ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ И ТЕЛЕКОММУНИКАЦИИ
СПбГУТ))) Июнь 2016 Том 4 № 2 ISSN 2307-1303 http://www.itt.sut.ru
TELECOM IT
June 2016 Vol. 4 Iss. 2 ISSN 2307-1303 http://www.itt.sut.ru

УТИЛИТА ДЛЯ ПОИСКА УЯЗВИМОСТЕЙ В ПРОГРАММНОМ ОБЕСПЕЧЕНИИ ТЕЛЕКОММУНИКАЦИОННЫХ УСТРОЙСТВ МЕТОДОМ АЛГОРИТМИЗАЦИИ МАШИННОГО КОДА. ЧАСТЬ 2. ИНФОРМАЦИОННАЯ АРХИТЕКТУРА

К. Е. Израилов[1]

[1] СПбГУТ, Санкт-Петербург, 193232, Российская Федерация
Адрес для переписки: konstantin.izrailov@mail.ru

Информация о статье
УДК 004.4'422
Язык статьи – русский.
Поступила в редакцию 26.02.16, принята к печати 28.03.16.

Ссылка для цитирования: Израилов К. Е.: Утилита для поиска уязвимостей в программном обеспечении телекоммуникационных устройств методом алгоритмизации машинного кода. Часть 2. Информационная архитектура // Информационные технологии и телекоммуникации. 2016. Том 4. № 2. С. 86–104.

Аннотация
Предмет исследования. Статья продолжает цикл статей, посвященных авторскому методу алгоритмизации машинного кода телекоммуникационных устройств, имеющему автоматизацию основного этапа в виде утилиты. Вслед за функциональным, исследуется информационный слой утилиты, формализующий используемые в ней данные и задающий принципы их преобразования. **Метод.** Основные предпосылки к формализации данных были получены анализом созданной функциональной архитектуры утилиты и синтезом схемы эволюции «рабочих» данных в контексте используемой в методе модели. **Основные результаты.** Предложена информационная архитектура утилиты, включающая форматы входного ассемблерного и выходного алгоритмизированного кода, а также структуру внутренних представлений на всех стадиях работы. Приводятся создаваемые утилитой промежуточные данные для сквозного примера. **Практическая значимость.** Предложенная архитектура может быть использована для представления в программных средствах различных моделей программного кода, от их создания до оптимизации и проведения модельного эксперимента.

Ключевые слова
телекоммуникационные устройства, поиск уязвимостей, утилита алгоритмизации, информационная архитектура.

UTILITY FOR VULNERABILITY SEARCH IN SOFTWARE OF TELECOMMUNICATION DEVICES BY METHOD ALGORITHMIZATION OF MACHINE CODE. PART 2. INFORMATION ARCHITECTURE

K. Izrailov[1]

[1]SPbSUT, St. Petersburg, 193232, Russian Federation
Corresponding author: konstantin.izrailov@mail.ru

Article info
Article in Russian.
Received 26.02.16, accepted 28.03.16.

For citation: Izrailov K.: Utility for Vulnerability Search in Software of Telecommunication Devices by Method Algorithmization of Machine Code. Part 2. Information Architecture // Telecom IT. 2016. Vol. 4. Iss. 2. pp. 86–104 (in Russian).

Abstract
Research subject. The article continues the series of articles devoted to the author's method algorithmization machine code telecommunications devices, having the automation of the main stage as a utility. Following the functional utility researched information layer formalizing data used in it, and defining the principles of their conversion. **Method.** The main prerequisites for the formalization of data were obtained by analyze of a functional architecture and synthesis of schema evolution «workers» in the context of the data used in the method of the model. **Core results.** An information architecture of the utility, including formats input and output assembly algorithmic code, as well as the structure of the internal representation at all stages of the work. The intermediate data created by the utility for through example is at the article. **Practical relevance.** The proposed architecture can be used to represent in software different models of software code, from creation to optimization and modeling of the experiment.
Keywords
telecommunication devices, vulnerability search, algorithmization utility, information architecture.

Схема информационной архитектуры

Организацию и формализацию данных, необходимых для работы и взаимодействия всех функциональных модулей Утилиты алгоритмизации машинного кода [1, 2], автоматизирующей основной этап одноименного авторского метода [3, 4, 5, 6], определяет, так называемая, *информационная архитектура*. Ее основными элементами будут эволюционирующие «рабочие» данные Утилиты в виде отдельных внутренних представлений (далее – Представлений), описывающих различные стороны входного ассемблерного кода (далее – АК), а их связями – зависимость получения одних Представлений из других в процессе выполнения стадий Утилиты. Ее информационная архитектура в схематичном виде приведена на рис. 1.

Как хорошо видно по схеме, разделение внутренних Представлений по стадиям Утилиты полностью соответствует аналогичному для функциональ-

ной архитектуры [1]. Это абсолютно закономерно, поскольку каждая стадия функционирует согласно составляющим ее модулям, взаимодействующим посредством внутренних представлений программного кода.

Согласно предлагаемой информационной архитектуре существуют следующие преобразования между внутренними Представлениями Утилиты [2]. В фазе Front-End, определяемой единственной Стадией 1, производится разбор входного ассемблерного кода (и корректировок алгоритмизации) с последующим построением внутреннего Представления дерева абстрактного синтаксиса, содержащего входной код в базово-формализованном виде. Затем выполняются стадии фазы Middle-End.

На Стадии 2 по дереву абстрактного синтаксиса собирается информация обо всех подпрограммах и их вызовах и строится граф взаимных вызовов подпрограмм, основным назначением которого является подготовка к построению дерева областей переменных и подпрограмм входного ассемблера. Далее, во всей фазе Middle-End, используется такое разделение, при котором все внутренние Представления описывают лишь отдельную подпрограмму, что позволяет реализовать алгоритмы их обработки более обобщенно. Полный же список таких Представлений для всех подпрограмм может быть получен путем обхода дерева областей. Также, на Стадии 2 каждое поддерево абстрактного синтаксиса, относящееся к отдельной подпрограмме, преобразуется к графу ее потока управления – сначала классическому, состоящему из базовых блоков, а затем и к «жесткому» – имеющему дополнительные служебные узлы для унификации управляющих конструкций. На Стадии 3 производится структуризация жесткого графа управления – перевод в форму Насси-Шнейдермана – позволяющая описывать входной АК в виде дерева; безусловные переходы GOTO в нем считаются исключениями и задаются с помощью пары узлов «переход-метка». Также, на этой стадии строится граф потока данных, используемый в дальнейшем. На Стадии 4, с использованием графа зависимостей между вычислениями, полученного по графу потока данных, производится оптимизация дерева потока управления. На Стадии 5, являющейся завершающей в фазе Middle-End, производится генерация внутреннего Представления псевдо-кода, описывающего входной АК; дерево такого псевдо-кода является окончательным и готовым для генерации по нему алгоритмов кода. Также, дерево содержит узлы с отметками об уязвимостях, созданные на основании соответствующего Представления.

На единственной Стадии 6 фазы Back-End генерируется текстовое описание алгоритмов кода по внутреннему Представлению в виде дерева псевдо-кода тривиальным способом; информация об уязвимостях в коде также добавляется в выходное описание.

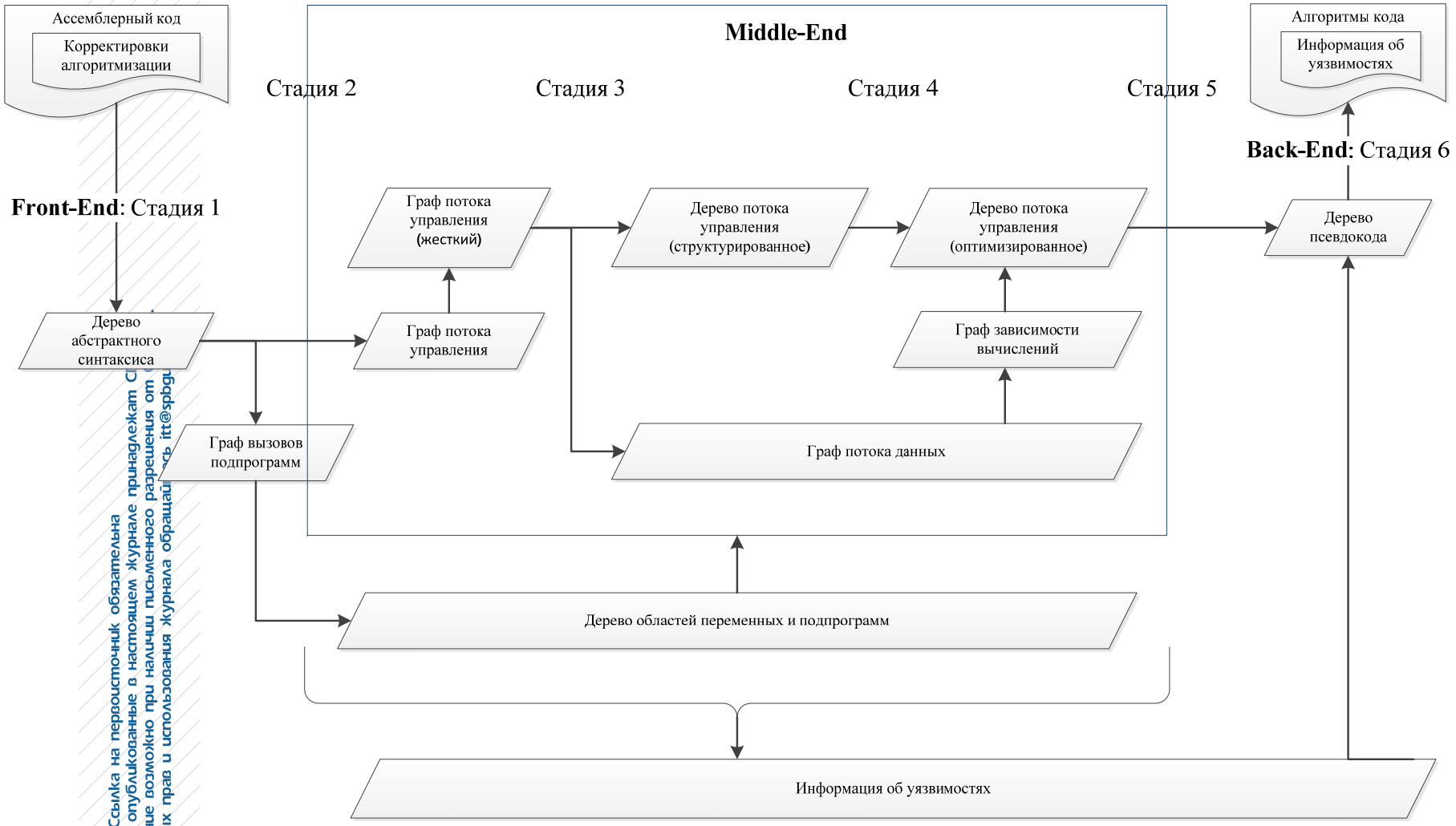

Рис. 1. Информационная архитектура Утилиты (схема)

Внутреннее Представление информация об уязвимостях не имеет отдельного четко-выделенного контейнера (списка, дерева, графа), поскольку отметки об уязвимостях добавляются в граф или дерево потока управления в момент обнаружения посредством специальных узлов и хранятся в нем вплоть до генерации алгоритмов кода.

Опишем элементы информационной архитектуры (а именно – эволюционирующие данные в виде отдельных Представлений кода) более детально. Для примеров Представлений будем использовать простейшую функцию нахождения максимального их двух чисел, АК которой для процессора PowerPC (размером 24 байта) с комментариями на языке С имеет следующий вид:

```
// int max2(int X, int Y)
              max2:           // { /* X(R3), Y(R4), T(R5) */
0x00000000:   cmpw r3, r4     //   if(X > Y)
0x00000004:   ble label_1     //   {
0x00000008:   mr r5, r3       //     T = X;
0x0000000C:   b label_2       //   }
              label_1:        //   else {
0x00000010:   mr r5, r4       //     T  = Y;
              label_2:        //   }
0x00000014:   mr r3, r5       //   return T;
0x00000018:   blr             // }
```

АК описывает функцию, принимающую на входе 2 параметра (X, Y) посредством регистров (R3, R4). После сравнения параметров, значение максимального из них присваивается локальной переменной *T* в регистре R5, которая впоследствии также присваивается регистру возвращаемого функцией значения – R3.

Входной ассемблерный код

Поскольку Утилита на вход должна принимать помимо АК еще и корректировки алгоритмизации, то целесообразно использовать специально-расширенный в этих интересах синтаксис языка ассемблер. При этом, исходя из того, что АК генерируется на предыдущем этапе Метода [3, 4, 5, 6] автоматически – с помощью скриптов в продукте IDAPro – это реализуется достаточно простым способом. Также, расширенный синтаксис ассемблера сможет улучшить его ручное создание и анализ экспертом, например, в интересах отладки.

Опишем расширение синтаксиса ассемблера на примерах без приведения формальной грамматики, поскольку оно является интуитивно-понятным.

Во-первых, АК подпрограмм размещается в С-подобных блоках «{…}» с указанием имени подпрограммы «funct()». Так, для текущего примера код на расширенном синтаксисе ассемблера будет иметь следующий вид:

```
max2(){
    0x00000000:  cmpw r3, r4
    ...
0x00000018:  blr
}
```

Примечание. Здесь и далее знак «…» означает пропущенный в интересах уплотнения записи код, а не специальный элемент синтаксиса.

Утилитой такой текст будет восприниматься, как АК подпрограммы с именем «max2».

Во-вторых, имеется возможность корректировки алгоритмизации путем явного задания частичного или полного списка параметров подпрограммы с помощью указания имени параметра и регистра процессора, через который он передается. Так, для используемого примера, Эксперт может указать, что подпрограмма «max2» принимает 2 параметра – «X» в регистре «R3 и «Y» в регистре R4 – следующим образом:

```
max2(X:r3, Y:r4){
    0x00000000:  cmpw r3, r4
...
    0x00000018:  blr
}
```

Здесь корректировкой считается именно С-подобное задание параметров подпрограммы: «X: r3, Y: r4». Корректировка будет использоваться Утилитой для пользовательского уточнения сигнатуры.

Также Эксперт может указать лишь часть из параметров, оставив возможность определения остальных Утилите – с помощью элемента синтаксиса «…» вместо отсутствующих параметров – следующим образом:

```
max2(X:r3, ...){
    0x00000000:  cmpw r3, r4
...
    0x00000018:  blr
}
```

Таким образом, Утилита в обязательном порядке отведет лишь одному параметру подпрограммы с именем «X» регистр R3; остальные же параметры и их регистры Утилита определит самостоятельно. Также, возможно указание подпрограммы без параметров с помощью ключевого слова «none»:

```
max2(none){
    0x00000000:  cmpw r3, r4
...
    0x00000018:  blr
}
```

Синтаксически, такая корректировка является корректной, хотя и приведет к неверной алгоритмизации Утилитой.

В-третьих, возможно явное указание регистров, используемых для возвращаемых значений подпрограммы, аналогичным с параметрами способом, но перед именем подпрограммы, следующим образом:

```
(r3) max2(X:r3, Y:r4){
    0x00000000:  cmpw r3, r4
...
    0x00000018:  blr
}
```

Такая запись сигнатуры подпрограммы указывает Утилите, что результат вычислений в подпрограмме «max2» возвращается через регистр R3. Аналогично указанию параметров подпрограммы, возможно использование ключевого слова «none», означающего в данном контексте, что функция не возвращает никаких значений.

В-четвертых, возможно явное указание абсолютного адреса подпрограммы, используемого различными оптимизациями, с помощью символа «&» и самого адреса между группой параметров подпрограммы и началом ее блока:

```
(r3) max2(X:r3, Y:r4) & 0x00001000 {
    0x00000000:  cmpw r3, r4
...
    0x00000018:  blr
}
```

Данная запись означает, что подпрограмма «max2» расположена по адресу 0x00001234. Необходимо отметить, что адреса, указанные перед инструкциями в блоке тела подпрограммы (для текущего примера с 0x00000000 до 0x00000018 с шагом 4), генерируются автоматически скриптами для продукта IDA Pro, носят лишь стилистический характер и никак не используются Утилитой.

В-пятых, возможно явное указание глобальных переменных с опциональным заданием их абсолютного адреса (что используется различными оптимизациями) следующим образом: в начале указывается ключевое слово «var», затем идет имя глобальной переменной с опциональными символом «&» и ее абсолютным адресом, и завершающий символ«;». Пример задания глобальной переменной GLOB по адресу 0x2000 является следующим:

```
varGLOB& 0x000020000;
```

В текущем примере подпрограмма «max2» не использует глобальных переменных, поэтому данная корректировка далее никак не упоминается.

Также, для полноценного представления возможностей работы Утилиты будем использовать изначальный пример без каких-либо корректировок, т. е. подпрограмму с сигнатурой «max2()».

Внутренние представления

После разбора текстового ассемблерного Представления вплоть до генерации текстового алгоритмизированного весь код в Утилите имеет формализованный вид согласно различным Представлениям, приведенным на схеме архитектуры (см. рис. 1). Назначение, форма и содержание каждого такого Представления и их конкретный «снимок» для исходного примера приводятся далее.

Примеры деревьев будут представлены в виде текстовых строк (генерируемых в отладочном выводе Утилиты), каждая из которых содержит описание одной вершины дерева, пробельные отступы означают глубину вложенности элемента, а более ранний в тексте элемент с меньшим отступом – родительский узел дерева. Примеры графов будут представлены в виде графических изобра-

жений, визуализированных с помощью программы GraphViz[1]. Программа принимает на вход данные в формате DOT, используемом и дл отладочного вывода графов [7].

Дерево абстрактного синтаксиса

Для базовой формализации входного АК традиционно в компиляторах и подобных программных средствах применяется дерево абстрактного синтаксиса (имеющее аббревиатуру – AST, *от англ.* AbstractSyntaxTree). Внутренние вершины дерева, как правило, сопоставляются с операторами языка, а листья – с опциональными операндами (переменными, константами, результатами вычисления вложенных операторов). Также, возможно задание служебных не языковых конструкций, таких как списки, временные узлы и т. п. Классическое дерево абстрактного синтаксиса (что также вытекает и из его названия) отражает специфику входного синтаксиса и в последствии преобразуется к дереву внутреннего представления, считающемуся полностью независимым от входного языка (как, кстати и от выходного). Тем не менее, по причине простоты синтаксиса ассемблера и его инструкций, возможно и целесообразно строить дерево абстрактного синтаксиса сразу же независимым от ассемблерных операций и регистров, т. е. переводя их в разряд *обезличенных* операций над переменными; что и было реализовано в Утилите. Пример дерева абстрактного синтаксиса для текущего примера следующий:

```
IrRoot()
IrList()      // GlobalDecl
IrList()      // Functs
    IrName('max2')
      IrList()// Args
        IrEmpty()
      IrList()// Rets
        IrEmpty()
      IrEmpty()
      IrReg('lr'), id=41, exprInfo=(NULL)
      IrList()// Code
        IrLabel('max2'), name='max2'
        IrOperation('cmpw'), kind='assign'
          IrOperation('cmpw'), kind='bit access'
            IrReg('cr'), id=32, exprInfo=(NULL)
            IrInteger(''), value=0, kind=dec, fSigned=true
          IrCond('cmpw'), kind='?true'
            IrCond('cmpw'), kind='<'
              IrReg('r3'), id=3, exprInfo=(NULL)
              IrReg('r4'), id=4, exprInfo=(NULL)
        IrOperation('cmpw'), kind='assign'
          IrOperation('cmpw'), kind='bit access'
            IrReg('cr'), id=32, exprInfo=(NULL)
            IrInteger(''), value=1, kind=dec, fSigned=true
          IrCond('cmpw'), kind='?true'
            IrCond('cmpw'), kind='>'
              IrReg('r3'), id=3, exprInfo=(NULL)
              IrReg('r4'), id=4, exprInfo=(NULL)
```

[1] Graphviz – Graph Visualization Software / Сайт программы GraphViz. URL: http://www.graphviz.org/

```
IrOperation('cmpw'), kind='assign'
  IrOperation('cmpw'), kind='bit access'
    IrReg('cr'), id=32, exprInfo=(NULL)
    IrInteger(''), value=2, kind=dec, fSigned=true
  IrCond('cmpw'), kind='?true'
    IrCond('cmpw'), kind='=='
      IrReg('r3'), id=3, exprInfo=(NULL)
      IrReg('r4'), id=4, exprInfo=(NULL)
IrBranch('ble')
  IrCond('ble'), kind='?false'
    IrOperation('ble'), kind='bit access'
      IrReg('cr'), id=32, exprInfo=(NULL)
      IrInteger(''), value=1, kind=dec, fSigned=true
  IrLabel('label_1'), name='label_1'
IrOperation('mr'), kind='assign'
  IrReg('r5'), id=5, exprInfo=(NULL)
  IrReg('r3'), id=3, exprInfo=(NULL)
IrBranch('b')
  IrEmpty()
  IrLabel('label_2'), name='label_2'
IrLabel('label_1'), name='label_1'
IrOperation('mr'), kind='assign'
  IrReg('r5'), id=5, exprInfo=(NULL)
  IrReg('r4'), id=4, exprInfo=(NULL)
IrLabel('label_2'), name='label_2'
IrOperation('mr'), kind='assign'
  IrReg('r3'), id=3, exprInfo=(NULL)
  IrReg('r5'), id=5, exprInfo=(NULL)
IrBranch('blr')
  IrEmpty()
  IrReg('lr'), id=41, exprInfo=(NULL)
```

При детальном рассмотрении дерева хорошо видно, что оно практически полностью повторяет весь АК, но сделав его структуризацию (выделив подпрограмму, операторы и операнды, блоки переходов управления) и используя независимые от процессора термины (IrReg может трактоваться как некая переменная; IrOperation имеет свойство kind, задающее обобщенный смысл операции; IrBranch задает любые переходы, как условные, так и безусловные, и даже выход из подпрограммы, и т. п.). При этом в дереве отсутствует, так называемый, «лексический мусор», такой, как пробелы, запятые, неиспользуемые адреса инструкций и т. п.

Граф потока управления

Для воспроизводства топологии алгоритма программы строится ориентированный граф потока управления (имеющий аббревиатуру – CFG, *от англ.* ControlFlowGraph). Узлы графа соответствуют базовым блокам – прямолинейным участкам кода без операций передачи точек получения управления, с тем исключением, что каждый базовый блок начинается с метки для получения управления и заканчивается инструкцией передачи. Направленные дуги в графе задают переходы между блоками согласно инструкциям. Каждый такой граф для подпрограммы начинается с входного блока и заканчивается выходным. Таким образом, граф описывает все множество путей выполнения программы. Пример графа потока управления для текущего примера приведен на рис. 2*а*.

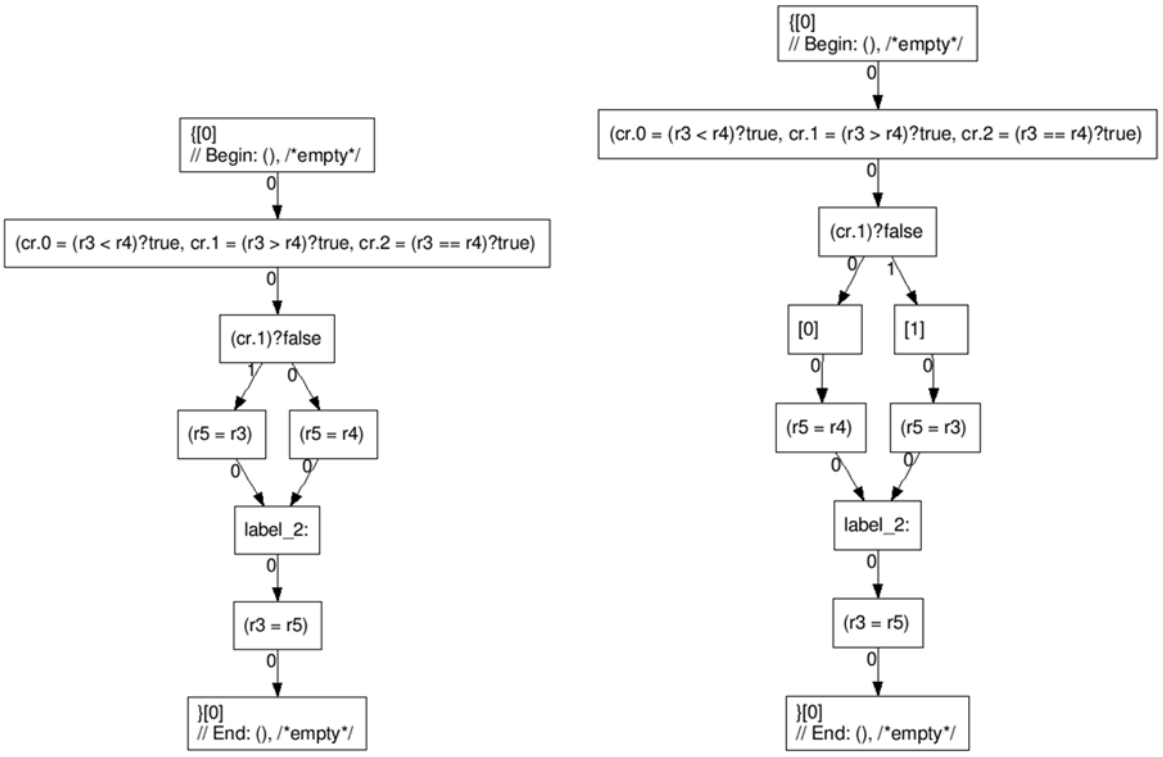

Рис. 2. Пример графа потока управления для функции max2()

Как хорошо видно из рисунка, граф имеет входной блок (с комментарием «// Begin») – соответствующий началу подпрограммы, и выходной блок (с комментарием «// End») – соответствующий завершению подпрограммы, т. е. выполнению инструкции BLR для PowerPC (или операции «return» в языке C). Также, на графе видны два пути выполнения программы в зависимости от значения 1-го бита регистра CR («(cr.1)?false»), определяемого операциями в предыдущем блоке графа по результату сравнения «R3 > R4».

Граф потока управления («жесткий»)

«Жесткий» граф потока управления не имеет применения в практике разработки компиляторов, хотя в последних и применяются иные гибридные решения. Основное назначения графа – в стандартизации топологии для упрощения алгоритмов последующей обработки; для этого в «не-жесткий» граф добавляются служебные узлы, являющиеся первыми в каждой из веток условного выполнения, даже если последние не содержат инструкций. Это приводит к тому, что у каждой управляющей конструкции для условного выполнения всегда есть два нижележащих служебных узла, которые не могут быть удалены в дальнейшем оптимизациями, тем самым разрушая структуру. Узлы позволяют реализовывать однотипные алгоритмы обработки таких конструкций, не заботясь о роде и сохранности нижележащих узлов. Пример «жесткого» графа потока управления для текущего примера приведен на рис. 2*б*.

Согласно изображению графа, он отличается от «не-жесткого» лишь наличием служебных узлов («[0]» и «[1]»), расположенных в каждой ветке условного выполнения.

Граф вызовов подпрограмм

Данный ориентированный граф описывает для каждой подпрограммы вызовы всех других (имеет распространенное англоязычное название – CallGraph) с помощью узлов, задающих подпрограммы, а также дуг между ними, определяющих вызовы других. Для текущего примера граф состоит из одного узла, соответствующего подпрограмме «max2», поскольку она никаких вызовов других подпрограмм не делает.

Дерево областей переменных и подпрограмм

Данное дерево задает программные области, в которых расположены переменные (как глобальные, так и локальные) и подпрограммы. Очевидно, что глобальные переменные и подпрограммы в дереве расположены на одном уровне, а локальные – на подуровне соответствующей подпрограммы. Отметим, что в отличие от языка С, к локальным переменным относятся как входные/выходные параметры, так и регистр с адресом возврата из подпрограммы (для PowerPC это LR). Шаблон дерева приведен на рис. 3.

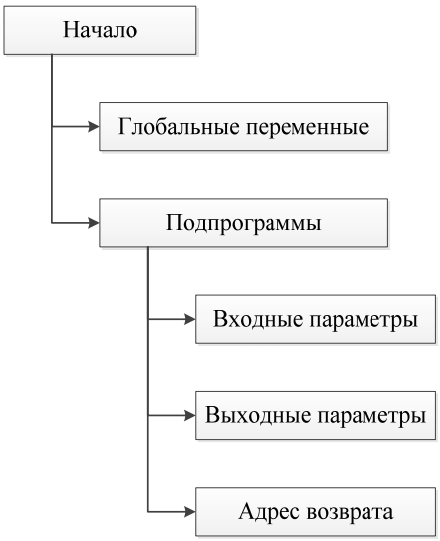

Рис. 3. Пример дерева областей переменных и подпрограмм для функции max2()

По причине тривиальности дерева его отладочный вид представляется в текстовом виде и для текущего примера следующий:

```
// Declaration of Global Variables
// Declaration of Subprograms
(...) max2(...) lr: 41{}
```

Согласно его описанию, дерево состоит из области глобальных переменных и области подпрограмм, локальные переменные которых заданы в их сигнатуре. По сути, текстовое описание дерева аналогично заголовочному файлу,

используемому в языке С. Например, такое созданное в результате работы Стадии 2 Утилиты дерево описывает подпрограмму «max2» с неизвестным списком входящих/выходящих параметров, у которой регистр LR (тождественный регистру R41) хранит адрес возврата из подпрограммы.

Информация об уязвимостях

Для хранения информации о найденных уязвимостях используются узлы деревьев и графов, которые могут хранить произвольные данные. В частности, существуют специальные служебные узлы, создаваемые в процессе построения дерева абстрактного синтаксиса и его анализа при обнаружении подозрительных на уязвимости мест; эти узлы сохранятся и при последующих преобразованиях. Таким образом, конечное дерево псевдо-кода уже хранит в себе все пометки об уязвимостях и их точных местах обнаружения, которые генерируются в текстовое алгоритмизированное Представление. Для указания уязвимостей, не имеющих жесткой привязки к коду, возможно добавление соответствующих узлов в близкие логически-связанные узлы (например, родительский блок) или к верхнему узлу дерева.

Дерево потока управления (структурированное)

В процессе выполнения алгоритмизации граф потока управления преобразуется Утилитой в соответствующее дерево, основным отличием которого является *размыкание* зацикленности на графе, что приводит к структуризации Представления программы и ее подобия диаграммам Насси-Шнейдермана. В случае невозможности избавления от безусловных переходов, они хранятся в дереве, как ссылки одних элементов (очевидно, листьев деревьев) на другие. Такое преобразование производится путем выделения в графе структурных метаданных [8, 9, 10], их *умной (*в смысле применения нетривиальных авторских алгоритмов) пересортировке и построения нового дерева на их базе. Наличие признака структурированности в названии дерева подчеркивает соответствующее произведенное действие. Пример дерева потока управления для текущего примера приведен на рис. 4.

Результат визуального сравнения дерева потока управления с графом позволяет утверждать, что Представление кода стало более структурированным, хотя некоторые узлы и хранят ссылки (аналоги безусловного перехода) на другие.

Дерево потока управления (оптимизированное)

На Стадии 4 производятся действия, направленные на «улучшение» структурированного дерева потока управления, приводя его к оптимизированному; соответствующий функционал приведен в описании модуля оптимизации [1]. Также, оптимизируются и выражения, которые хотя и не влияют на топологию дерева, но их упрощение и анализ может потенциально приводить к удалению неисполняемых веток и отработке других оптимизаций, что впоследствии приведет к перестроениям дерева. Пример оптимизированного дерева потока управления для текущего примера приведен на рис. 5.

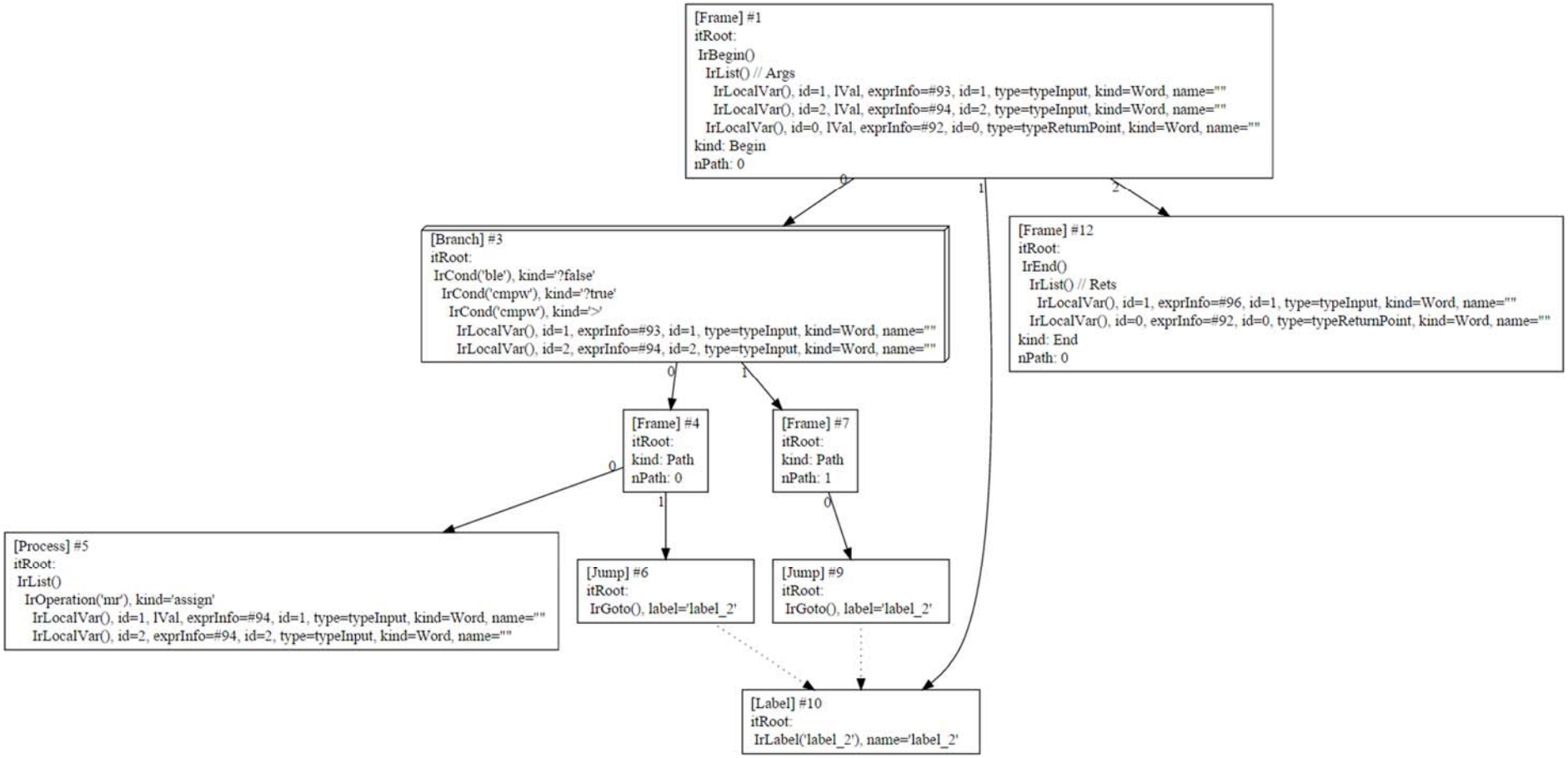

Рис. 4. Пример дерева потока управления для функции max2()

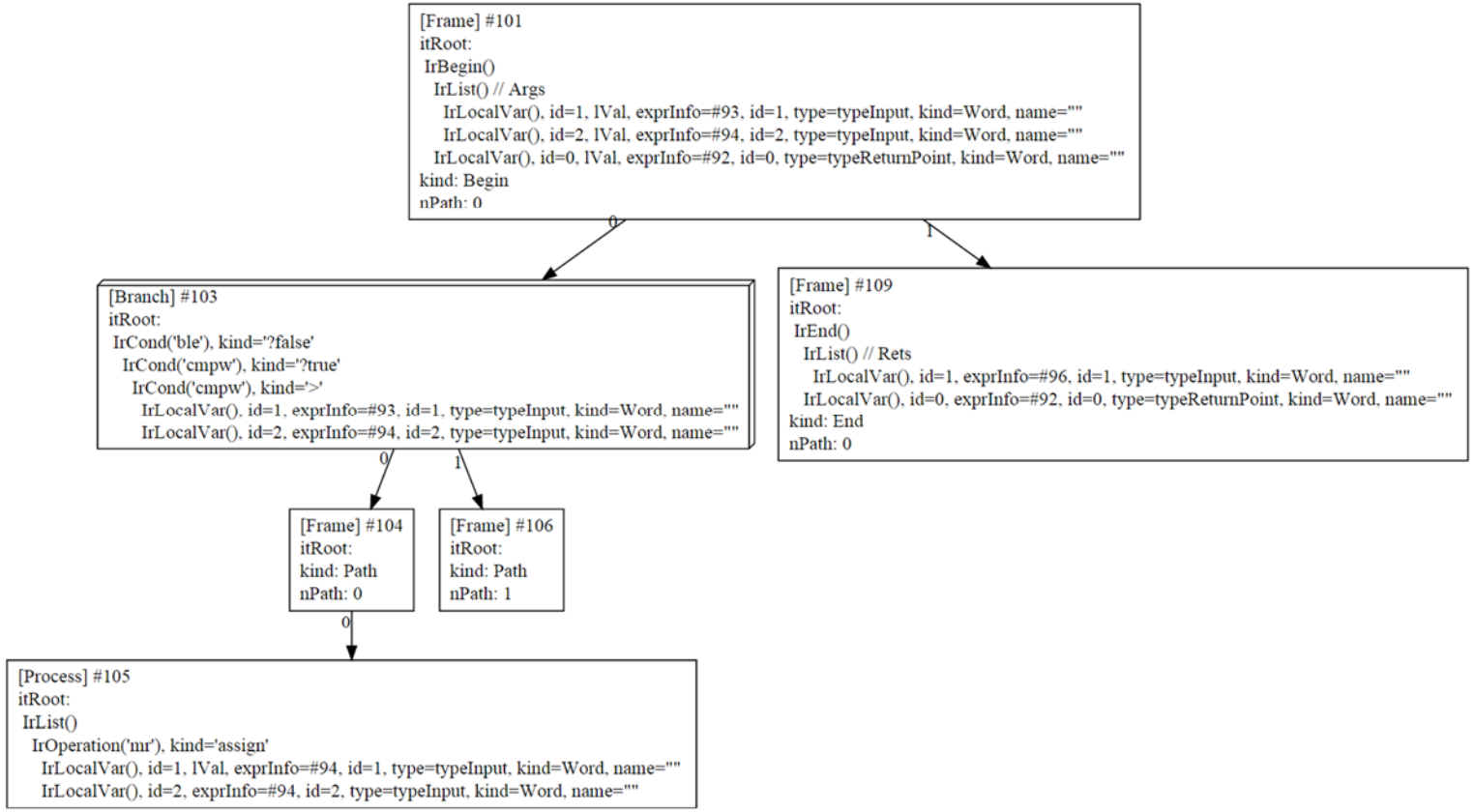

Рис. 5. Пример оптимизированного дерева потока управления для функции max2()

Как хорошо видно из рисунка, оптимизированное дерево подобно предыдущему – структурному, но выглядит более компактно, не имеет ссылок для безусловных переходов, а также лишних программных меток.

Граф потока данных

Граф топологически аналогичен потоку управления, но предназначен для содержания информации о качественных значениях переменных – не конкретных значений, а неких условных идентификаторов их множества и времен их жизни – областях узлов графа, в которых значение переменной инициализируется, храниться или используется. Операции в узлах графа определяют изменения значений переменных, а точки схождения веток – их объединения. Также, первая инициализация переменной в данной ветке графа означает начало ее жизни, а последняя – конец. Верность этой информации определяется внесенными вручную корректировками и успешностью работы Модуля уточнения сигнатуры подпрограммы [1]. Пример оптимизированного графа потока данных для текущего примера приведен на рис. 6.

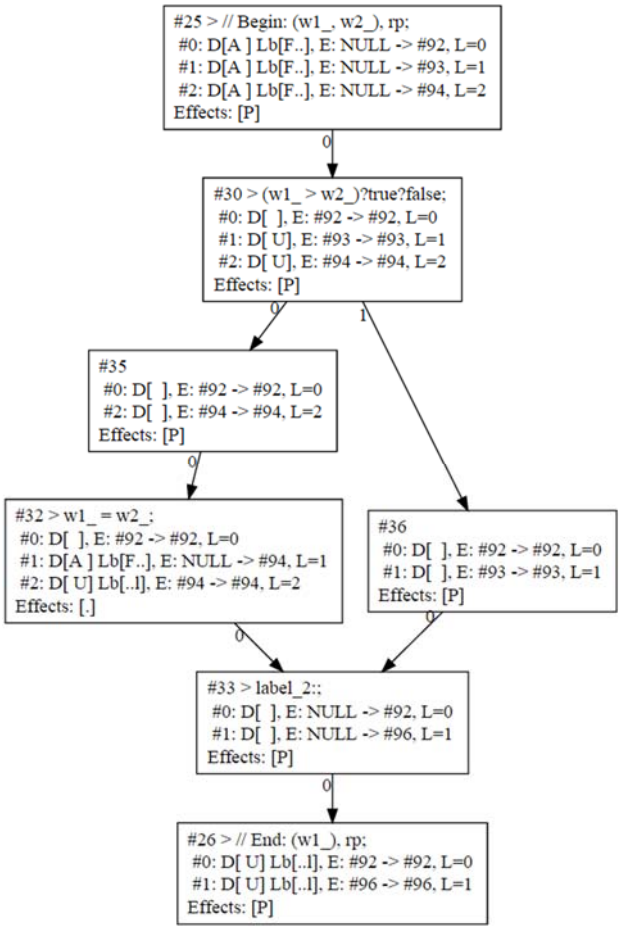

Рис. 6. Пример графа потока данных для функции max2()

Согласно графическому виду графа потока данных, внешне он полностью аналогичен графу потока управления, однако его узлы содержат иную информацию – исключительно о переменных кода. Такая информация закодирована помощью специальных структур и флагов, основные из которых на изображе-

нии графа имеют следующие обозначения: #N – идентификатор переменной, D[] – массив флагов операций над переменной (A или Assign – присваивание, U или Usage – использование), Lb – массив флагов времени жизни переменной (F – начало жизни, l – конец жизни), E: #N1 -> #N2 – идентификатор значения переменной с присвоением его другому, L = N – идентификатор времени жизни переменной, Effects – массив флагов эффектов узлов (P или Permanent – узел служебный и не может быть удален).

Граф зависимости вычислений

Граф предназначен для хранения информации о связи между вычисляемыми значениями. Такая информация используется для распределения значений, хранящихся на множестве регистров начального ассемблера, на более компактное множество переменных конечного восстановленного алгоритма, а также для упрощения математических выражений – т. е. используется в процессе оптимизации. Узлы графа хранят идентификаторы выражений, а дуги – связи между их вычислениями. Пример оптимизированного дерева потока управления для текущего примера приведен на рис. 7.

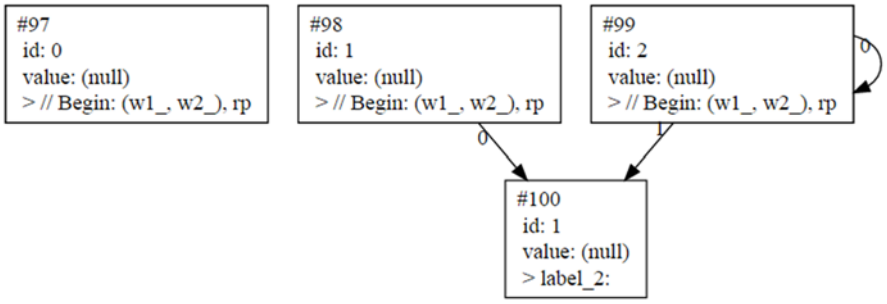

Рис. 7. Пример графа зависимостей вычислений для функции max2()

Необходимо отметить, что идентификаторы выражений на приведенных графе потока данных и уна графе зависимостей вычислений не совпадают, поскольку они в процессе оптимизаций неоднократно перестраивается с использованием новых значений идентификаторов. Тем не менее, можно определить следующее соответствие между идентификаторами выражений этих двух графов: #92 = #97, #93 = #98, #94 = #99, #96 = #100.

Дерево псевдокода

Дерево описывает восстановленные алгоритмы в виде псевдокода, сгенерированного по оптимизированному дереву потока управления и прошедшему лаконизацию (перестроение и замену узлов дерева с целью получения вида, лучше воспринимаемого человеком). Представление дерева можно считать конечным, полностью готовым к генерации текстового описания (или, например, графического, как потенциального развития Утилиты). Пример дерева псевдокода для текущего примера следующий.

```
IrList() // Root
IrFunction()
IrIdent('max2')
    IrList()     // Args
```

```
  IrLocalVar(), id=1, lVal, exprInfo=#93, id=1, type=typeInput, kind=Word, name=""
  IrLocalVar(), id=2, lVal, exprInfo=#94, id=2, type=typeInput, kind=Word, name=""
IrList()     // Rets
  IrReg(), id=3, exprInfo=(NULL)
IrLocalVar(), id=0, lVal, exprInfo=#92, id=0, type=typeReturnPoint, kind=Word, name=""
IrBlock()
  IrIfElse()
    IrCond('ble'), kind='?true'
      IrCond('cmpw'), kind='<='
        IrLocalVar(), id=1, exprInfo=#93, id=1, type=typeInput, kind=Word, name=""
        IrLocalVar(), id=2, exprInfo=#94, id=2, type=typeInput, kind=Word, name=""
    IrBlock()
      IrOperation('mr'), kind='assign'
        IrLocalVar(), id=1, lVal, exprInfo=#94, id=1, type=typeInput, kind=Word, name=""
        IrLocalVar(), id=2, exprInfo=#94, id=2, type=typeInput, kind=Word, name=""
    IrBlock()
  IrReturn()
    IrList() // Rets
      IrLocalVar(), id=1, exprInfo=#96, id=1, type=typeInput, kind=Word, name=""
    IrLocalVar(), id=0, exprInfo=#92, id=0, type=typeReturnPoint, kind=Word, name=""
```

В дереве присутствуют новые типы узлов, зависимые от конечного языка представления алгоритмов и не встречающиеся в АК (как и в его дереве абстрактного синтаксиса), такие как: IrIfElse – конструкция веток условного перехода «IF () THEN {} ELSE {}»; IrReturn – возврат из подпрограммы; IrLocalVar – локальные переменные (с типами: typeInput – для входных параметров и typeReturnPoint – для адреса возврата подпрограммы). Также очевидно, что данное дерево стало значительно меньше дерева абстрактного синтаксиса (24 строки против 59), притом с сокращением используемых переменных (2 против 3-х). Таким образом, псевдокод можно считать более структурированным и компактным, чем ассемблерный.

Вспомогательные данные

В Утилите используются также вспомогательные данные, обеспечивающие построение и обработку внутренних Представлений. Во-первых, это хранилище глобальных настроек, определяющих визуальный стиль восстановленных алгоритмов, генерацию комментариев и информации об используемых процессорных регистрах, выбор внутренних представлений для отладочного вывода. Во-вторых, это цвета раскраски узлов, используемые как для самой работы алгоритмов, так и для визуального отображения результатов их работы на графах и деревьях. В-третьих, это таблица регистров процессора входного ассемблера, содержащая их имена со свойствами и используемая в процессе синтаксического анализа. И, в-четвертых, это объект с текущей версией Утилиты и историей основных предыдущих изменений.

Продолжение следует

Литература

1. Буйневич М. В., Израилов К. Е. Утилита для поиска уязвимостей в программном обеспечении телекоммуникационных устройств методом алгоритмизации машинного кода. Часть 1. Функциональная архитектура // Информационные технологии и телекоммуникации. 2016. Том 4. № 1. С. 115–130. URL: http://sut.ru/doci/nauka/review/20161/115-130.pdf

2. Буйневич М. В., Израилов К. Е. Автоматизированное средство алгоритмизации машинного кода телекоммуникационных устройств // Телекоммуникации. 2013. № 6. С. 2–9.

3. Буйневич М. В., Израилов К. Е. Метод алгоритмизации машинного кода телекоммуникационных устройств // Телекоммуникации. 2012. № 12. С. 2–6.

4. Buinevich M., Izrailov K., Vladyko A. Method for partial recovering source code of telecommunication devices for vulnerability search // 17th International Conference On Advanced Communications Technology (ICACT). 2015. pp. 76–80.

5. Buinevich M., Izrailov K., Vladyko A. Method and utility for recovering code algorithms of telecommunication devices for vulnerability search // Всборнике: 16th International Conference on Advanced Communication Technology (ICACT). 2014. С. 172–176.

6. Buinevich M., Izrailov K., Vladyko A. Method and prototype of utility for partial recovering source code for low-level and medium-level vulnerability search // 18th International Conference on Advanced Communication Technology (ICACT). 2016. pp. 700–707.

7. Израилов К. Е. Расширение языка «*C*» для описания алгоритмов кода телекоммуникационных устройств // Информационные технологии и телекоммуникации. 2013. № 2 (2). С. 21–31. URL: http://www.sut.ru/doci/nauka/review/2-13.pdf

8. Buinevich M., Izrailov K., Vladyko A. The life cycle of vulnerabilities in the representations of software for telecommunication devices // 18th International Conference on Advanced Communication Technology (ICACT). 2016. pp. 430–435.

9. Буйневич М. В., Щербаков О. В., Израилов К. Е. Модель машинного кода, специализированная для поиска уязвимостей // Вестник Воронежского института ГПС МЧС России. 2014. № 2 (11). С. 46–51.

10. Буйневич М. В., Щербаков О. В., Израилов К. Е. Структурная модель машинного кода, специализированная для поиска уязвимостей в программном обеспечении автоматизированных систем управления // Проблемы управления рисками в техносфере. 2014. № 3 (31). С. 68–74.

References

1. Buinevich M. V., Izrailov K. E. Utility for Searching Vulnerabilities in Software of Telecommunication Devices by Method of Algorithmization Machine Code. Part 1. Functional Architecture // Telecom IT. 2016. Vol. 4. no. 1. pp. 115–130. URL: http://sut.ru/doci/nauka/review/20161/115-130.pdf

2. Buinevich M. V., Izrailov K. E. Automated Tool of Algorithmization Machine Code of Telecommunication Devices // Telecommunications. 2013. no. 6. pp. 2–9.

3. Buinevich M. V., Izrailov K. E. Method of Algorithmization Machine Code of Telecommunication Devices // Telecommunications. 2012. no. 12. pp. 2–6.

4. Buinevich M., Izrailov K., Vladyko A. Method for Partial Recovering Source Code of Telecommunication Devices for Vulnerability Search // 17th International Conference On Advanced Communications Technology (ICACT). 2015. pp. 76–80.

5. Buinevich M., Izrailov K., Vladyko A. Method and Utility for Recovering Code Algorithms of telecommunication devices for vulnerability search // 16th International Conference on Advanced Communication Technology (ICACT). 2014. pp. 172–176.

6. Buinevich M., Izrailov K., Vladyko A. Method and Prototype of Utility for Partial Recovering Source Code for Low-Level and Medium-Level Vulnerability Search // 18th International Conference on Advanced Communication Technology (ICACT). 2016. pp. 700–707.

7. Izrailov K. E. The «C» Language Extension for Describe Code Algorithms of Telecommunication Devices // Telecom IT. 2013. no. 2 (2). pp. 21–31. URL: http://www.sut.ru/doci/nauka/review/2-13.pdf

8. Buinevich M., Izrailov K., Vladyko A. The Life Cycle of Vulnerabilities in the Representations of Software for Telecommunication Devices // 18th International Conference on Advanced Communication Technology (ICACT). 2016. pp. 430–435.

9. Buinevich M. V., Sherbakov O. V., Izrailov K. E. Model of Machine Code Specialized for Vulnerabilities Search // Vestnik Voronezhskogo insituta GPS MCHS Rossii. 2014. no. 2 (11). pp. 46–51.

10. Buinevich M. V., Sherbakov O. V., Izrailov K. E. Structural Model of the Machine Code Specialized for Vulnerabilities Search in Software of the Automated Control Systems // Problemy upravleniya v tekhnosfere. 2014. no. 3 (31). pp. 68–74.

Израилов Константин Евгеньевич – аспирант, СПбГУТ, Санкт-Петербург, 193232, Российская Федерация, konstantin.izrailov@mail.ru

Izrailov Konstantin – postgraduate, SPbSUT, St. Petersburg, 193232, Russian Federation, konstantin.izrailov@mail.ru