ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ И ТЕЛЕКОММУНИКАЦИИ
СПбГУТ))) Март 2016 Том 4 № 1 ISSN 2307-1303 http://www.itt.sut.ru
TELECOM IT
March 2016 Vol. 4 Iss. 1 ISSN 2307-1303 http://www.itt.sut.ru

УТИЛИТА ДЛЯ ПОИСКА УЯЗВИМОСТЕЙ В ПРОГРАММНОМ ОБЕСПЕЧЕНИИ ТЕЛЕКОММУНИКАЦИОННЫХ УСТРОЙСТВ МЕТОДОМ АЛГОРИТМИЗАЦИИ МАШИННОГО КОДА. ЧАСТЬ 1. ФУНКЦИОНАЛЬНАЯ АРХИТЕКТУРА

М. В. Буйневич[1], К. Е. Израилов[1]

[1] СПбГУТ, Санкт-Петербург, 193232, Российская Федерация
Адрес для переписки: bmv1958@yandex.ru

Информация о статье
УДК 004.4
Язык статьи – русский
Поступила в редакцию 01.02.16, принята к печати 29.02.16

Ссылка для цитирования: 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.

Аннотация
Предмет исследования. Статья посвящена авторскому методу алгоритмизации машинного кода телекоммуникационных устройств, позволяющему искать уязвимости в условиях отсутствия исходного кода. Исследуется основной этап метода на предмет его автоматизации с помощью специального программного средства в виде утилиты. **Метод.** Основные предпосылки к реализации утилиты были получены анализом близких к этапу подходов компиляции и декомпиляции, а так же используемой в методе модели. **Основные результаты.** Предложена функциональная архитектура утилиты, включающая ее деление на стадии выполнения и функциональные модули. Определены типы представлений внутренних данных утилиты, используемых для работы и взаимодействия модулей. Произведено гипотетическое моделирование примеров работы стадий и отдельных модулей на тестовых данных. **Практическая значимость.** Предложенная архитектура может быть применена для реализации целой группы средств, решающих близкие к алгоритмизации задачи.

Ключевые слова
телекоммуникационные устройства, программное обеспечение, поиск уязвимостей, машинный код, метод алгоритмизации, функциональная архитектура.

UTILITY FOR VULNERABILITY SEARCH IN A SOFTWARE OF TELECOMMUNICATION DEVICES BY METHOD ALGORITMIZATION OF MACHINE CODE. PART 1. FUNCTIONAL ARCHITECTURE

M. Buinevich[1], K. Izrailov[1]

[1] SPbSUT, St. Petersburg, 193232, Russian Federation
Corresponding author: bmv1958@yandex.ru

Article info
Article in Russian
Received 01.02.16, accepted 29.02.16

For citation: Buinevich M., Izrailov K.: Utility for Vulnerability Search in a Software of Telecommunication Devices by Method Algoritmization of Machine Code. Part 1. Functional Architecture // Telecom IT. 2016. Vol. 4. Iss. 1. pp. 115–130 (in Russian).

Abstract
Research subject. The article is devoted to the author's method algorithmization machine code telecommunications device that allows you to search for vulnerabilities when the source code is absent. We study the main stage of the method for its automation with the help of special software in the form of a utility. **Method.** The main prerequisites for the realization of utility were obtained with analysis of compilation and decompilation approaches, that is close to the phase, as well as the model, that is used in the method. **Core results.** A functional architecture utility, including its division at the stage of implementation and functional modules has been proposed. The types of internal data representations inside the utility, that are used to work and interaction modules has been defined. The hypothetical modeling examples of the work stages and the individual modules on the test data was conducted. **Practical relevance.** The proposed architecture can be used to implement an entire group of applications close to solving the algorithmization task.
Keywords
Telecommunication Devices, Software, Vulnerability Search, Computer Code, Algorithmization Method, Functional Architecture.

Введение

Функционал современных телекоммуникационных устройств полностью реализован с помощью программного кода, уязвимости в котором могут нарушать конфиденциальность, целостность и/или доступность обрабатываемой информации [1]. Их нейтрализация усложняется проприетарным характером поставляемого программного обеспечения, что приводит к отсутствию у оператора связи исходного программного кода этих устройств [2]. Возможным подходом здесь является поиск уязвимостей напрямую в их машинном коде, для чего существует некоторый набор методов и программных средств. Однако последние, по ряду причин (скорость и трудоемкость работы, субъективность выявления уязвимостей, высокая требуемая квалификация специалистов), не обеспечива-

ют удовлетворительного решения данной задачи, для чего предлагается авторский метод алгоритмизации (далее – Метод).

Суть Метода заключается в восстановлении алгоритмов работы кода по его машинному представлению, полученному путем компиляции исходного кода. Такое алгоритмизированное представление (далее – Представление) в специальной нотации [3, 4] может быть использовано для ручного поиска уязвимостей кода. Принцип работы Метода хотя и схож с *декомпиляцией*, однако имеет существенные отличия; в частности, восстановление исходного кода не только не является основным предназначением, но даже ухудшило бы эффективность его применения. Также, Метод предназначен для поиска среднеуровневых и высокоуровневых уязвимостей, ответственных за ошибки в архитектуре и алгоритмах кода, не учитывая при этом низкоуровневые – связанные с ошибками в отдельных операциях кода [5, 6].

Метод осуществляет преобразование конкретного экземпляра машинного кода к некой S-модели [7, 8], в которой отражены частично восстановленные данные, содержащие информацию о структурных особенностях исходного кода – архитектуре, модулях, подпрограммах и их алгоритмах. В связи с этим метаданные и модель названы *структурными* (СМД, структурные метаданные).

Из [9, 10] следует, что основополагающим (ключевым) этапом Метода, влияющим на успешность получения необходимых результатов, является «Этап 2. Алгоритмизация ассемблерного представления», реализуемый специальным программным средством алгоритмизации (далее – Утилитой).

Проектирование архитектуры Утилиты, предшествующее ее реализации, позволит создать ее представление с различных позиций: функциональной – раскрывающей основные модули и их взаимодействие на уровне взаимных вызовов и передачи данных; информационной – задающей Представление внутренних данных Утилиты и их преобразования; и программной – описывающей структуру программного кода, реализующего это специальное средство.

Далее, в качестве первой, будет рассмотрена функциональная архитектура, определяющая весь процесс работы Утилиты без детализации используемых внутренних Представлений и конкретной программной реализации.

Архитектурные элементы и схема функциональной архитектуры

Согласно [11, 12, 13] работа Утилиты состоит из следующих фаз (последовательно выполняющих основной функционал средства), имеющих в области разработки компиляторов следующие англоязычные названия: Front-End, Middle-End и Back-End. В первой фазе осуществляется разбор ассемблерного кода (АК), поэтому для функционирования Front-End потребуется реализация собственного лексического анализатора, разработка формальной грамматики расширенного ассемблера, реализация правил грамматики для построения дерева абстрактного синтаксиса и алгоритмов преобразования дерева в платформенно-независимое внутреннее Представление. Во второй фазе производится обработка построенных деревьев, выделение СМД, сбор информации об уязвимостях и построение S-модели (с помощью набора графов, таблиц, хэшей, дополнительных деревьев и т. п.). Также возможно восстановление потерянных СМД на основании распространенных практик программирования; основная масса заданных корректировок моделирования учитывается именно здесь. Для функ-

ционирования Middle-End потребуется реализация собственных алгоритмов обходов графов и соответствия их элементов шаблонам уязвимостей. В третьей фазе создается алгоритмизированное Представление АК и осуществляется его генерация в текстовом виде согласно выходному синтаксису. При наличии информации, собранной в Middle-End, построение такого Представления и генерация являются технически тривиальными.

Поскольку очевидна близость цели и функционала Утилиты к подходам, используемым при компиляции и декомпиляции, то и ее архитектуру целесообразно сделать аналогичной. При этом основополагающим элементом в архитектуре будет S-модель, вокруг которой будет строиться остальной функционал. Утилита может иметь вид консольного приложения, поскольку все взаимодействия в ней пользователя сведены до предоставления входных данных и анализа выходных. Для отладки Утилиты достаточно использование отладочного вывода ее внутренних представлений. Принцип работу Утилиты может быть сведен к последовательному (пофазному) выполнению действий согласно этапам Метода (в отличии, например, от приложений с графическим интерфейсом, имеющим принцип работы, связанный с обменом сообщениями или переходам между внутренними состояниями). В данной Утилите отсутствует прямая необходимость обмена данными с внешними объектами (другими программными средствами и базами данных). Внутреннее Представление S-модели в такой архитектуре определяется, как совокупность различных структур, описывающих Представления входного АК, на котором определены алгоритмы модулей Утилиты. Предлагаемая функциональная архитектуры Утилиты в схематичном виде показана на рис. 1.

На рис. 1 присутствуют следующие архитектурные элементы: во-первых, это вышерассмотренные фазы Утилиты; во-вторых, это модули архитектуры и их взаимодействия; в-третьих, это входные и выходные данные Утилиты; и, в-четвертых, это внутренние данные Утилиты, определяющие S-модель. Как хорошо видно, архитектура являет собой пофазное многостадийное выполнение преобразования АК в его алгоритмизированное Представление. Каждая стадия состоит из модулей, реализующих ее функционал. Модули производят преобразования внутренних данных, используемых также и в качестве способа обмена между ними. Совокупность таких данные образует внутреннее Представление S-модели, которая строится на первой стадии по машинному коду и обрабатывается на всех последующих; притом последняя стадия производит генерацию алгоритмизированного Представления в виде алгоритмов входного АК с информацией о потенциальных уязвимостях.

Модульная структура

Все модули архитектуры выполнены как независимые единицы выполнения. Обмен между модулями на различных стадиях работы Утилиты осуществляется посредством внутренних Представлений; таким образом, каждая совокупность модулей одной стадии выполняет четко заданный набор действий по созданию нового или преобразованию старого Представления кода.

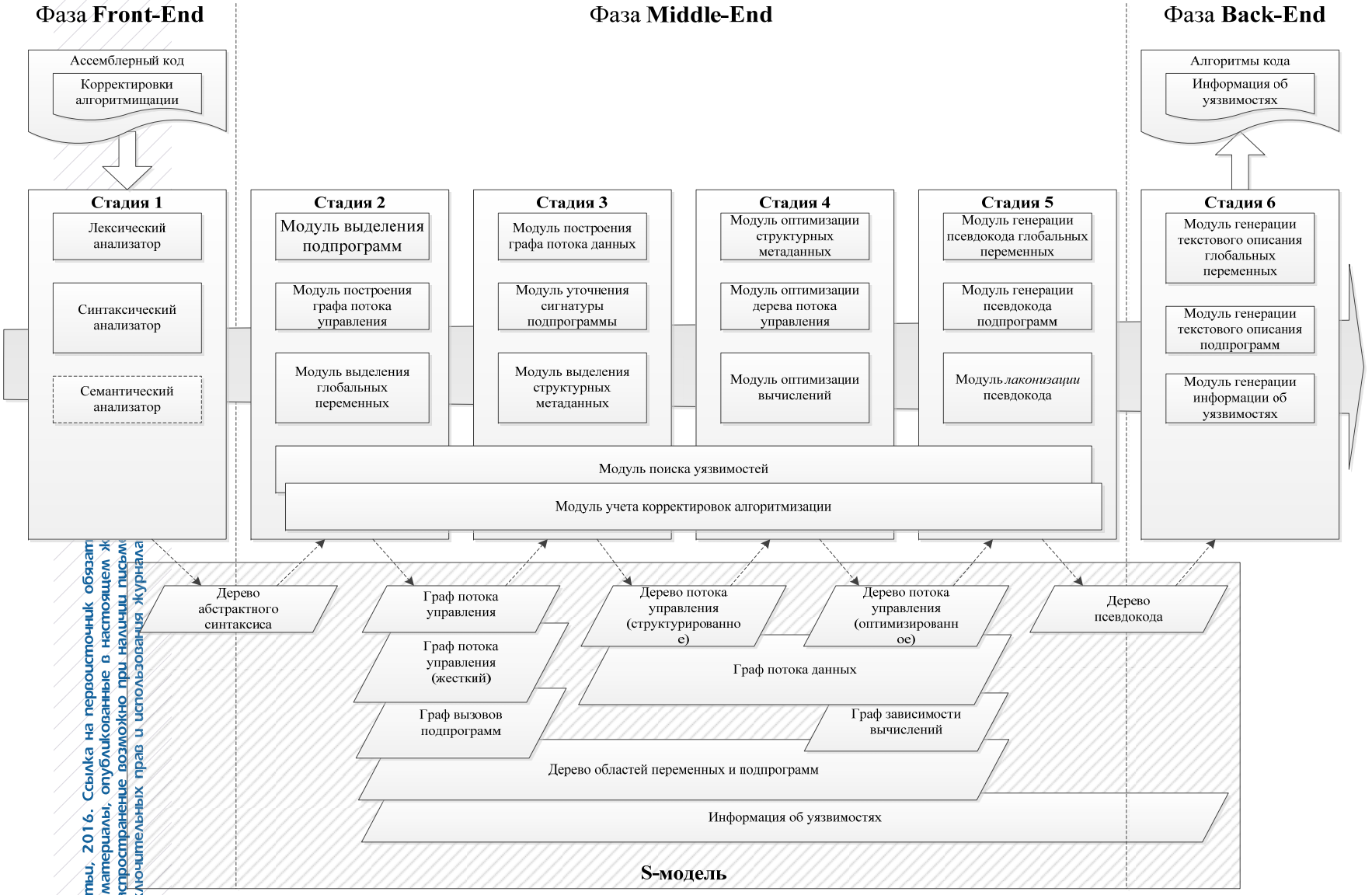


Рис. 1. Функциональная архитектура Утилиты (схема)

Взаимодействие модулей по стадиям выглядит следующим образом: модули Стадии 1 принимают на вход АК и создают дерево абстрактное синтаксиса; модули Стадии 2 принимают на вход дерево абстрактного синтаксиса и создают графы потока управления и вызовов подпрограмм, заполняя дерево областей переменных и подпрограмм; модули Стадии 3 принимают на вход граф потока управления и, обновляя дерево областей переменных и подпрограмм, создают дерево потока управления; модули Стадии 4 оптимизируют это дерево потока управления, используя дерево областей переменных и подпрограмм; модули Стадии 5 строят по нему дереву псевдо кода, которое используется модулями Стадии 6 для генерации текстового описания алгоритмов АК.

Также, на Стадии 4 строится граф зависимости вычислений (на основании используемых в них общих переменных), используемый на этой же стадии для оптимизации.

Прямое взаимодействие модулей между стадиями практически отсутствует, поскольку каждый из модулей реализует собственный ограниченный функционал над входным в стадию или выходным из нее Представлением (графом или деревом). Такая организация модулей позволяет, в частности, без существенных трудозатрат добавлять новый функционал для оптимизации и лаконизации деревьев. Последняя, в отличие от оптимизации, призвана улучшить субъективные особенности кода, такие, как: компактность и воспринимаемость человеком за счет использования в конечном Представлении алгоритмов более компактных выражений, специальных языковых конструкций, именованных адресов и т. п. Исключением являются два модуля – учета корректировок алгоритмизации и поиска уязвимостей, поскольку они предоставляют данные и собственный функционал всем остальным модулям, не зависимо от стадии выполнения последних.

Опишем стадии работы Утилиты и реализующие их модули более подробно.

Стадия 1

Стадия предназначена для разбора входного АК и построения его внутреннего Представления в фазе Front-End. Стадия представляет собой классический пример компилятора и состоит из 3-х модулей: лексического, синтаксического и семантического анализаторов. Первый предназначен для разбиения входного потока символов АК на отдельные *лексемы* (подобно сборке букв в слова), осуществляя тем самым базовую формализацию. Второй собирает отдельные лексемы, сопоставляя их с заранее заданными правилами (подобно словесным предложениям). Для такого разбора используются формальные грамматики, заданные синтаксисом входного языка с помощью рекурсивных правил; последние дополняются пользовательскими действиями (на используемом языке программирования), выполняемыми при *сборке* правил. В результате строится дерево абстрактного синтаксиса, отражающее входной код в полностью формализованном и структурированном виде. Для разбора смысловых значений правил предназначен третий модуль стадии – семантический анализатор. И хотя в классических компиляторах ему отводится значимое место, в текущей архитектуре модуль может считаться лишь условным, поскольку он выполняет лишь вспомогательные действия, такие, как добавление в дерево абстрактного син-

таксиса узлов, хранящих результаты операций сравнения для соответствующих инструкций процессора. В результате работы модулей будет создано дерево абстрактного синтаксиса входного АК, подходящее для обработки на дальнейших стадиях.

Необходимо отметить, что, хотя стадия и является зависимой от процессора исполнения (поскольку в ней использован синтаксис входного АК), однако реализацию синтаксического анализатора возможно сделать таким образом, чтобы генерируемое им абстрактное дерево не использовало инструкций процессора, а оперировало абстрактными операциями и переменными. Например, следующий АК для процессора PowerPC:

```
LI R0, 0x1          ; R0 = 0x1
ADDI R1, R0, 0x2    ; R1 = R0 + 0x2
```

можно уже в процессе синтаксического анализа преобразовать в следующее абстрактное дерево (рис. 2), аналогичное процессорно-независимому коду:

```
X = 0x1;
Y = X + 0x2.
```

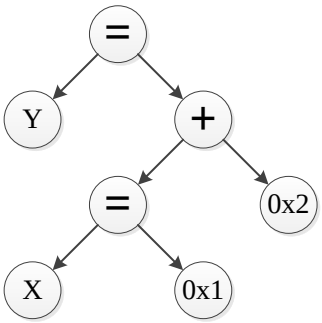


Рис. 2. Пример абстрактного дерева простейшего выражения

Такое обеспечение процессорной инвариантности последующего уровня от Front-End позволяет разрабатывать обобщенные (шаблонные) алгоритмы, что повышает, как переносимость кода, так и его надежность.

Стадия 2

Стадия является первой в фазе Middle-End и предназначена для начальной обработки дерева абстрактного синтаксиса и разделению его на различные слои, содержащие информацию о подпрограммах, потоке управления, глобальных и локальных переменных. Такое преобразование является побочным результатом работы следующих модулей стадии. Во-первых, Модуль выделение подпрограмм анализирует дерево абстрактного синтаксиса, идентифицируя подпрограммы и занося информацию о них в дерево областей переменных и подпрограмм. Во-вторых, используя результаты идентификации, Модуль построения потока управления строит соответствующий граф, определяющий все переходы внутри тела подпрограммы. Граф потока управления целесообразно преобразовать к, так называемому, «жесткому» (путем добавления служебных узлов, строго задающих структуру графа), имеющему более формальную струк-

туру, чем первый – это необходимо для упрощения реализации алгоритмов его обработки. В нем, например, в начале веток условного ветвления добавлены служебные узлы, чтобы ветка всегда имела, по крайне мере, один узел – в «нежестком» графе пустая ветка не будет иметь узлов, что усложнит алгоритмы ее обработки необходимыми проверками наличия узлов и т. п. В процессе построения потока управления анализируются вызовы внешних подпрограмм, что позволяет параллельно строить и граф их вызовов. И в-третьих, анализ дерева абстрактного синтаксиса Модулем выделения глобальных переменных идентифицирует и занесет информацию о последних также в дерево областей переменных и подпрограмм. Данное дерево является упрощенным аналогом дерева областей видимости, используемого в компиляторах. Таким образом, дерево областей переменных и подпрограмм хранит информацию, как обо всех переменных, так и о подпрограммах, включая косвенно графы потока управления последних. Обработка дерева абстрактного синтаксиса, аналогичного следующему примеру С-подобного псевдокода:

```
var GLOBAL;
FUNCT(R1) {
  R0 = R1 + GLOBAL ;
  return R0;
}
```

построит графа потока управления и занесет данных в дерево областей переменных и подпрограмм, как показано на рис. 3.

Рис. 3. Пример взаимосвязи дерева абстрактного синтаксиса и дерева областей переменных и подпрограмм

Стадия 3

Стадия предназначена для выделения СМД во внутренних Представлениях АК, полученных на предыдущих стадиях. Для этого, в частности, Модуль построения графа потока данных создает привязанную к графу потока управления информацию о времени жизни переменных, их значениях, первых/последних точках использования и т. п. Данная информация может быть использована как на данной стадии, так и на последующих. Модуль уточнения сигнатуры подпрограмм анализирует граф потока управления и, используя граф потока данных, предсказывает такие свойства сигнатуры подпрограмм, как входные и выходные параметры. В рамках АК под параметрами подпрограммы подразумеваются наборы регистров, с помощью которых подпрограмма получает внешние значения и выдает результаты своих вычислений. Например, для следующего примера ассемблерной подпрограммы на С-подобном псевдо-коде:

```
??? FUNCT(???) {
  R0 = R1 + R2;
  return R0;
}
```

очевидно, что, скорее всего подпрограмма FUNCT() принимает на входе параметры посредством двух регистров – *R*1 и *R*2, поскольку они используются без какой либо явной инициализации, и возвращает на выходе результат вычислений через регистр *R*0 поскольку ему присваивается значение без какого-либо последующего использования; вариант же генерации неоптимального кода современным компилятором невозможен. Так, используя приведенную выше логику, Модуль уточнения сигнатуры для вышеприведенного примера определит сигнатуру подпрограммы следующим образом:

```
( R0 ) FUNCT( R1, R2 );
```

Необходимо отметить, что, хотя для завершенных сигнатур подпрограммы (с точки зрения, например, языка С) требуются также типы аргументов и возвращаемого значения, тем не менее, для описания логики работы алгоритмов они не существенны и данным модулем не восстанавливаются.

Основным модулем стадии (и, в некотором смысле, всей Утилиты), является Модуль выделения СМД, параллельно переводящий графовидное Представление потока управления подпрограммы в древовидное, подобное диаграммам Насси-Шнейдермана. Основные функции модуля состоят из выделения всех условных переходов (включая управляющие конструкции, ветки и завершающие метки) и циклов на графе управления (включая условия выхода из цикла и переходы по итерациям). Анализ модуля осуществляется с применением рекурсий и раскраски графов. Перестроение графа с использованием конструкций Насси-Шнейдермана производится в процессе этого анализа. Примеры такого перестроения для простейшего условного перехода и цикла показаны на рис. 4.

Ситуация, когда граф потока управления не может быть сведен к древовидной структуре (например, при наличии оператора безусловного перехода

GOTO) считается вырожденной и описывается в дереве потока управления введением дополнительной связи (рис. 5).

Как хорошо видно на примерах, переход к древовидному Представлению потока управления увеличивает структурируемость формы кода, что должно положительно сказаться на его восприятии человеком.

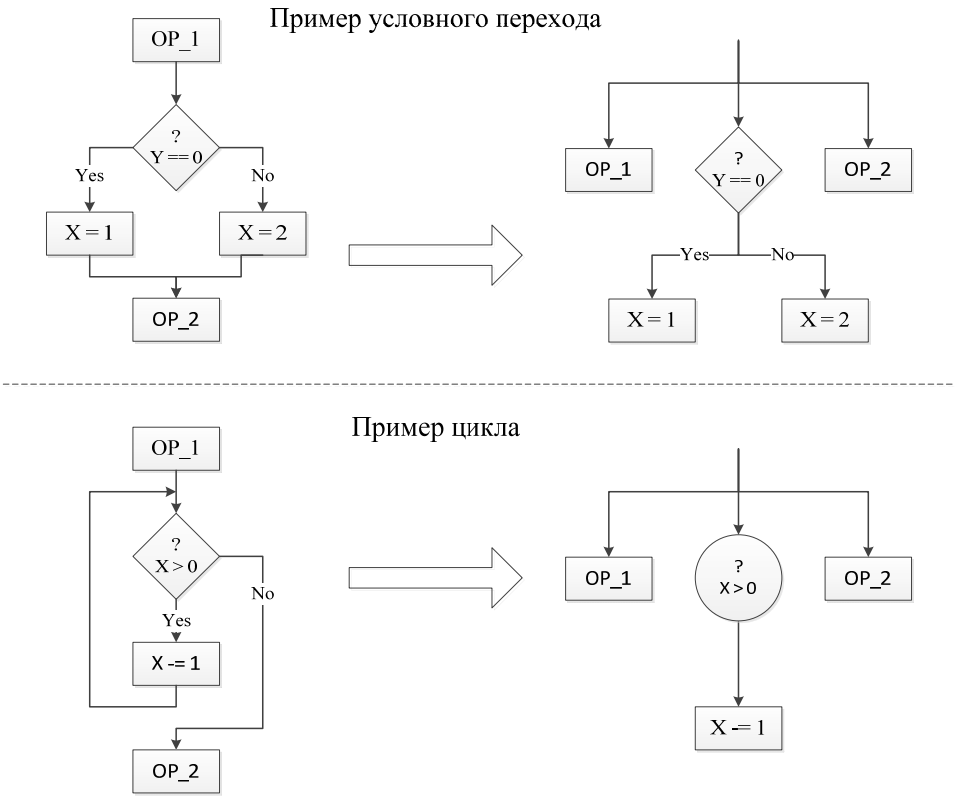


Рис. 4. Примеры построения диаграмм Насси-Шнейдермана для условного перехода и цикла

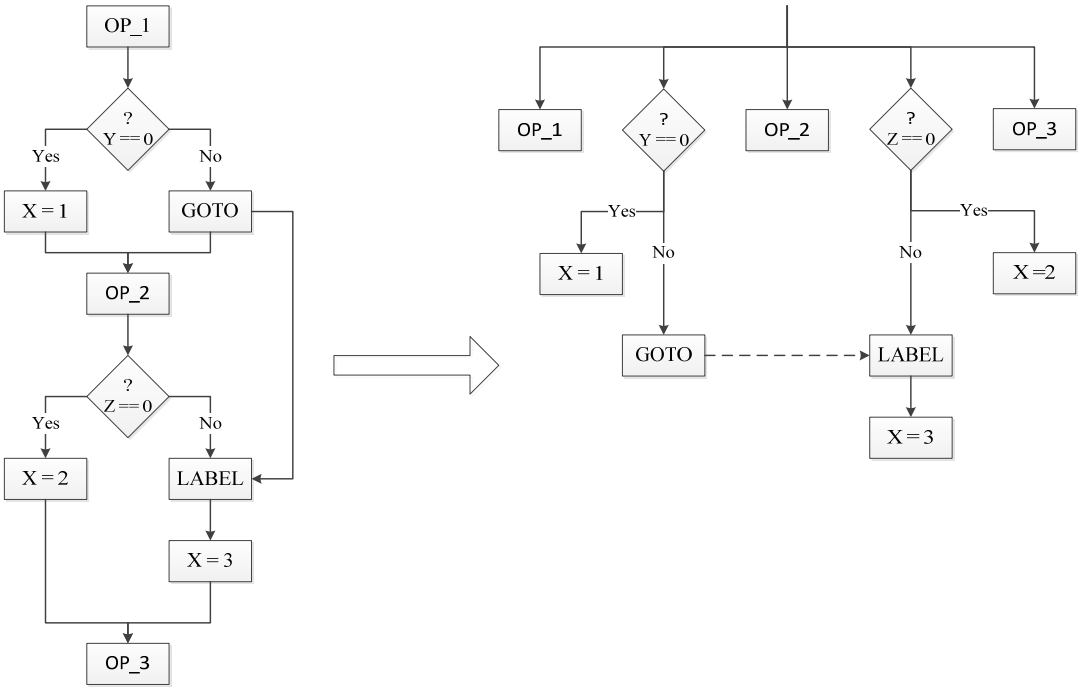


Рис. 5. Пример с безусловным переходом, не сводимый к древовидной структуре

Стадия 4

Стадия предназначена для оптимизации представления кода в интересах сокращения его размера и упрощения управляющих структур; она состоит из 3-х модулей:

1) Модуля оптимизации СМД, выполняющего следующие действия:

– удаление неиспользуемых меток, а также безусловного перехода на следующую операцию после ветвления, на начало цикла и на первую инструкцию после цикла;

– объединение меток, а также нескольких веток условных переходов и выхода из цикла;

– пересортировка веток условных переходов;

– вынесение выхода из подпрограммы вне цикла;

2) Модуля оптимизации дерева потока управления, выполняющего замену безусловного выхода из цикла на конструкцию BREAK и безусловного начала следующей итерации цикла на конструкцию NEXT.

3) Модуля оптимизации вычислений, выполняющего следующие действия:

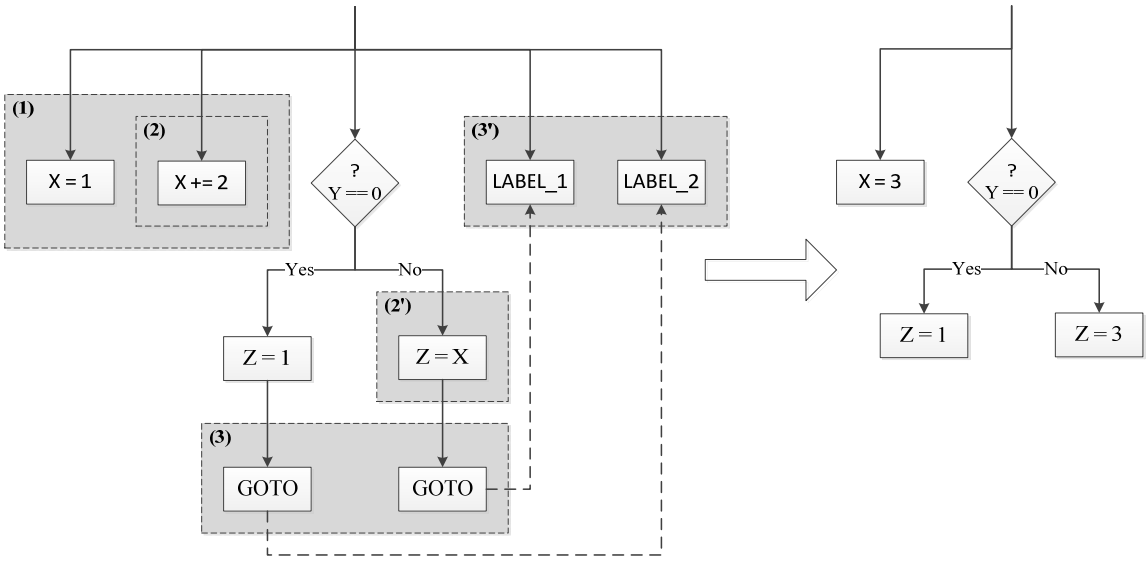
– вычисление значений выражений, включая промежуточные;

– замена вычисленных значений выражений на соответствующие константы, а также битового доступа к переменным на специальные конструкции, вида VAR.N_BIT;

– упрощение выражений путем подстановки инициализационных значений переменных;

– удаление операций, не имеющих эффекта, а также операций, эффект которых не изменяет состояние программы.

Пример оптимизации дерева потока управления показан на рис. 6.



Примененная оптимизация:
(1) – Вычисление значений выражений
(2)-(2') – Подстановка значений выражений
(3)-(3') – Удаление переходов на операцию после ветвления

Рис. 6. Пример оптимизации дерева потока управления

В процессе работы стадии также строится граф зависимости вычислений, который отражает выражения в коде и переменные, от которых зависят вычисления. Граф используется только в рамках стадии Модулями оптимизации и обновляется после большинства крупных операций по перестроению дерева потока управления.

Все модули выполняются в несколько проходов, пока ни одна из их оптимизаций не перестанет иметь эффект – т. е. производить какие-либо изменения во внутренних представлениях Утилиты. Зацикливание стадии предотвращает то, что все оптимизации модулей реализуются таким образом, чтобы всегда упрощать или не изменять Представление кода.

Стадия 5

Стадия предназначена для генерации псевдокода алгоритмов в Представлении соответствующего дерева (включая как подпрограммы, так и глобальные переменные) с последующим приведением их в лаконичный вид, и реализуется следующими модулями.

Модуль генерации псевдокода глобальных переменных производит их добавление в дерево псевдокода с указанием адресов размещения (при возможности). Модуль генерации псевдокода подпрограмм практически без изменений переносит Представление каждой подпрограммы в дерево псевдокода. Затем на данном дереве Модуль лаконизации псевдокода производит его видоизменение в интересах повышения восприятия кода человеком, что обеспечивается следующими действиями:

– сортировка и комбинирование коммутативных операций;

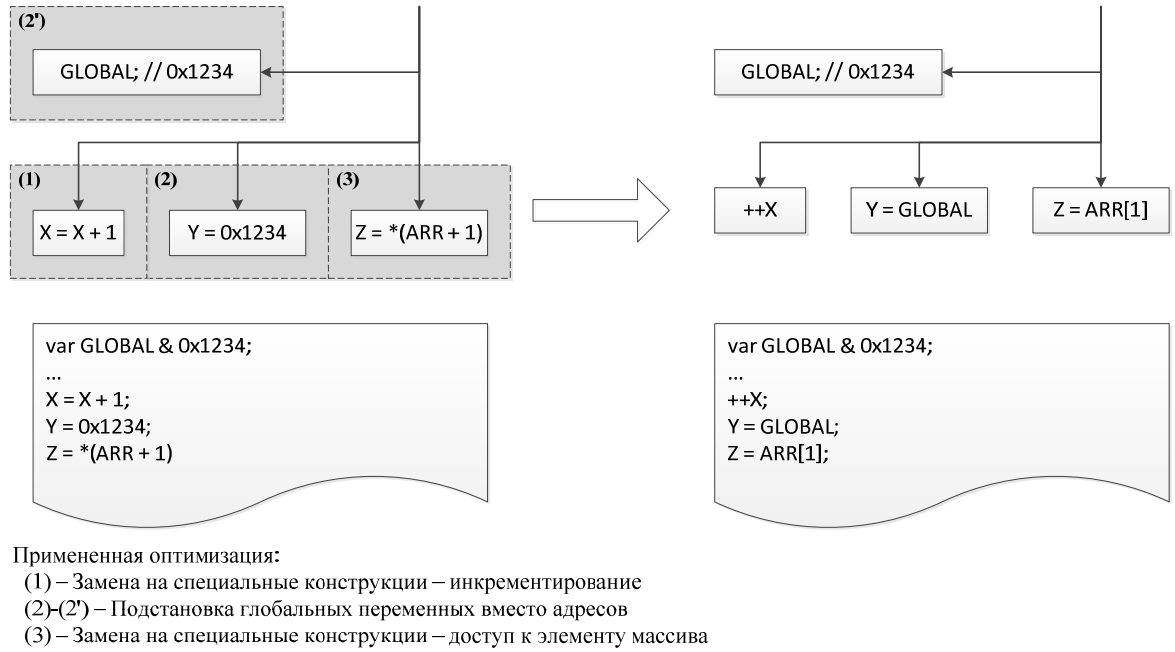
– упрощение булевских операций, операций сравнения, а также операций сложения/вычитания отрицательных чисел;

– замена абсолютных адресов на соответствующие глобальные переменные, а также вычислительных операций на специальные конструкции – инкрементирование/декрементирование (++, --), операция с присваиванием ([+-*/]=), доступ к элементу массива.

Пример лаконизации дерева псевдокода показан на рис. 7.

Таким образом, стадия практически не сокращает объем кода, а только делает его более воспринимаемым за счет использования специальных конструкций и выбора других форм представления элементов кода. Стадия является заключительной в фазе Middle-End.

Все модули стадии выполняются в несколько проходов, пока ни одна из их лаконизаций не перестанет производить изменения. Стадия не может зацикливаться по причинам, аналогичным Стадии 4.



Примененная оптимизация:
(1) – Замена на специальные конструкции – инкрементирование
(2)-(2') – Подстановка глобальных переменных вместо адресов
(3) – Замена на специальные конструкции – доступ к элементу массива

Рис. 7. Пример лаконизации дерева псевдокода

Стадия 6

Стадия предназначена для генерации выходного описания алгоритмов кода на основании построенных, оптимизированных и лаконизированных внутренних Представлений в фазе Back-End. Стадия представляет собой классический пример компилятора и состоит из трех модулей.

Модуль генерации текстового описания глобальных переменных создает описание всех глобальных переменных, восстановленных по АК, в текстовом виде, при возможности указывая их адреса. Пример такого вывода для глобальных переменных global_1 и global_2, последняя из которых размещенная по адресу 0x1234, будет следующим:

```
var global_1;
var global_2 & 0x1234;
```

Модуль генерации текстового описания подпрограмм создает описания всех подпрограмм (включая их сигнатуры и алгоритмы) в текстовом виде. Пример такого вывода для подпрограммы FUNCT(), инкрементирующей и возвращающей значение параметра (через регистр r0), будет следующим:

```
(r0) FUNCT(r0) {
        ++ r0;
        return r0;
}
```

Модуль генерации информации об уязвимостях добавляет пометки о возможных уязвимых местах (используя данные, собранные Модулем поиска уязвимостей) в генерируемый код. Пример такого вывода для функции

Destructed_Funct(), структура алгоритма которой была разрушена, будет следующим:

```
Destructed_Funct() {
   /* ATTENTION!!! Possible, destruction of the structure. */;
}
```

Модули фазы Middle-End

На всех стадиях фазы Middle-End функционируют два модуля – учета корректировок алгоритмизации и поиска уязвимостей. Первый, используя информацию о корректировках алгоритмизации, полученную при разборе входного ассемблера, управляет работой всех модулей в этой фазе, например, явно задавая для указанной ассемблерной подпрограммы регистры, используемые в качестве ее аргументов. Таким образом, модуль влияет как на топологию и отдельные характеристики внутренних Представлений, так и на конечный текстовый вид алгоритмов. Второй же предоставляет различные шаблоны и алгоритмы для выделения информации об потенциальных уязвимостях, которые, в конечном итоге, используются Модулем генерации информации об уязвимостях для их вывода. С точки зрения линейности выполнения стадий, данные модули можно назвать *ортогональными*, поскольку они, так или иначе, могут взаимодействовать со всеми другими.

Продолжение следует

Литература

1. Буйневич М. В., Владыко А. Г., Доценко С. М., Симонина О. А. Организационно-техническое обеспечение устойчивости функционирования и безопасности сети связи общего пользования. СПб. : СПбГУТ, 2013. 144 с.
2. Израилов К. Е. Анализ состояния в области безопасности программного обеспечения // В сборнике: Актуальные проблемы инфотелекоммуникаций в науке и образовании. II Международная научно-техническая и научно-методическая конференция. 2013. С. 874–877.
3. Израилов К. Е. Алгоритмизация машинного кода телекоммуникационных устройств как стратегическое средство обеспечения информационной безопасности // Национальная безопасность и стратегическое планирование. 2013. № 2 (2). С. 28–36.
4. Израилов К. Е. Расширение языка «С» для описания алгоритмов кода телекоммуникационных устройств // Информационные технологии и телекоммуникации. 2013. № 2 (2). С. 21-31.
5. Буйневич М. В., Щербаков О. В., Владыко А. Г., Израилов К. Е. Архитектурные уязвимости моделей телекоммуникационных сетей // Научно-аналитический журнал «Вестник Санкт-Петербургского университета Государственной противопожарной службы МЧС России». 2015. № 4. С. 86–93.
6. 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. С. 430–435.
7. 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.
8. 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.
9. Буйневич М. В., Израилов К. Е. Метод алгоритмизации машинного кода телекоммуникационных устройств // Телекоммуникации. 2012. № 12. С. 2–6.

10. 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. С. 76–80.

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

12. 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. С. 172–176.

13. 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. С. 700–707.

References

1. Bujnevich M. V., Vladyko A. G., Docenko S. M., Simonina O. A. Organizacionno-tekhnicheskoe obespechenie ustojchivosti funkcionirovaniya i bezopasnosti seti svyazi obshchego pol'zovaniya (Organizational and Technical Support of Stability of Functioning and Security of Public Communications Networks). SPb. : SPbGUT, 2013. 144 p.

2. Izrailov K. E. Analysis of the State in the Field of Security Software // V sbornike: Aktual'nye problemy infotelekommunikacij v nauke i obrazovanii. II Mezhdunarodnaya nauchno-tekhnicheskaya i nauchno-metodicheskaya konferenciya. 2013. pp. 874–877.

3. Izrailov K. Algorithmization Machine Code of Telecommunication Devices As a Strategic Means of Ensuring the Information Security // Nacional'naya bezopasnost' i strategicheskoe planirovanie. 2013. no. 2 (2). pp. 28–36.

4. Izrailov K. E. The «C» Language Extension for Describe Code Algorithms of Telecommunication Devices // Telecom IT. 2013. no. 2 (2). pp. 21-31.

5. Bujnevich M. V., Shcherbakov O. V., Vladyko A. G., Izrailov K. E. Architectural Vulnerability of Telecommunications Networks Models // Nauchno-analiticheskij zhurnal «Vestnik Sankt-Peterburgskogo universiteta Gosudarstvennoj protivopozharnoj sluzhby MCHS Rossii». 2015. no. 4. pp. 86–93.

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

7. Bujnevich M. V., Shcherbakov O. V., Izrailov K. E. Model Machine Code Specialized for Vulnerabilities Search // Vestnik Voronezhskogo instituta GPS MCHS Rossii. 2014. no. 2 (11). pp. 46–51.

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

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

10. 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.

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

12. 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.

13. 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.

Буйневич Михаил Викторович – доктор технических наук, профессор, ведущий научный сотрудник, СПбГУТ, Санкт-Петербург, 193232, Российская Федерация, bmv1958@yandex.ru

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

Buinevich Michael – D.Sc., professor, leading scientific researcher, SPbSUT, St. Petersburg, 193232, Russian Federation, bmv1958@yandex.ru

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