Трассировочный колбек будет последовательно перебирать все инструкции, следующие за **Cx**-ветвлением, и искать инструкции, изменяющие и использующие флажки по маске:

**TraceCallback**(J, NL, CallbackContext):

Ip = Gp.Address[J] *; Ссылка на опкод.*

if INEFLAGS(Ip) & OUTEFLAGS(OPCODE(“Test”))  *; Проверяем используемые флажки по маске для инструкции* ***Test****:* ***CF OF PF SF ZF (AF)****.*

Ret STATUS\_END\_TRACE *; Инструкция* ***[Ip]*** *использует значение одного из регистров, изменяемых инструкцией* ***Test****. Завершаем трассировку, морфинг без сохранения флагов не может использоваться.*

fi

*; Значение флажков не используется инструкцией.*

if OUTEFLAGS(Ip) & OUTEFLAGS(OPCODE(“Test”)) *; Проверяем изменяются ли флажки.*

Ret STATUS\_ END\_FLOW *; Флажки изменяются, завершаем ветвь.*

fi

if Gp.Type[J] = TYPE\_LINE *; Описатель линейного блока.*

if !Gp.Flink[J] & !NL *; Если пямая ссылка отсутствует, то это инструкция* ***Ret****. Если* ***!NL****, то возврат из процедуры, с которой начата трассировка. Далее нельзя определить использование флагов, так как вызывающая процедура не описана в графе.*

Ret STATUS\_END\_TRACE

fi

fi

Ret

**Idle-ветвления.**

Наряду с удалением **Cx**-ветвлений из графа используется удаление паразитных ветвлений. Паразитное или холостое ветвление(**Idle**) – это ветвление на безусловное ветвление, на ветвление с одинаковым условием или на следующую инструкцию, причём такое ветвление не должно изменять контекст(например инструкция **Loop** изменяет регистр **Ecx**).

Опишем условия(**Idle**-условия), при которых ветвление является паразитным(**J** – описатель ветвления):

* Безусловное ветвление на следующую инструкцию:

J.Type = TYPE\_JMP

J.BRANCH\_DEFINED\_FLAG *; Адрес ветвления определён.*

J.BranchLink = J + sizeof(Gp) *; Так как прямая ссылка* ***Flink*** *в безусловных ветвлениях отсутствует(такие ветвления начинают новую ветвь), то определить следующую инструкцию можно проверив адрес описателя и только в линейном графе(так как в нелинейном графе адрес инструкции не связан с положением описателей). Это необходимо делать сразу перед сборкой графа.*

Jmp Flow

Flow: ...

* Безусловное ветвление на безусловное:

J.Type = TYPE\_JMP

J.BRANCH\_DEFINED\_FLAG

J.BranchLink.Type = TYPE\_JMP

J.BranchLink.BRANCH\_DEFINED\_FLAG

Jmp Flow

...

Flow: Jmp Lavel

* Условное ветвление на следующую инструкцию:

J.BranchLink = J.Flink

J.Type = TYPE\_JCC

!J. BRANCH\_CX\_FLAG *; Не* ***Cx****-ветвление[[7]](#footnote-7).*

Jnz Flow

Flow: ...

Jecxz Flow

Flow: ...

* Условное ветвление на безусловное:

J.Type = TYPE\_JCC

J.BranchLink.Type = TYPE\_JMP

J.BranchLink.BRANCH\_DEFINED\_FLAG

Jz Flow

...

Flow: jmp Label

Loop Flow

...

Flow: jmp Label

* Два условных ветвления с одинаковым условием:

J.Type = TYPE\_JCC

!J.BRANCH\_CX\_FLAG *; Исключая* ***Jecxz****.*

J.Fink.Type = TYPE\_JCC

!J.Fink.BRANCH\_CX\_FLAG

J.Condition = J.Fink.Condition

Jz Flow

Jz Flow

* Условное ветвление на условное, с одинаковым условием:

J.Type = TYPE\_JCC

!J.BRANCH\_CX\_FLAG *; Исключая* ***Jecxz****.*

J.BranchLink.Type = TYPE\_JCC

!J. BranchLink.BRANCH\_CX\_FLAG

J.Condition = J.BranchLink.Condition

Jz Flow

...

Flow: jz Label

* Процедурное ветвление на безусловное(паразитное здесь безусловное ветвление, процедурное повышает **NL** и образует две ветви, поэтому не может быть удалено):

J.Type = TYPE\_CALL

J.BRANCH\_DEFINED\_FLAG

J.DISCLOSURE\_CALL\_FLAG *; Ветвление раскрыто(ветвь описана в графе).*

J.BranchLink.Type = TYPE\_JMP

J.BranchLink.BRANCH\_DEFINED\_FLAG

Call Flow

...

Flow: Jmp Label

* Если на блок нет прямых ссылок(**Flink**), а имеется хоть одно безусловное ветвление, то это ветвление паразитное. Такое ветвление удаляется, а предыдущий блок(перед **Jmp**) и тот, на который выполняется ветвление объединяются. При этом ветвь, которая заканчивается удаляемым ветвлением удлиняется. Например, следующий код:

mov eax,ebx

Jmp Flow

...

Flow: inc esi

Если ссылок на **Flow** кроме инструкции **Jmp Flow** больше нет, то эта инструкция удаляется и код превратится в следующий:

mov eax,ebx

Flow: inc esi

В **GCBE** интересно поведение оптимизатора для подобных конструкций. Рассмотрим для примера следующий код:

test eax,eax ; 1

jz L2 ; 2

js L1 ; 3

ret ; 4

L2:

nop ; 5

ret ; 6

L1:

inc eax ; 7

jmp L2 ; 8

test eax,eax ; 1

jz L1 ; 2

js L2 ; 3

ret ; 4

L2:

nop ; 7

ret ; 8

L1:

inc eax ; 5

jmp L2 ; 6

test eax,eax ; 1

jz L1 ; 2

js L2 ; 3

ret ; 4

L1:

inc eax ; 5

L2:

nop ; 6

ret ; 7

При пересборке кода слева, он останется без изменений. Если поменять метки в ветвлениях, как в центре, то после пересборки получится код справа. Во втором случае ветвление удаляется, причём без применения метода, описанного в данном пункте условий. Причина такого поведения заключается в следующем. При создании графа приоритет описания блоков аналогичен как при трассировке(описывается текущая ветвь, затем раскрываются условные ветвления, процедурные ветвления раскрываются в последнюю очередь). Последовательность описателей при парсинге кода, как и при трассировке, указана в колонках справа от кода. Соответственно это номер описателя в линейном графе(индекс, тоесть **@J = N\*sizeof(Gp)**). Для кода слева отсутствует **Idle**-условие **Jmp $'**(безусловное ветвление на следующую инструкцию), так как ветвь **L2** описывается раньше, чем **L1** и таким образом ветвление **Jmp L2** не раскрывается(так как ветвь **L2** уже описана). Для кода в центре соблюдается такое условие. Ветвление **Jmp L1** раскрывается, так как ветвь **L2** не описана. В графе за описателем ветвления будет находиться описатель **7**, на который выполняется ветвление. Оптимизатор обнаруживает условие **J.BranchLink = @J + sizeof(Gp)** и удаляет ветвление. При этом описатели связываются(**J.BranchLink.Blink = J.Blink, J.Blink.Flink = J.BranchLink**). После билдинга получается код справа. Очевидно в обоих вариантах кода ветвление **Jmp L2** паразитное, так как соблюдается условие, описанное в данном пункте – на ветвь **L2** нет прямых ссылок(**Gp(Jcc L2).BranchLink = @Gp(Jcc L2)**). Условие же **Jmp $'** применимо только для кода в центре, так какзависит от положения описателя, следовательно и от последовательности парсинга/трассировки.

Ссылки на паразитные ветвления правятся, например для **Call Flow** ссылка заменяется с **Flow** на **Label**(см. **CsRedirectAllBranchLinks()**). После такой правки выполняется рекурсивный вызов оптимизатора, так как после удаления паразитного ветвления может сформироваться новое, например:

Jz Flow

jmp Flow

Flow: jmp Label

Здесь **Idle**-ветвление **Jmp Flow**. После первого прохода оптимизатора оно будет удалено и получится код:

Jz Flow

Flow: jmp Label

Таким образом, формируется ещё одно **Idle**-ветвление **Jz Flow**, которое при следующем проходе будет удалено оптимизатором(останется **jmp Label**).

При конвертации графа в линейный все разлинкованные(ссылки на которые были удалены или изменены) описатели **Idle**-ветвлений будут автоматически удалены(так как на них нет ссылок, а трассировка выполняется проходом по всем ссылкам).

**Инверсия условий.**

Рассмотрим конструкцию следующего вида:

Jcc Flow *;* ***J***

Jmp Label *;* ***K = J.Flink****. Не* ***Cx****-ветвление.*

Flow: ...

Здесь ветвление на ветвь **Label** выполняется, если условие ветвления **Jcc** не соблюдается. В случае **Cx**-ветвления оно удлиняется и выполняется инверсия условия(например для **Jecxz** длинный(**Near**) переход на **Label**, если в регистре **Ecx** не нулевое значение). Для не **Cx**-ветвлений требуется оптимизация. Конструкция эквивалентна условному ветвлению(условие инверсное):

Jcc Label *;* ***J***

Flow: ...

Оптимизация заключается в удалении безусловного ветвления и изменения условия в **Jcc**. Такое удаление приводит к объединению ветвей, которое возможно, только если прямых ссылок на присоединяемый блок нет([1.10](#r1)). Опишем свойства данной конструкции.

* J.Type = TYPE\_JCC *; Условное ветвление.*

!J. BRANCH\_CX\_FLAG *; Не* ***Cx****-ветвление.*

J.Flink.Type = TYPE\_JMP *; Следующий описатель – безусловное ветвление.*

J.Flink.BRANCH\_DEFINED\_FLAG  *; Адрес ветвления определён.*

!J.BranchLink.Blink *; Прямые ссылки на* ***J.BranchLink*** *отсутствуют.*

Опкод условного ветвления образуется сложением базы(**\_OPCODE\_BASE**) с кодом условия(ref1)[[8]](#footnote-8). Коды прямых и обратным им условий различаются на 1, причём код пары прямого и обратного ему условия всегда кратен 2. Таким образом, это инверсия младшего бита в коде условия: . Напишем псевдокод, выполняющий это преобразование:

if (Gp.Type[J] = TYPE\_JCC) & !Gp.BRANCH\_CX\_FLAG[J] *; Условные ветвления всегда имеют прямую ссылку(****Flink****).*

K = Gp.Flink.[J] *; Описатель инструкции* ***Jmp****.*

if (Gp.Type[K] = TYPE\_JMP) & Gp.BRANCH\_DEFINED\_FLAG[K]

if !Gp.BranchLink.Blink[J]  *; Прямых ссылок на* ***Flow*** *нет(она есть только если есть обратная ссылка).*

Gp.Flink[J] = Gp.BranchLink[J] *; Прямая ссылка.*

Gp.BranchLink.Blink[J] = @J *; Обратная ссылка.*

Gp.BranchLink[J] = Gp.BranchLink[K] *; Ветвь.*

Gp.Condition[J] xor 1 *; Инверсия условия.*

Gp.BRANCH\_IDLE\_FLAG[K] *; Помечаем ветвление как удалённое.*

CsRedirectAllBranchLinks(K, Gp.BranchLink[K]) *; Редирект ссылок в ветвлениях с* ***K*** *на новый* ***Gp.Flink[J]*** *(это ветвь* ***Label****).*

else *; Прямая ссылка на* ***Flow*** *есть. Попытаемся присоединить ветвь* ***Label*** *к* ***J****.*

if !Gp.BranchLink.Blink[K] *; Прямых ссылок на* ***Label*** *нет.*

Gp.Flink[J] = Gp.BranchLink[K] *; Прямая ссылка.*

Gp.BranchLink.Blink[K] = @J *; Обратная ссылка.*

*; Помечаем ветвление как удалённое и выполняем редирект ссылок с ветвления на ветвь. Можно перенести первый в ветви описатель на место* ***K*** *и выполнить корректировку* ***Flink*** *и* ***Blink****. В этом случае не потребуется редирект ссылок в ветвлениях на* ***K****, но потребуется редирект ссылок с* ***Gp.BranchLink[K]*** *на* ***K****.*

Gp.BRANCH\_IDLE\_FLAG[K]

CsRedirectAllBranchLinks(K, Gp.BranchLink[K])

fi

fi

fi

fi

При применении данного алгоритма к коду:

72 04 jb L1

85 C0 test eax,eax

74 05 jz short Flow *;* ***J****,* ***KitFlags & JCC\_TYPE\_MASK = JCC\_Z(4)****.*

L1: E9 XXXX jmp Label *;* ***K***

Flow: ...

Получим следующий код:

0F 82 jb Label  *; Ссылка изменилась в* ***CsRedirectAllBranchLinks()****.*

85 C0 test eax,eax

Label: 0F 85 XXXX jnz Label *;* ***J****,* ***(JCC\_Z(4) xor 1) +* *JCC\_NEAR\_OPCODE\_BASE(0x80) = 0x85****.*

Flow: ...

При применении данного алгоритма к коду:

jz Flow  *;* ***Flink = Gp(jmp Label)***

jmp Label *;* ***Blink = Gp(Jz Flow), !Flink***

...

Label: inc ebx  *;* ***Flink = Gp(Flow), !Blink***

Flow: inc ecx  *;* ***Blink = Gp(Label)***

Так как инструкция по метке **Flow** не является началом ветви(имеется **Blink**), то выполнить инверсию условий нельзя. Вторая часть алгоритма удаляет инструкцию **jmp Label**. Получится код:

jz Flow  *;* ***Flink = Gp(Label)***

Label: inc ebx  *;* ***Flink = Gp(Flow), Blink = Gp(jz Flow)***

Flow: inc ecx  *;* ***Blink = Gp(Label)***

При подобных изменениях необходимо учитывать образование паразитных ветвлений и необходимость повторного вызова оптимизатора. Допустим имеется конструкция:

Jnz L3

L1: ...

L2: ...

Jz L1

Jmp L2 *;* ***Idle***

L1: ...

L2: Jnz L3

...

Jnz L2

L1: ...

L2: Jnz L3 *;* ***Edle***

...

→

→

Здесь один блок, для которого будет выполняться инверсия условия, кроме него никакие **Idle**-условия не соблюдаются. После инверсии получится код в центре. Условие **ZF** изменилось на **!ZF**, безусловное ветвление удалено и произошло слияние ветвей. Это привило к образованию **Idle**-ветвления(безусловное ветвление на безусловное, условия которых совпадают). Если на метку **L2** ссылок больше нет, то ветвление по метке **L2** удаляется. Образуется код справа. Таким образом, инверсия условия требует перезапуск оптимизатора.

**Вычисление размера ветвлений.**

Перед сборкой графа необходимо определить смещения в ветвлениях. Для этого вычисляется размер всех инструкций от ветвления до ветви. Ветвления входящие в эту область могут быть длинными и короткими, таким образом необходимо вычислить также их размер. Например следующий код:

L1: ...

Jcc L2 *; Размер инструкции зависит от размера* ***Jcc L1****.*

...

Jcc L1 *; Размер этой инструкции зависит от размера* ***Jcc L2****.*

L2: ...

Необходимо задать изначальный размер ветвлений(в данном случае короткий). Тогда можно определить размер **Jcc L1**. Если смещение выходит за пределы смещения короткого ветвления, то изменяем размер ветвления на длинное и выполняем повторное вычисление смещений всех ветвлений графа.

Так как в линейном графе расположение описателей соответствует расположению описываемых инструкций в памяти, то выполняется последовательный проход по описателям, начиная от ветвления до ветви. Необходима процедура, вычисляющая смещение ветви относительно ветвления(**Branch** – описатель ветвления):

**CsCalculateBranchDisplacement**(Branch):

J = Branch

!Disp *; В этой переменной будем хранить смещение.*

Flow = Gp.BranchLink[J] *; Ссылка на ветвь.*

if Flow = J *; Ветвление на себя, завершаем обработку.*

> Save

fi

if J > Flow *; Так как граф линейный, то будем выполнять проход вверх по графу. Направление перечисления постоянно, выполняем обмен значениями базы и лимита части графа.*

J ⬄ Flow

fi

Do *; Цикл вычисления смещения,* ***Gp ~ [J % Flow]****.*

Type = Gp.Type[J] *; Тип описателя.*

if Type = TYPE\_LINE *; Описатель линейного блока.*

Disp + Gp.Size[J] *; Смещение увеличиваем на размер блока.*

elseif Type = TYPE\_CALL *; Процедурное ветвление. Размер постоянен.*

if BRANCH\_DEFINED\_FLAG[J] & Gp.DISCLOSURE\_CALL\_FLAG[J] *; Адрес ветви определён и она раскрыта.*

Disp + LDE(Call near rel) *; Увеличиваем смещение на размер инструкции(****+5*** *байт),* ***!Far****.*

else *; Ветвь не раскрыта или адрес её не определён.*

Disp + LDE(Gp.Address[J]) *; Используем* ***LDE*** *для определения размера инструкции.*

fi

elseif Type = TYPE\_JMP *; Безусловное ветвление.*

if !Gp.BRANCH\_IDLE\_FLAG[J] *; Не холостое ветвление[[9]](#footnote-9).*

if !Gp.BRANCH\_DEFINED\_FLAG[J] *; Адрес ветви не определён.*

Disp + LDE(Gp.Address[J])

else *; Адрес ветви определён.*

if Gp.BRANCH\_SIZE\_MASK[J] *; В зависимости от типа ветвления увеличиваем смещение.*

Disp + LDE(Jmp near rel) *;* ***+5*** *для длинного ветвления(****Near****).*

else

Disp + LDE(Jmp short rel) *;* ***+2*** *для короткого(****Short****).*

fi

fi

fi

elseif Type = TYPE\_JCC *; Условное ветвление.*

if !Gp.BRANCH\_IDLE\_FLAG[J] *; Не холостое ветвление.*

if Gp.BRANCH\_SIZE\_MASK[J]

Disp + LDE(Jcc near) *;* ***+6***

else

Disp + LDE(Jcc short) *;* ***+2***

fi

fi

fi

J + sizeof(Gp) *; Следующий описатель в линейном графе.*

Loop J < Flow *; Цикл от ветви до ветвления(или наоборот).*

Save:

J = Branch

Gp.BranchOffset[J] = Disp *; Сохраняем смещение в описателе.*

if Flow < J *; Определяем знак смещения.*

Gp.BRANCH\_DELTA\_SIGN[J]

else

!Gp.BRANCH\_DELTA\_SIGN[J]

fi

if Gp.Type[J] = TYPE\_JMP or TYPE\_JCC *; Определяем размер ветвления.*

if Disp > 0xFF/2 *;* ***+/-******0x7F***

Gp.BRANCH\_SIZE\_MASK[J] *;* ***Near****.*

else

!Gp.BRANCH\_SIZE\_MASK[J] *;* ***Short****.*

fi

fi

Теперь можем написать псевдокод вычисляющий размер всех ветвлений:

*; Изначально все ветвления короткие(****Short****). Если смещение более* ***0x7F*** *байт, то увеличиваем размер ветвления(****Near****) и повторно вызываем оптимизатор.*

**CsAdjustBranches**(GpBase, GpLimit):

Rst:

J = CsBase

Do *; Цикл прохода по всем описателям графа.*

Type = Gp.Type[J] ; Тип описателя.

if Type = TYPE\_CALL

if BRANCH\_DEFINED\_FLAG[J] & Gp.DISCLOSURE\_CALL\_FLAG[J] *; Адрес ветви определён и она описана.*

CsCalculateBranchDisplacement(J) *; Вычисляем смещение. Размер процедурного ветвления постоянен.*

fi

elseif Type = TYPE\_JMP  *; Безусловное ветвление.*

if BRANCH\_DEFINED\_FLAG[J] *; Адрес ветви определён.*

> Check

fi

elseif Type = TYPE\_JCC *; Условное ветвление.*

Check:

if !Gp.BRANCH\_IDLE\_FLAG[J] *; Пропускаем холостые ветвления.*

OldSize = Gp.BRANCH\_SIZE\_MASK[J] *; Предыдущее значение размера.*

CalculateBranchDisplacement(J) *; Вычисляем размер.*

if OldSize <> Gp.BRANCH\_SIZE\_MASK[J] *; Если после вычисления размера ветвления его размер изменится, то изменятся размер других ветвлений, поэтому рекурсивно вызываем оптимизатор(это не стековая рекурсия, а новый проход по графу).*

> Rst

fi

fi

fi

Next:

J + sizeof(Gp) *; Следующий описатель.*

Loop J < CsLimit  *; Выход из цикла по достижению конца графа.*

1. Если первый описатель вставляемого графа участвует в цикле, то на него есть ссылки подлежащие редиректу. Обычно ссылок на первый описатель графа нет. [↑](#footnote-ref-1)
2. Так как адрес ветвления не определён, будет сброшен флаг **BRANCH\_DEFINED\_FLAG**. [↑](#footnote-ref-2)
3. Значение условия **Cx**-ветвления будет индексировать описатель ветвления на примитив(напр. **je $Loopdne**) в графе, описывающем **CxReplaceTable()**. Так как при использовании префикса **0x67** условие изменится (размер регистра станет **16** бит, тоесть **Cx**), то для каждого условия вводится два блока примитивов. Таким образом(**J** – описатель заменяемого **Cx**-ветвления, граф линейный): **Gp(Je $X) = Gp(CxReplaceTable) + 2 \* sizeof(Gp) \* Gp.Condition[J]**. Если размер переопределён(**JCC\_X16\_MASK**), то **Gp(Je $X) + sizeof(Gp)**. Так как блок примитивов это ветвь **$X** в ветвлении **Je $X**, то раскрыв ссылку, определим адрес описателя блока примитивов: **Gp($X) = Gp(Je $X).BranchLink**. [↑](#footnote-ref-3)
4. Так как граф не изменяется(положение описателей относительно начала графа постоянно, в данном случае относительно второго описателя блока примитивов в **CxReplaceTable**), а перемещается в памяти(в конец графа для которого выполняется морфинг), то каждая из локальных ссылок(это например ветвление **jz @f** в блоке **$Loopdne**) вставляемого блока изменяется на одинаковое значение. Доказательство:

   Пусть **I** – ссылка на копируемый описатель в графе с базой **G**. Аналогично **I’** – ссылка на вставляемый описатель в графе с базой **G’**(описатель, который адресуется ссылкой **I** в графе **G** копируется в граф **G’** по адресу **I’**). Пусть смещение базы графа **G'** относительно графа **G** равно **Delta**, тоесть **G’ = G + Delta**. Смещение описателя **I** относительно начала графа **G** равно **I -** **G**. Так как граф, описывающий копируемый блок не изменяется, то **I -** **G = I' - G’**. Таким образом **I - G = I’ - (G + Delta)**. Отсюда **I' = I + Delta**. Прибавление константы не нарушит равенство **I' + sizeof(Gp) = I + Delta + sizeof(Gp)**. Так как следующий за **I** описатель **J** находится в линейном графе, то **J = I + sizeof(Gp)**. Учитывая предыдущее равенство, заключим что **J' = J + Delta**, тоесть разница смещений описателей в двух графах равна смещению баз графов. Для каждой ссылки выполняется поправка **I’.Flink = I.Flink + Delta**. Тоже и для обратной ссылки **Blink** и ссылки на ветвь **BranchLink**.

   . [↑](#footnote-ref-4)
5. Инструкции **pushf**/**popf** редко используются осевым кодом, поэтому вызывают подозрения и служат для создания **AV**-сигнатур. [↑](#footnote-ref-5)
6. Это условные ветвления **Jcc**, **Cx**-ветвления **Loopcc**, инструкции **Pushf**, **Setcc**, **Cmovcc, Rcl** и пр. [↑](#footnote-ref-6)
7. Учитывая что **Cx**-ветвление **Jecxz** не изменяет регистры, то может использоваться также, как не **Cx**-ветвление(например **Jz**). Для этого условие **J.BRANCH\_CX\_FLAG** и **J.Condition = !Ecx**. В описателе условие определено в **KitFlags**(маска **JCC\_TYPE\_MASK**): **KitFlags & JCC\_TYPE\_MASK = 3(JCC\_ECXZ)**. Так как установлен флаг **BRANCH\_CX\_FLAG**, то база для формирования опкода **JCX\_OPCODE\_BASE(0xE0)**, таким образом, опкод **0xE3**. Для инструкции **Jz** условие: **KitFlags & JCC\_TYPE\_MASK = 4(JCC\_Z)**. База выбирается в зависимости от типа ветвления(**Short/Near**, определяется флагом **BRANCH\_SIZE\_MASK**). Если ветвление короткое, то база **JCC\_SHORT\_OPCODE\_BASE = 0x70** и соответственно опкод **0x74**. При использовании префикса **0x67** (флаг **JCC\_X16\_MASK**) условие(мнемоника **Jcxz**) в инструкции **Jecxz** изменится(ветвление если **!Cx**, старшее слово в регистре **Ecx** не проверяется). Очевидно, что условие, при котором выполняется переход, определяется не опкодом, а набором флагов в описателе. [↑](#footnote-ref-7)
8. Условия это константы **JCC\_**. Находится в поле **KitFlags**(маска **JCC\_TYPE\_MASK**). [↑](#footnote-ref-8)
9. Процедурные ветвления не помечаются флагом **IDLE**, так как повышают **NL**. Только описатели безусловного и условного ветвлений содержат флаг **BRANCH\_IDLE\_FLAG**. Как видно из приведённого псевдокода, холостые ветвления игнорируются и присутствие их в линейном графе никак не влияет на смещение. Это позволяет не выполнять повторную конвертацию графа в линейный после оптимизации ветвлений(для удаления описателей). Так как такие описатели пропускаются, то определить следующий за ним описатель можно только в линейном графе(**Gp' = Gp + sizeof(Gp)**). В нелинейном графе ссылка на следующий описатель не может быть определена, так как она содержится в помеченном описателе, который пропускается. Также безусловное ветвление завершает ветвь и если оно помечено как удалённое нет возможности определить конец ветви. По этим причинам флаг не может использоваться при трассировке и связанных с ней операциях. [↑](#footnote-ref-9)