Научная статья
УДК 004.42
DOI:10.31854/1813-324X-2023-9-1-75-93

Моделирование программы с уязвимостями с позиции эволюции ее представлений. Часть 1. Схема жизненного цикла

Константин Евгеньевич Израилов, konstantin.izrailov@mail.ru

Санкт-Петербургский Федеральный исследовательский центр Российской академии наук,
Санкт-Петербург, 199178, Российская Федерация

Аннотация: *Изложены результаты исследования процесса создания программ и возникающих при этом уязвимостей. В первой части цикла статей предлагается графическая схема жизненного цикла представлений (а именно следующих: Идея, Концептуальная модель, Архитектура, Двухмерная структурная схема, Функциональная диаграмма, Блок-схема, Структурограмма, Псевдокод, Классический, Метакод генерации, Сценарный код, Ассемблерный код, Дерево абстрактного синтаксиса, Машинный код, Байт-код), через которые проходит любая типовая программа. Указываются основные свойства таких представлений – назначение, форма и содержание, способы получения и восстановления представлений, а также возможные уязвимости и способы их обнаружения. Вводится вложенная классификация уязвимостей, состоящая из их деления по структурному уровню в программе, изменению содержания функционала и воздействию на обрабатываемую информацию.*

Ключевые слова: *программная инженерия, информационная безопасность, уязвимости, представления программы, жизненный цикл, моделирование*

Благодарность: Автор выражает благодарность своему научному консультанту – доктору технических наук, профессору М.В. Буйневичу за существенный вклад в текущее исследование.

Ссылка для цитирования: Израилов К.Е. Моделирование программы с уязвимостями с позиции эволюции ее представлений. Часть 1. Схема жизненного цикла // Труды учебных заведений связи. 2023. Т. 9. № 1. С. 75–93. DOI:10.31854/1813-324X-2023-9-1-75-93

Modeling a Program with Vulnerabilities in the Terms of Its Representations Evolution. Part 1. Life Cycle Scheme

Konstantin Izrailov, konstantin.izrailov@mail.ru

Saint-Petersburg Federal Research Center of the Russian Academy of Sciences,
St. Petersburg, 199178, Russian Federation

Abstract: *The investigation results of the creating programs process and the resulting vulnerabilities are presented. The first part of the articles series offers a life cycle graphical scheme of the representations (namely, the following: Idea, Conceptual model, Architecture, 2D block diagram, Function diagram, Flowchart, Structogram, Pseudo-code, Classical code, Generation metacode, Script code, Assembly code, Abstract Syntax Tree, Machine Code, Bytecode) through which any sample program passes. The main properties of such representations are indicated - the purpose, form and content, obtaining and restoring representations methods, as well as possible vulnerabilities and ways to*

detect them. A vulnerabilities nested classification is introduced, consisting of their division according to the structural level in the program, the change in the content of the functionality and the impact on the information being processed.

Keywords: *software engineering, information security, vulnerabilities, program representations, life cycle, modeling*

Acknowledgment: The author expresses his gratitude to his scientific adviser - Doctor of Technical Sciences, Professor M.V. Buinevich for significant contributions to the current research.

For citation: Izrailov K. Modeling a Program with Vulnerabilities in the Terms of Its Representations Evolution. Part 1. Life Cycle Scheme. *Proc. of Telecom. Universities.* 2023;9(1):75–93. (in Russ.) DOI:10.31854/1813-324X-2023-9-1-75-93

1. ВВЕДЕНИЕ

Одной из актуальнейших проблем в сфере информационной безопасности (далее – ИБ) является недостаточное количество удовлетворительных способов поиска в программном обеспечении (далее – ПО) уязвимостей, наличие которых может приводить к нарушениям конфиденциальности, целостности и доступности критически важной информации. Существующие же способы можно считать подходящими лишь для отдельных типов уязвимостей или условий применения. Например, отслеживание программных исключений путем «фаззинга» данных в памяти тестируемой программы [1]. Для разрешения данной проблемы необходимо, в первую очередь, исследовать сам породивший ее объект – уязвимость: при этом не только статические, но и динамические свойства, определяющие ее жизненный цикл (далее – ЖЦ) в процессе создания программ). Анализ динамических свойств позволит более широко охватить все возможные способы поиска, поскольку они напрямую зависят от того, какой вид имеет уязвимость, что непосредственно влияет на ее обнаруживаемость. В свою очередь, статические свойства дают лишь удобные классификации уязвимостей и частично формируют дополнительные знания о них. В такой постановке проблемного вопроса актуальным является моделирование ЖЦ программы с уязвимостями, для чего используется авторская нотация. Предметная область исследования будет задана, как рассмотрение эволюции программы в процессе ее создания и появляющихся при этом уязвимостей.

2. ОСНОВНЫЕ ПОНЯТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ В НОТАЦИИ ПРЕДСТАВЛЕНИЙ

Для проведения любого исследования требуется однозначный понятийный аппарат. В этом аспекте базовые термины настоящей предметной области используются современными исследователями в крайне неоднозначной трактовке, приводя их к трудно сопоставимым результатам и противоречивым выводам.

Так, только на международном уровне[1] декларировано 16 критериев классификации программных средств, что свидетельствует о большом разнообразии «софта» и не могло не сказаться на его определении, как IT-термина. В результате имеется множество, в том числе стандартизованных[2] в РФ определений ПО с акцентом на вторую составляющую – *обеспечение* компьютера (компьютерной системы), как средства решения задачи. Причиной последнего, скорее всего, является направленность IT-мысли лишь на субъект исполнения, а не на объект приложения ПО, а именно – на информацию.

Еще более дискуссионным, но весьма характерным для программы, является понятие (ее) уязвимости. Попытки использования устоявшихся и общепринятых[3], не всегда тождественных (но по смыслу сводящихся к некому эксплуатируемому источником угрозы дефекту или «слабости» ПО), определений в немалой степени затрудняет систематизацию и формализацию предметной области; за редким положительным исключением [2].

С учетом вышеизложенного предлагается введение следующих, более конкретных и «ортогональных» определений используемых далее понятий, что позволит повысить однозначность результатов проводимого исследования.

Будем понимать под компьютерной программой некую автоматическую процедуру (конкретную или обобщенную) решения вполне определенной задачи, связанной с обработкой информации – т. н. информационной задачи. Ввиду технологичности процесс создания программы состоит из типовых стадий; в рамках каждой стадии определенные

[1] ISO/IEC TR 12182:2015. Systems and software engineering. Framework for categorization of IT systems and software, and guide for applying it.

[2] ГОСТ 19781-90 Обеспечение систем обработки информации программное. Термины и определения. М.: Стандартинформ, 2010; ГОСТ 34.003-90 Информационная технология. Автоматизированные системы. Термины и определения. М.: Стандартинформ, 2009.

[3] ГОСТ Р 56545-2015 Защита информации. Уязвимости информационных систем. Правила описания уязвимостей. М.: Стандартинформ, 2018.

процессы (или их объединения) производят соответствующие преобразования. Тем самым программа находится в своих различных представлениях (состояниях, точках развития), каждое из которых создается на основании предыдущего; также оно являются входным для одних стадий или процессов и выходным для других.

Каждое такое представление (далее – Представление) является отдельным этапом эволюции решения задачи, не изменяя основополагающего содержания ее решения (в смысле его суть), постепенно конкретизируясь путем перехода от человеко-понятной к машинно-ориентированной форме. Т. е. содержание решения сохраняется на протяжении всего процессе создания программы, постепенно обретая конечную реальную форму из начального заданного идеала; используемая переменчивая форма для неизменного содержания соответствует применяемым стадиям-процессам. При этом, как и любая естественная эволюция, развитие решения задачи может иметь разветвления на параллельные Представления вследствие применения процессов их формирования различного рода (например, алгоритмы могут быть описаны графически или текстуально, а в качестве выполняемой сборки может быть получен машинный или байт-код).

Каждое Представление имеет назначение (или цель своего существования), определяющее основное его отличие от остальных и, как следствие, уникальную форму, наиболее подходящую для хранения содержания.

Для уменьшения неоднозначности в делении программы на Представления будем последние определять на основании их реального существования в виде законченной совокупности формы и содержания, а также учитывая распространенные практики применения. Так, например, Представление в виде блок-схем алгоритмов имеет право на существование, поскольку несмотря на то, что при реальном создании программ блок-схемы применяются редко, они используются для документирования и стандартизованы[4]. Впрочем, такая практика *неиспользования* отдельных Представлений исходит исключительно из экономических соображений, завышенных амбиций и умения разработчиков кода держать в голове и его алгоритмы, что в конечном итоге негативно сказывается на качестве создаваемого ПО. Следуя такой же логике, внутреннее представление байт-кода в виртуальной машине (имеющее, возможно, форму машинных инструкций для режима Just-In-Time – осуществляющего такое преобразование в процессе выполнения) не будет считаться отдельным Представлением, поскольку оно относится к стадии выполнения, а создание программы в этом случае заканчивается на Представлении выполняемого байт-кода.

Полный цикл процесса создания программы осуществляется ее переходом из Представления изначальной идеи в код, готовый для выполнения. Для работы последнего необходима соответствующая среда, воспринимающая только определенную форму Представления, через которую и передается содержание программы для непосредственного выполнения. Отсюда органично вытекает определение жизненного цикла программы как череды Представлений от изначального (придуманной идеи человеком) до одного из финальных (реализующего идею кода для автомата).

Наиболее известным и широко используемым Представлением можно считать исходный код, представляющий собой некие текстовые читаемые сообщения на специальном языке или нотации, предназначенные для строгого описания процедуры решения. Такой код может как выполняться напрямую (сценарный), так и преобразовываться в управляемый и выполняемый виртуальной машиной (байт-код) или платформенно-зависимый (машинный). Последний случай считается классически используемым при создании встроенного ПО, например, для телекоммуникационных, мобильных или киберустройств.

В рамках введенных понятий сразу отметим, что ни здесь, ни далее не рассматривается особый вид программ, предназначенный не столько для решения задач, сколько для их спецификации и ожидаемого результата – соответствующий декларативной парадигме программирования (примерами исходного кода таких программ служат языки запросов SQL и разметки HTML). Такое исключение как раз и призвано избавиться от смешения используемых понятий и парадигм.

Введем авторское – основанное на уже частично сформированной понятийной базе – определение уязвимости. Уязвимостью будем называть отличия между начальным (идеальным, исходно задуманным) и текущим (промежуточным или реально выполняемым) содержанием Представлений; отличия в их форме, очевидно, являются вполне законными, поскольку они как раз и служат способом «транспортировки» содержания от ориентированного на человека до понятного машине (т. е. выполняющему программу автомату). Поскольку все же уязвимости в большинстве случаев связаны именно с вопросами ИБ, то уточним указанные отличия теми, которые приводят к изменениям в состояниях конфиденциальности, целостности и доступности информации при ее обработке программой в этих Представлениях. Так, если начальное

[4] ГОСТ 19.701-90 Единая система программной документации. Схемы алгоритмов, программ, данных и систем. Обозначения условные и правила выполнения. М.: Стандартинформ, 2010.

Представление имеет функционал, обеспечивающий одно состояние ИБ – например, полная конфиденциальность информации, а конечное имеет функционал, обеспечивающий другое – например, потерю конфиденциальности, то в процессе преобразования Представлений, очевидно, появляется уязвимость, приводящая к утечке информации. Отсюда следует несколько парадоксальное утверждение, что если изначальное Представление (по содержанию) никак не уточняет состояние задачи в части ИБ (например, не стремится обеспечить конфиденциальность), то утечка информации в конечном Представлении вполне допустима и не будет ассоциироваться с уязвимостью.

Это утверждение верно с оговоркой на недекларированные условия использования, которые следует понимать как аналогию недекларированных возможностей (НДВ) в том смысле, что программу начинают использовать не в соответствии с первоначальным замыслом, не по предназначению, т. е. не в задуманных условиях.

Здесь можно привести достаточно яркий «антипример» того, что сегодня называется уязвимостями протоколов стека TCP/IP. 2 ноября 1988 г. сеть ARPANET была атакована программой, впоследствии получившей название «червь Морриса» – временно из строя было выведено 6 800 компьютеров. Для этого использовалось несколько особенностей функционирования сетевых сервисов, а также некоторые «слабые» места компьютерных систем, впоследствии названные уязвимостями. С учетом того, что ARPANET, явившаяся прототипом сети Интернет, была создана в далеком 1969 г. (за 20 лет до «легендарной» DoS-атаки), то эти «уязвимости» было закономерно обусловленные недостаточным вниманием к вопросам ИБ в то время.

Искусственное расширение назначения и изменение среды функционирования хотя и относится к условиям применения, однако, по сложившейся терминологической практике, приписывается к слабостям и/или дефектам программ – то есть, к его уязвимостям в широком смысле слова; хотя проблемы возникали не в самом ПО, а в его использовании. С учетом этих соображений уточним введенное выше в узком (контекстном, ассоциированным с Представлениями) смысле понятие уязвимости как «отличия между начальным и конечным Представлениями»: уязвимость – злоумышленно или случайно внесенный дефект (программная закладка) или НДВ, которые в любом случае проявляются искажением содержания относительно предыдущих Представлений. А так как у замысла нет предыдущего Представления, поэтому он может считаться «неуязвимым».

Несмотря на отмеченную парадоксальность, все же можно предполагать, что введенные понятия более четко и глубоко очерчивают границы предметной области исследования по сравнению с привычными [3], в том числе стандартизованными[5].

Следствием этого является более ясное понимание как причин, точек и последствий появления уязвимостей, так и способов их обнаружения. Исходя из особенностей введенной системы Представлений (а именно: в начальном Представлении уязвимости отсутствуют, Представления сами по себе не изменяются, каждое новое Представление получается из старого путем специального процесса создания программы), можно сделать нижеследующие принципиальные выводы.

Во-первых, каждое новое (текущее, очередное) Представление содержит все уязвимости старого (предыдущего) за двумя исключениями (о которых речь пойдет ниже), а также дополнительные, внесенные в текущем процессе его создания. Как результат, конечное Представление хранит уязвимости, накопленные в процессе создания промежуточных Представлений.

Во-вторых, исчезновение уязвимостей в процессе создания программы возможно в двух случаях: неявно при создании нового Представления и целенаправленно специальными действиями. Первый случай соответствует случайному событию, когда отличие в Представлениях А и Б зеркально противоположно отличию в Представлениях Б и В, что приводит к отсутствию каких-либо отличий в Представлениях А и В (эффект «двойной ошибки»); такая ситуация может проявляться в результате антропоморфического взаимодействия как отдельных уязвимостей [4, 5], так и целых информационных подсистем [6–9]. Второй случай означает применение на стадиях создания программы дополнительных процессов обнаружения и нейтрализации уязвимостей.

В-третьих, поскольку создание программ идет по пути уменьшения уровня абстракции в Представлениях, то очевидно, что любая возникающая уязвимость, построенная на таких же сущностях, должна иметь подобный уровень, что и является ее статическим свойством. Динамическое же свойство уязвимости в виде ЖЦ может быть определено на базе Представлений, как путь от ее возникновения до обнаружения.

Согласно последнему сделанному выводу относительно уровней абстракций Представлений можно утверждать, что способ классификации уязвимостей может строиться (в том числе) на структурном уровне их «зарождения» нахождения в программе. Также в классификации целесообразно учесть такую ее сущность, как отличие со-

[5] ГОСТ Р 56546-2015. Защита информации. Уязвимости информационных систем. Классификация уязвимостей информационных систем. М.: Стандартинформ, 2018.

держаний, и скоррелировать с классической триадой нарушений конфиденциальности, целостности и доступности информации. Конкретная же таксономия уязвимостей будет дана далее.

В реальной практике многие Представления для разных программных разработок могут субъективно опускаться, объединяются с другими или выносятся в среду выполнения. Так, небольшая программа вполне может быть создана путем написания исходного кода и его компиляции в машинный, выполняться напрямую на центральном процессоре. Однако, если такое упрощение и оправдано для создания простейших программ, то для крупных проектов подобная практика негативно скажется на обеспечении ИБ обрабатываемых данных, поскольку не позволит обеспечить эффективный поиск уязвимостей. Естественно, в ходе общей эволюции программной инженерии могут появляться и новые Представления, качественно отличные от текущих.

3. КРАТКИЙ ОБЗОР СОВРЕМЕННЫХ ИССЛЕДОВАНИЙ В ОБЛАСТИ МОДЕЛИРОВАНИЯ ЖИЗНЕННОГО ЦИКЛА

Для оценки степени проработанности предметной области проведем краткий обзор научных работ, посвященных моделированию ЖЦ с уязвимостями, указывая при этом выводы в аспекте решения текущей задачи. Работа [10] посвящена модели цикла разработки программ, используемых в обучении. Исходя из такого предназначения, подтверждаемого жизненной практикой автора, предлагается коэволюция учителей и разработчиков. Указывается, что для включения некоторого ПО в процесс обучения может потребоваться его адаптация: параметрическая (настройкой параметров), модульная (выбором элементов из существующих) и конструктивная (разработкой недостающих элементов). Также, отмечается важность коммуникации участников процесса создания ПО. Выводы: должны присутствовать научные стадии (со стороны учителя), которые лишь потом переходят в более технические (со стороны разработчика); возможно объединения некоторых Представлений (например, предназначенных для разработчиков и программистов); необходимость в коммуникации логично ведет и к появлению промежуточных Представлений на базе различных терминологических аппаратов, способствующих участникам разработки (оперирующих с различными формами) передавать идею программы (т. е. ее содержание).

В статье [11] исследуется ЖЦ программных продуктов. Приводятся стадии создания автоматизированных систем в соответствии с ГОСТ 34.601 – 90 «Информационная технология. Комплекс стандартов на автоматизированные системы. автоматизированные системы. Стадии создания». Помимо этого, авторы статьи выделяют собственные стадии ЖЦ продукта: планирование, анализ и формирование требований, проектирование, разработка, тестирование, отладка, внедрение, эксплуатация и сопровождение. Также указываются классические модели ЖЦ (каскадная, V-образная, спиральная и др.). В работе обосновывается, что применение компьютерных методов обучения (и в особенности, ситуационных) позволяет более эффективно получать профессиональных пользователей программного продукта, упрощая тем самым стадию его внедрения. Выводы: помимо чисто технических стадий по созданию продуктов, могут иметь место и более высокоуровневые – концептуальные и архитектурные; процесс формирования требований является достаточно крупным и может занимать существенное место в ЖЦ; обучение пользователей является достаточно сложной и неотъемлемой частью внедрения программного продукта, поскольку, скорее всего, даже при четко сформулированной идее программы и безошибочной ее реализации различия в понятийных аппаратах не позволят пользователям работать с продуктом также «логично и безошибочно», как создателю.

Статья [12], являясь одной из множества от молодых авторов и, по совместительству, разработчиков программных продуктов, описывает подробный процесс создания программы с достаточно распространенным функционалом (в данном случае это «личный блокнот для записей мыслей»). Вначале автор провел опрос и собрал мнения о будущем продукте – сформирована его некоторая идея, затем были визуализированы действия с продуктом в виде диаграммы прецендентов с выделением основных функций – создан некий аналог концептуальной модели и базовой архитектуры, далее была получена диаграмма активности, детализирующая операции и прецендеты – алгоритмы записаны в блок-схемной форме (параллельно описан интерфейс продукта), после этого была проведена непосредственная реализация кода алгоритмов на Python и компиляция в выполняемый код – путем получения машинного кода с помощью утилиты Pyinstaller. Также автор провел тестирование продукта, что можно не относить к циклу разработки программ. Выводы: даже процесс создания простейшего программного продукта может быть разложен в виде создания его характеризующих представлений – идеи, концептуальной модели, архитектуры, алгоритмов, исходного и машинного кода.

Работа [13] посвящена базовым основам *программной инженерии* – т. е. проектированию, кодированию, распространению и поддержке ПО. В рамках методологии инженерии авторы указывают основные этапы ЖЦ программы: спецификация требований, проектирование, кодирование, тестирование, документирование и сопровождение.

Приведены такие стандарты программной инженерии, как ISO/IEC 12207 – Information Technology – Software Life Cycle Processes, SEI CMM – Capability Maturity Model (for Software), PMBOK – Project Management Body of Knowledge, SWBOK – Software Engineering Body of Knowledge и ACM/IEEE CC2001 – Computing Curricula 2001. Авторы подчеркивают, что ЖЦ ПО считается период его жизни от принятия решения о создании до полного изъятия из эксплуатации; такой цикл разбивается на последовательность процессов, имеющих собственные цели. Выводы: существует достаточное количество стандартов программной инженерии; классически под ЖЦ понимается не только период создания программ, но и их дальнейшее сопровождение (*прим. от автора* – в данной статье момент после создания выполняемой версии программы не рассматривается); программа проходит через несколько обособленных и довольно четко разграниченных стадий.

Цель работы [14] – ответ на 4, фундаментальных для ИБ программ, вопроса, касательно уязвимостей: 1) как они вносятся; 2) по каким причинам; 3) насколько долговечны; 4) как нейтрализуются. Данное исследование имеет непосредственное отношение к текущему, как раз и посвященному ЖЦ программ с уязвимостями. Источником информации в исследовании послужило описание уязвимостей из открытого источника – NVD (аббр. от англ. National Vulnerability Database – Национальная База Уязвимостей). Частичные ответы на указанные вопросы следующие: 1) обнаруженная уязвимость, как правило, появляется не из-за одной ошибки разработчика, а является следствием накопления «слабых» (с позиции безопасности элементов); 2) большинство уязвимостей появляется уже после 1 года начала проекта и сохраняются вплоть до 30-дневного срока до релиза, что объясняется высокой нагрузкой разработчиков в этот период; 3) средняя живучесть уязвимостей чуть менее 2-х лет, а вероятность нейтрализации уязвимостей начинает увеличиваться после 1 года с последнего внесенного исправления; 4) исправления, как правило, включают один файл, число добавленных строк в котором превышает в 2 раза число удаленных, а наиболее частым подходом устранения уязвимости является проверка входных данных. Также, в статье делаются следующие основные заключения: уязвимости возникают вскоре после создания новых файлов и добавляются туда при их сохранении; основной вклад уязвимостей вносят разработчики с высокой нагрузкой; живучесть уязвимостей оценивается, как высокая. Выводы: уязвимости появляются уже на первых стадиях разработки программ (т. е. до исходного кода, например, в архитектуре); их нейтрализация существенно затруднена на более поздних стадиях разработки (например, в исходном коде); возможность исправление уязвимостей со временем растет, вероятно, по причине перехода на Представления программы, более раннее, чем исходный код (т. е. на те, в которых уязвимость была внесена); затрудненность вносить исправления в текущем Представлении может быть следствием того, что недостатки в программе (которые пока еще не переросли в уязвимости) появились в предыдущих Представлениях, а понятийные аппараты этих Представлений (а значит и недостатков) качественно отличаются.

Избегая лишней скромности, необходимо упомянуть и собственные авторские работы, непосредственно посвященные ЖЦ программ, поскольку они легли в основу текущего исследования. Так, в [15] описан цикл, состоящий из Представлений, упоминаемых в выводах к [12]; при этом, для Представлений указаны 3 типа уязвимостей по глубине их нахождения в абстракции кода – высоко-, средне- и низкоуровневые. В [16] делается такая же «разбивка» ЖЦ на Представления; однако, основная часть работы посвящена более глубокому изучению переходов между Представлениями с позиции формы и содержания программы.

4. СХЕМА ЖИЗНЕННОГО ЦИКЛА ПРЕДСТАВЛЕНИЙ ПРОГРАММЫ С УЯЗВИМОСТЯМИ

Всесторонний и кропотливый анализ элементов предметной области (включая выводы, сделанные по обзорам статей) – стадий и процессов создания программ, возможных Представлений, дерева их эволюции, различных парадигм программирования, способов Представления алгоритмов, языков программирования, сред исполнения, типизации уязвимостей и их точек возникновения и обнаружения – с последующей систематизацией и частичной группировкой позволил создать достаточно обобщающую схему ЖЦ Представлений программы (далее – Схема).

Граф эволюции Представлений программы (с делением на стадии и процессы создания) с потенциально возникающими уязвимостями, который и составляет обозначенную Схему, представлен на рисунке 1. На Схеме узлы графа являются существующими Представлениями (имеют вид прямоугольников с синим фоном), а ребра (стрелками отмечено направление) – пути их создания из предыдущих. Так, граф начинается с Представления Идеи программы (№ 1 на схеме), и заканчивается 3-мя Представлениями: Машинным (№ 14 на схеме), Сценарным (№ 11) и Байт-кодом (№ 15), каждое из которых готово для непосредственного выполнения с помощью специальных программно-аппаратных механизмов (имеют вид прямоугольников с боковыми линиями и желтым фоном, к которым ведут указатели прямоугольного вида; механизм прямоугольника с более тусклым фоном включает остальные механизмы).

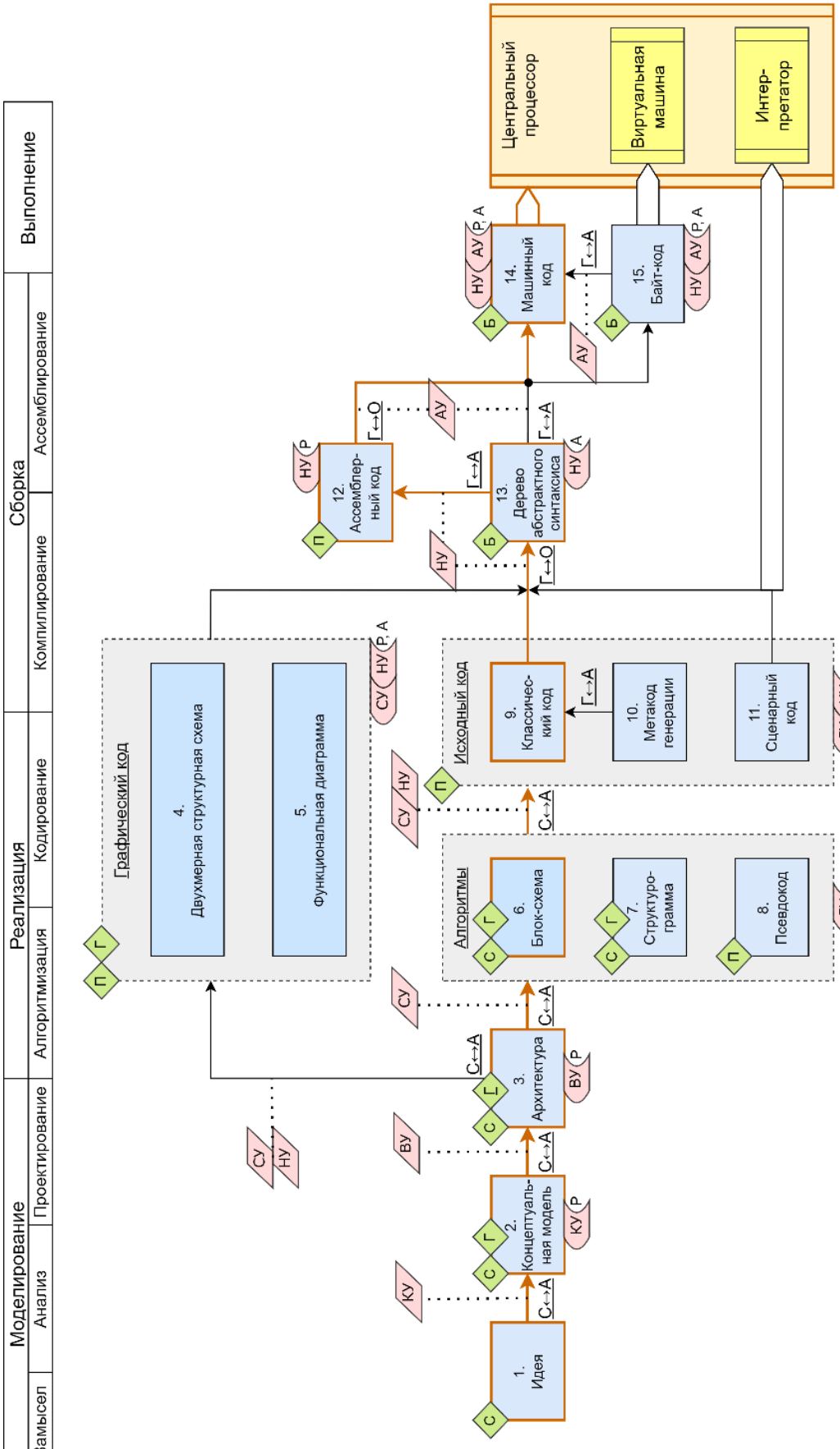

Рис. 1. Схема жизненного цикла Представлений программы с уязвимостями

Fig. 1. Life Cycle Scheme of Program Representation with Vulnerabilities

Согласно схеме, Представление № 14 исполняется на Центральном процессоре, для работы Представления № 15 необходима специализированная Виртуальная машина (имеющая, как правило, вид программного продукта и представляющая собой аналог реальной аппаратной платформы), а для выполнения Представления № 11 достаточно наличие Интерпретатора (часто, построчно выполняющего код). Некоторые Представления объединены в группы (имеют вид блоков с пунктирными границами и серым фоном) – входящее в группу и выходящее из нее ребро продлевается по умолчанию соответствующим образом на каждый элемент группы (за исключением Метакода генерации, из которого, как правило, может продуцироваться только Классический код).

Исходя из Схемы и ряда предыдущих умозаключений, предложим следующую 3-х компонентную классификацию уязвимостей.

Подкласс уязвимости – «структурный уровень»

Схема отражает уязвимости различных классов согласно их структурным уровням (что было обосновано ранее), а именно, следующие.

Во-первых, это концептуальные уязвимости (далее – КУ), заложенные в программу практически на первых стадиях ее создания; например, отсутствие учета понятий, связанных с аутентификацией пользователя (очевидно, если изначально не была указана столь базовая сущность, то и в архитектуре вряд ли она может появиться).

Во-вторых, это высокоуровневые уязвимости (далее – ВУ), такие, как ошибки в архитектуре программы: нарушение общих принципов ее функционирования, низкий уровень механизмов обеспечения ИБ и т. п.

В-третьих, это среднеуровневые уязвимости (далее – СУ), такие, как неверная реализация алгоритмов подпрограмм, передачи входных параметров, возврата из нее и т. п.

В-четвертых, это низкоуровневые уязвимости (далее – НУ), такие как ошибки в вычислениях, структурах данных, доступе к ним и т. п.

И, в-пятых, это атомарные уязвимости (далее – АУ), такие как ошибки в действиях для текущей среды выполнения программы, с помощью которых реализуются «низкоуровневые» вычисления и хранение данных; например, неверный выбор инструкций процессора или его регистров при «развертывании» операции сложения переменных для Центрального процессора.

Такое деление уязвимостей является более чем логичным, поскольку каждый новый класс присущ своей стадии разработки. Пожалуй, единственным исключением являются Представления группы Исходного кода, поскольку в них могут возникать одновременно СУ и НУ, которые также присущи предыдущему и последующему Представлениям. Впрочем, это объясняется тем, что работа с Исходным кодом занимает основное время разработки любой программы, и оно, так или иначе, поглощает в себя окружающие Представления. Особое место в Схеме занимают Представления Графического кода, поскольку они объединяют в себе как Алгоритмы, так и сам Исходный код; соответственно, это ведет и к появлению в них одновременно СУ и НУ.

Подкласс уязвимости – «изменение содержания»

В связи с тем, что понятие уязвимости было введено, как отличие между Представлениями (в общем смысле – между начальным и конечным, а в частном – последовательность отличий между промежуточными Представлениями), то появление уязвимости в некотором Представлении можно понимать, как факт отличия содержания текущего Представления от Предыдущего. Саму же суть понятия «отличие» можно трактовать через результат сравнения множеств из элементов этих Представлений; что позволит расширить таксономию уязвимостью следующими подклассами:

1) во втором Представлении отсутствует элемент содержания, который был в первом – уязвимость подкласса «потеря функционала»;

2) во втором Представлении присутствует новый элемент содержания, который отсутствовал в первом – уязвимость подкласса «внесение функционала»;

3) элемент содержания второго Представления отличается от аналогичного элемента первого – уязвимость подкласса «модификация функционала».

Наглядная графическая интерпретация такой типизации подклассов уязвимостей представлена на рисунке 2.

Соответственно, совпадение множеств означает, что уязвимость не возникла (по крайне мере, при преобразовании этих двух Представлений); следовательно, ее подкласс отсутствует, поскольку функционал программы не изменился.

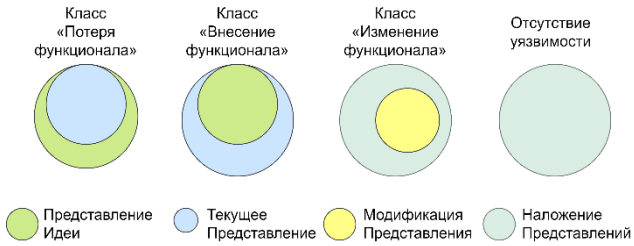

Рис. 2. Графическая интерпретация классификации уязвимостей по изменению содержания

Fig. 2. Graphical Interpretation of the Vulnerabilities Classification by Content Change

Необходимо отметить, что уязвимость 3-го подкласса в некотором смысле тождественна уязвимостям первых двух подклассов, поскольку удале-

ние одного элемента множества и добавление другого может быть истолковано как его изменение; данную коллизию разрешим соображением минимизации количества уязвимостей путем приоритета выбора для них одного 3-го подкласса вместо первых двух.

Подкласс уязвимости – «воздействие на информационные потоки»

Поскольку суть действия любой программы заключается в некотором изменении информации (что верно не только для императивных парадигм программирования – строго описывающих алгоритм работы, но даже и для декларативных – описывающих желаемый результат без составления шагов его достижения), то внесение изменений в содержание программы должно иметь некоторый эффект и на итоговую обработку информации. Так, например, удаление из программы проверок входных параметров приведет к появлению новых иноформенных потоков, не обрабатываемых ранее, и нарушению ее работы. С другой стороны, неверная реализация комплексной логики проверки параметров (например, ее реализация на основе логического И вместо логического ИЛИ) приведет к тому, что при корректных входных данных программа будет выдавать ошибку. Возможна и третья ситуация, отчасти близкая к комбинированию первых двух, когда вместо проверки параметра на один набор значений используется другой – т. е. информационный поток с верными входными данными исчез, а поток с неверными данными появился (например, ошибочная проверка установленного старшего бита числа вместо младшего). Вышесказанные размышления обосновывают введение еще одного способа классификации, наиболее субъективного из всех (хотя и продолжающего иметь достаточный уровень формализации), а именно – воздействие на информационные потоки вследствие внедрения уязвимости. Важно заметить, что эти классы достаточно хорошо коррелируют с классической триадой нарушений конфиденциальности, доступности и целостности (хотя и действуют на более «тонком» уровне – вместо глобального влияния всей программы на данные учитывается локальное влияние условно незначительных изменений ее содержания на внутренние информационные потоки), и, поэтому, будут введены подобным образом:

– аналог нарушения конфиденциальности информации, поскольку данные попадают к несанкционированным для этого обработчикам – уязвимость подкласса «появление информационного потока»;

– аналог нарушения доступности, поскольку существующие данные не доходят до санкционированных для этого обработчиков) – уязвимость подкласса «потеря информационного потока»;

– аналог нарушения целостности, поскольку данные доходят до санкционированных обработчиков, но в измененном виде – уязвимость подкласса «изменение информационного потока».

Объединение трех способов классификации уязвимостей (структурный уровень, изменение в содержании и воздействие на информационные потоки) как раз и определяет общую их таксономию, полностью построенную на базисе предложенной Схемы, что говорит о целостности вводимого методологического аппарата. Классификации могут применяться независимо, что позволяет их считать вложенными. Так, класс любой уязвимости определяется совокупностью введенных подклассов.

Возникновение уязвимостей (имеющих вид параллелограммов с красным фоном) в процессе создания Представлений отражено в Схеме связывающими их пунктирными линиями (см. рисунок 1). Уязвимости, которые могут быть найдены в каждом из Представлений, отражены примыкающими к каждому Представлению сверху или сверху (имеют вид прямоугольника с гнутыми боками и красным фоном). Способ обнаружения уязвимостей в каждом Представлении может иметь как *ручной* тип – субъективно человеком, так и *автоматический* – специализированными программными средствами; естественно, возможно их объединение. На Схеме тип(ы) поиска уязвимостей указан(ы) справа от обозначения уязвимости (в вогнутой части прямоугольника).

Исходя из исторически сложившейся практики программирования, а также авторского опыта участия в создании крупных программных проектов, можно выделить следующие стадии создания программ и составляющие их процессы, расположенные как условная временна́я ось в верхней части Схемы.

Стадия *моделирования* – состоит из замысла (Идеи), разработки Концептуальной модели решения и проектирования Архитектуры программы.

Стадия *реализации* – состоит из алгоритмизации (т. е. создания Алгоритмов решения) и кодирования Исходного кода; также возможно их объединение, создавая тем самым Графический код (совмещающий алгоритмы и их высокоуровневую реализацию).

Стадия *сборки* – состоит из компиляции Исходного кода в Ассемблерный (через внутренние структуры утилит – Дерево абстрактного синтаксиса), и их ассемблирования в Машинный; современная практика создания программ также позволяет компилировать Исходный код напрямую в машинный.

Стадия *выполнения* – не входит по определению в процесс создания программ, но может использоваться для его отладки и тестирования в специ-

альной среде, которая приведена на Схеме для указания на факт и средства выполнения конечных Представлений. Так, Центральный процессор предназначен для непосредственной интерпретации инструкций Машинного кода, Виртуальная машина производит выполнение Байт-кода, а Интерпретатор предназначен для выполнения Сценарного кода. Важно отметить, что фактически Виртуальная машина и Интерпретатор являются лишь комплексами программам, также исполняемыми Центральным процессором (что отмечено на Схеме вложенностью графических форм); тем не менее, в рамках моделирования программы они являются отдельными независимыми сущностями.

Термины, использованные выше для обозначения стадий и процессов, не всегда совпадают со стандартизованными[6] (некоторым из них около полувека), однако, по авторскому мнению, наилучшим образом (по аналогии с понятийным аппаратом) отражают суть и содержание исследуемого процесса.

Каждое из Представлений может принимать следующие формы или их комбинацию (имеют вид ромбов с зеленым фоном в левом верхнем Представления): *словесная* – с помощью естественного языка; *графическая* – в виде блок-схем, диаграмм, изображений графов; *программно-языковая* – с помощью формально-знакового текста на языках программирования; *бинарная* – в виде набора байтов (реже битов).

Получение последующих Представлений из текущих осуществляется с помощью превращений следующих типов: *синтез* – результат интеллектуальной деятельности человека, как правило, слабо формализуемый (например, получение из Блок-схем алгоритмов программы ее Исходного кода требует выполнение существенной исследовательской деятельности по выбору подходящих шаблонов и способов реализации логики); *генерация* – результат работы специализированных программных средств, как правило, основанный на заранее заложенных строгих правил преобразования (например, получение Ассемблерного кода из Исходного в упрощенном виде представляет собой перевод конструкций языка программирования через их внутреннее представление в текстовую запись соответствующих инструкций процессора).

Исходя из вышесказанного, логично ввести и обратные типы превращений (если существующий не практически, то хотя бы теоретически), а именно следующие: *анализ* – по аналогии с синтезом, также результат деятельности человека, но направленный на восстановление Представления (например, исходя из совокупного понимания Исходного кода подпрограммы можно восстановить и ее Алгоритмы в виде Блок-схем); *обратная генерация* – также, результат работы программных средств по формальным правилам, но в виде «реверс-инжиниринга» (например, восстановление Исходного кода из Машинного, называемое термином «декомпиляция», было описано автором в диссертационном исследовании [17, 18]). На Схеме получение каждого Представления из предыдущего указано в следующем формате: «$X \leftrightarrow Y$», где X – первая буква типа прямого преобразования (т. е. «А» или «С»), а Y – первая буква типа обратного преобразования (т. е. «Г» или «О»). Можно предположить, что тип превращений напрямую зависит от степени формализации Представлений, поскольку генерация (прямая и обратная) может производиться лишь при строгом соблюдении правил описания программы; в отличие от синтеза и анализа, где отсутствие формализованных правил преобразования нивелируется творческой составляющей эксперта.

Важно отметить, что, согласно предыдущему авторскому исследованию [19], в части учета когнитивного аспекта восприятия кода, содержание может быть декомпозировано на две компоненты: *логику* и *словарь (или базис)*. В этом случае содержание решения определяется некоторой логикой, построенной на базе словаря (например, Архитектура в виде логических модулей или Исходный код в виде конструкций языка). Форма в данном случае является связкой между словарем и способом взаимодействия с содержанием. При описании каждого Представления далее примеры компонент содержания будут пояснены на интуитивно понятных примерах.

5. ПРЕДСТАВЛЕНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ VS УЯЗВИМОСТИ

Приведем все существующие Представления с последовательным указанием следующих их характеристик: общее назначение, форма и содержание, связь с другими Представлениями, появление уязвимостей и их обнаружение, примеры Представлений и способы их выполнения (в случае наличия возможности). Это позволит создать законченную эволюционную картину возможных ветвей ЖЦ всех Представлений и уязвимостей, включая особенности их появления и исчезновения.

I. Идея (№ 1 на Схеме)

Назначением Идеи программы является отражение некоторого замысла программы и ее функционала с выделенными характеристиками и свойствами.

Форма Представления является словесной, поскольку отсутствует какая-либо возможность

[6] ГОСТ Р ИСО/МЭК 12207-2010 Информационная технология. Системная и программная инженерия. Процессы жизненного цикла программных средств. М.: Стандартинформ, 2011; ГОСТ 19.102-77 Единая система программной документации. Стадии разработки. М.: Стандартинформ, 2010.

формализации. Содержание задается совокупностью осмысленных предложений в базисе используемого разговорного языка.

Представление считается началом любой программы и создается как результат творческого процесса решения крупной задачи, его вектор. Оно может быть восстановлено из Концептуальной модели путем анализа, поскольку данное и последующее Представления практически не формализуемы.

Представление не может иметь уязвимостей (об этом выше), поскольку оно используется как будущий прообраз желаемого результата, т. е. является «идеальным», несмотря на все свои недостатки и просчеты. Как результат, поиск уязвимостей тут также не имеет смысла.

Описание Представления возможно любыми способами (преимущественно когнитивными), поскольку никакого конкретного вида на данном шаге решение пока не имеет. Типичным примером Представления может быть записанные (систематизированные) результаты «мозгового штурма».

II. Концептуальная модель (№ 2 на Схеме)

Концептуальная модель программы является его Представлением, определяющим основные понятия, их структуру, взаимосвязи и особенности с указанием направления решения и используемых подходов. Здесь необходимо выделение базовой смысловой структуры и соответствующего терминологического аппарата.

Форма Представления является словесно-схемной (или словесно-графической), как наиболее подходящей для описания обобщенных моделей с привязанным текстовым пояснением. Содержание задается совокупностью утверждений, логически связанных с помощью графической схемы (или схем).

Представление создается в результате синтеза возможностей по воплощению Идеи разрабатываемой программы путем первоначальной условной формализации в направлении выбранного решения. По причине все также слабой формализации текущего и последующего (Архитектуры) Представлений, данное может быть восстановлено только путем применения анализа.

В данном Представлении могут появляться КУ вследствие неверного концепта основной идеи. Их поиск на сегодня возможен только ручным способом по причине крайне высокого уровня абстракции и уникальности Представления, оперирующего понятиями.

По этим же причинам какие-либо стандартные виды Представления отсутствуют. Примером Представления могут быть любые модели достаточного уровня абстрактности и носящий определяющий для создаваемой программы характер, имеющие текущее назначение и наиболее подходящий вид.

III. Архитектура (№ 3 на Схеме)

Архитектура программы является его наиболее полным высокоуровневым Представлением в виде схемы, близкой к Исходному коду, описываемой с использованием специфических особенностей (технологий, языков программирования, шаблонов/парадигм, средств реализации, форматов и т. п.). Архитектура определяет общую структуру программы, его физические и логические модули, их взаимодействие. С этой позиции Архитектура задает зоны ответственности сущностей (а также их взаимосвязи) каждого понятия, введенного Концептуальной моделью.

Форма Представления является графической (со словесными комментариями). Содержание задается аналогичным с Концептуальной моделью образом, но с раскрытием ее базовых элементов в сторону непосредственной реализации.

Представление создается на основании Концептуальной модели в результате синтеза требований и возможностей с применением опыта эксперта-архитектора. Хотя последующие возможные Представления (Алгоритмы или Графический код) и обладают определенной степенью формализации, тем не менее, какие-либо удовлетворительные средства обратной генерации из них Архитектуры отсутствует, поэтому последняя восстанавливается с помощью ручного анализа.

В данном Представлении могут появляться ВУ вследствие ошибок в проектировании. Исходя из крайне сложной формализации и субъективности оценки, в большинстве случаев такие уязвимости могут быть найдены только ручным способом. Примером Представления может быть описание Архитектуры с помощью языков специального назначения ADLs (*аббр. от англ.* Architecture Description Languages – языки описания архитектуры) или их потенциального преемника в виде UML.

Различные пути эволюции алгоритмов до стадии выполнения позволяют объединить в группы их некоторые Представления, которые имеют по умолчанию общие входы и/или выходы, а также, как правило, одинаковые формы и способы поиска уязвимостей.

Группа графического кода (№№ 4–5 на Схеме) содержит в себе Представления, совмещающие алгоритмы программы и ее исходный код, расширяя область программирования из простого текстового в более наглядное – визуальное. Данная группа состоит из Двухмерной структурной схемы и Функциональной диаграммы, отличающейся от первой спецификой целевого применения.

Группа алгоритмов (№№ 6–8 на Схеме) позволяет разделить различные варианты Представления Алгоритмов программы: Блок-схема (в виде графа последовательности шагов решения зада-

чи), Структурограмма (в виде структуры блоков решения задачи) и Псевдокод (в виде текстового описания на специальном языке, еще не готовом для компиляции).

Группа Исходного кода (№№ 9–11 на Схеме) содержит различные варианты текстового решения задачи человеком на специальном языке программирования. Наиболее общепринятым и полноценным Представлением является код на классических языках (C/C++/C#/Java/Pascal и пр.), часть которого может быть создана из Метакода генерации с помощью автоматических средств. Сценарный код является упрощением Классического кода путем большего функционала внешних библиотек, минимизации синтаксиса и его лучшей наглядности.

IV. Двухмерная структурная схема (№ 4 на Схеме)

Представление используется в процессе двухмерного структурного программирования, совмещая в себе две важнейшие, и при этом принципиально разные функции: графическая визуализация алгоритмов выполнения и текстовая программная реализация его элементов. Таким образом, Представление считается симбиозом двух других – из групп Алгоритмов и Исходного кода. Представления может совместно использоваться при создании ПО как *инженерами* (не являющимися программистами), описывающими более абстрактную логику решения задачи, так и *программистами*, задающими ее менее абстрактную и детальную реализации: таким образом, Представление применимо в качестве языка общения и взаимодействия специалистов этих двух типов.

Формой Представления является совокупность графической и программно-языковой форм – *программно-графическая*. Содержание задается совокупностью следующих слоев: логикой работы блоков программы в базисе стандартных графических элементов, и текстового наполнения этих элементов деталями реализации их функционала в базисе некоторого языка программирования.

Представление создается в результате синтеза алгоритмов из Архитектуры программы с последующей их конкретизацией на языке программирования. Видоизменения Представления могут производиться одновременно как на графической составляющей, так и на текстовой, поскольку они представляются жестко-связанными сущностями. Двухмерная структурная схема может быть восстановлена из Дерева абстрактного синтаксиса путем обратной генерацией, поскольку Дерево частично содержит необходимую для этого информацию (деление на подпрограммы, именованные переменные, структуры циклов и ветвлений и т.п.).

В данном Представлении могут появляться как СУ, так и НУ. Первые связаны с неверными алгоритмическими решениями задач, определенных архитектурой (т. е. с инженерами), вторые же – с их реализацией или интерпретацией (т. е. с программистами). Применимы и ручные, и автоматические способы поиска уязвимостей. Первые более захватывают область СУ, вторые же – НУ. Примером Представления могут быть схемы языка ДРАКОН [20, 21] или используемые в Р-программировании, а также некоторые диаграммы активности языка UML [22].

V. Функциональная диаграмма (№ 5 на Схеме)

Представление является аналогом Двухмерного структурного в части симбиоза, но имеет собственное применение, что вносит определенные существенные отличия и в итоговый функционал программы. Относится к разряду графических языков, предназначенных в основном для программирования логических контроллеров, применяемых для автоматизации промышленного производства и ориентированных на работу посредством сигналов ввода/вывода. В отличие от Двухмерного структурного, Представление может быть полноценно использовано специалистами одного типа – *инженерами* логических контроллеров.

Впрочем, качественных отличий в принципах формирования формы и содержания от Двухмерной структурной схемы нет – используется графическая форма записи логики работы блоков программы с программно-языковой формой реализацией их функционала.

Согласно декларации о группировке Представлений, Функциональные диаграммы получаются из Архитектуры в результате синтеза и могут быть восстановлены из Дерева абстрактного синтаксиса путем обратной генерации. Аналогичным образом, диаграммы могут иметь как СУ, так и НУ, для поиска которых применимы как ручные, так и автоматические способы. Примером Представления могут быть языки программирования FBD (*аббр. от англ.* Function Block Diagram – функциональные диаграммы блоков) [23], SFC (*аббр. от англ.* Sequential Function Chart – последовательные функциональные схемы) [24] и LD (*аббр. от англ.* Ladder Diagram – релейная логика) [25].

VI. Блок-схема (№ 6 на Схеме)

Одной из канонических форм Представления Алгоритмов является графическая в виде Блок-схем со словесным содержанием ее элементов (совместно, в базисах разговорного и программного языков). По сути, Представление переназначено для частично формализованного описания алгоритмов (или процессов) в «гостированном» для Российской Федерации виде.

Представление создается в результате синтеза алгоритмов из Архитектуры, явно следуя назначению ее модулей или подсистем. Поскольку каждый

алгоритм решает определенную задачу, зачастую являющуюся уникальной и/или имеющую особые требования, то процесс создания представляется полностью творческим. Восстановление Блок-схемы, как правило, осуществляется экспертом в результате анализа [26] (например, в рамках изучения Исходного кода на предмет СУ).

В данном Представлении могут появляться СУ вследствие неверного решения алгоритмами подзадач, определенных модулями. По данному Представлению возможен поиск СУ, поскольку оно не определяет Архитектуру (имеющую ВУ) и не содержит конкретной реализации алгоритмов (имеющих НУ), а находится между этими уровнями абстракций. Способ поиск уязвимостей является исключительно ручным. Примером Представления являются блок-схемы алгоритмов в соответствии с ГОСТ 19.701-90 (см. сноску 4).

VII. Структурограмма (№ 7 на Схеме)

Структурограмма является своеобразным аналогом Блок-схемы и предназначена для описания алгоритмов, составленных в соответствии со структурной парадигмой программирования. Последнее означает, что в данном Представлении должны отсутствовать безусловные переходы, на которых с формальной точки зрения построены Блок-схемы (в них безусловный переход является основным способом связывания элементов).

В остальном используемые форма и содержания Представлений под №№ 6 и 7 совпадают.

Согласно декларации о группировке Представлений, способы получения и восстановления Структурограмм такие же, как и для Блок-схем. В ней также могут возникать СУ, хотя для ручного поиска используются несколько иные подходы, делая его более эффективным. Примером Представления могут быть диаграммы Насси – Шнейдермана [27].

VIII. Псевдокод (№ 8 на Схеме)

Псевдокод, хотя и отнесен к группе Алгоритмов, тем не менее имеет существенные отличия от остальных Представлений группы.

Форма Представления является программно-языковой, что позволяет дополнительно применять автоматические средства поиска уязвимостей. Содержание задается совокупностью описания логики выполнения блоков программы в базисе некоторого алгоритмического языка (например, С-подобного, но без использования типов переменных и с заменой тела некоторых подпрограмм на соответствующие человеко-ориентированные комментарии).

В остальном способы получения и восстановления Представления, а также уязвимости и их поиск, аналогичны другим элементам группы Алгоритмов. Примером Представления может быть учебный алгоритмический язык и, с некоторыми поправками, семейство языков Algol (*сокр. от англ.* Algorithmic Language – алгоритмический язык) [28]. В современном понимании любой императивный язык программирования является алгоритмическим, однако такая трактовка здесь лишь приведет к подчеркнутому смешению терминологии и понятийной базы.

IX. Классический (исходный) код (№ 9 на Схеме)

Классический исходный код является основным «рабочим» Представлением, используемым программистами, и последним, подверженным влиянию человека. Использование термина «классический» призвано отделить данное Представление от других – Метакода генерации и Сценарного кода, имеющим иное предназначение и существенно более сложны парадигмы программирования. Назначением Представления является реализация алгоритмов на выбранном языке программирования с последующей отладкой, поддержкой, документированием и т. п.

Форма Представления является программно-языковой и достаточно точно отражает содержание Алгоритмов, поскольку определяет их пошаговую конкретную реализацию. Содержание задается детальным описанием реализации программы в базисе выбранного языка программирования.

Следуя классической схеме программной инженерии, Исходный код получается из Алгоритмов (чаще – стандартизованных Блок-схема и Структурограмм, реже – из Псевдокода) в процессе рутинной деятельности типового программиста. Более частным случаем является получение Представления из Метакода путем применения специальных генераторов (некоторые детали этого будут описаны далее). Восстановить же Представление можно аналогичным с Графическом кодом образом – из Дерева абстрактного синтаксиса; впрочем, в случае Исходного кода данное восстановление является более распространенным. Так, например, различного рода декомпиляторы часто вначале по Машинному коду строят внутреннее Дерево абстрактного синтаксиса (с дополнительно навешанными на него связанными структурами), производят на нем оптимизирующие и гармонизирующие (для лучшего восприятия результата человеком) действия, а затем генерируют Классический исходный кода или его аналоги [29].

Поскольку Представление является соединительным звеном между собственно алгоритмами и их итоговой низкоуровневой реализацией, то в нем могут появляться как НУ, так и СУ вследствие ошибок в реализации алгоритмов или их неверной интерпретации программистом. Такие же классы уязвимостей могут появляться и при ошибках в реализации утилит, генерирующих новый код по Метакоду. Способы поиска уязвимостей примени-

мы как ручные, так и автоматические. Первые более захватывают область СУ, вторые же – НУ. Примером Представления может быть любой большого множества средне- и высокоуровневых императивных языков программирования (поскольку единого универсального языка еще пока не создано [30]), наиболее яркими представителями которого являются Pascal, Java и C/C++/C#-подобные.

Несмотря на то, что изначально Классический исходный код не создавался для непосредственного выполнения, однако существуют проекты, которые с некоторыми ограничениями позволяют это делать для некоторых языков программирования. Так, проект Picoc (https://github.com/jpoirier/picoc) представляет собой достаточно миниатюрный интерпретатор языка C (естественно, без полноценной поддержки работы с указателями и других платформенно-зависимых особенностей программ).

X. Метакод генерации (№ 10 на Схеме)

Представление является промежуточным, предназначенным для генерации Исходного кода (в преобладающем большинстве – Классического).

Оно имеет программно-языковую форму и часто соответствует декларативной парадигме программировании (что является допустимым исключением из рамок введенных ранее понятий). Содержание задается аналогичным для Классического кода способом. Представление косвенно создается по Алгоритмам программы и, как правило, содержит правила для генерации отдельных частей реализации шагов решения задачи.

В остальном Представление и его уязвимости аналогично Классическому исходному коду. Исключением является теоретическая возможность восстановления Метакода из созданного по нему Классического кода, что, впрочем, крайне трудоемко и имеющий малую практическую значимость.

Примером Представления могут быть формальные грамматики, заданные с помощью синтаксиса для генераторов лексического анализатора Lex (Flex для GNU) и синтаксического анализатора Yacc (Bison для GNU) [31].

XI. Сценарный код (№ 11 на Схеме)

Представление Сценарного кода достаточно близко к Исходному (и имеет такие же форму и содержание), хотя и оперирует более крупными программными компонентами и функционалом; такое отличие в конечном итоге находит отражение, как в синтаксисе кода, так и в его дальнейших превращениях и заключительном выполнении.

Согласно декларации о группировке Представлений, Сценарный код может быть получен из Алгоритмов в результате синтеза и восстановлен из Дерева абстрактного синтаксиса путем обратной генерации; он может иметь как СУ, так и НУ, для поиска которых применимы как ручные, так и автоматические способы [32]. Примером Представления могут быть скриптовый язык JavaScript, интерпретируемый язык Perl, сценарий командной строки Shell Script [33].

Код данного представления готов для непосредственного выполнения соответствующим интерпретатором.

XII. Ассемблерный код (№ 12 на Схеме)

Представление задает конечные регистры и инструкции для выполнения процессором; оно также содержит пользовательские имена, сегменты с данными, а также ряд другой более специфичной информации.

Форма Представления является программно-языковой, которая уже в меньшей степени ориентирована на человека, но пока еще не подходит для выполнения машиной. Содержание задается описанием логики работы аппаратной части в базисе синтаксиса инструкций процессора.

Представление генерируется из Дерева абстрактного синтаксиса (которое будет описано далее) и зачастую не используется, поскольку конкретные машинные инструкции, готовые для выполнения, могут быть созданы без промежуточного Ассемблерного кода. Восстановить Ассемблерный код можно из Машинного в процессе дизассемблирования, что в общем случае является тривиальной и хорошо отлаженной программной процедурой.

В данном Представлении могут появляться НУ вследствие неверной работы модуля генерации (по предыдущему представлению) или низкоуровневой оптимизации Машинного кода; впрочем, вероятность этого крайне мала. По данному Представлению возможен поиск НУ трудоемким ручным способом; в ряде случаев применяется автоматизации. Однако вследствие того, что Ассемблерный код считается промежуточным, их поиск откладывается на последующие Представления. Примером Представления может быть любой машинно-ориентированный язык низкого уровня для заданного процессора, например: MASM и TASM [34] для семейства x86, vasm для семейств PowerPC и ARM.

Хотя, как было сказано, код данного Представления классически не считается выполняемым; тем не менее существуют проекты, которые позволяют это делать, даже непосредственно через Интернет-ресурсы (например, https://www.tutorialspoint.com/compile_assembly_online.php или https://www.jdoodle.com/compile-assembler-nasm-online/).

XIII. Дерево абстрактного синтаксиса (№ 13 на Схеме)

Представление является промежуточным между Исходным кодом, ориентированным на челове-

ка, и автоматоориентированным Ассемблерным, а в последствие и Машинным, кодом. Оно, как правило, используется для преобразования кода в процессе работы соответствующих программных средств – *компиляторов*, при этом, частично отбрасывая ненужную для работы программы мета-информацию (например, комментарии к исходному коду, типы переменных и т. п.).

Форма Представления достаточно сильно зависит от конкретной реализации компилятора, но, как правило, она имеет бинарный вид; хотя в ряде случает Дерево абстрактного синтаксиса может храниться во внешних файлах XML- или JSON-формата. Соответственно, способы задания содержания чаще всего имеют вид связей между разнотипными узлами (т. е. представляют множество классических графов и/или таблиц).

Исходя из назначения Дерева абстрактного синтаксиса, оно получается из Исходного кода (в процессе компиляции). Восстановить Представление можно из последующих Представлений – Ассемблера, Машинного кода и Байт-кода. Способ восстановления, как правило, является ручным (т. е. требует применения экспортного анализа), хотя и существует некоторый пул программных средств, проводящих такую процедуру – *декомпиляцию* (следуя названию – процесс, обратный компиляции). Тем не менее, отсутствие теоретических и практических возможностей полноценной реализации декомпиляторов (с учетом ограниченности поддерживаемых входных нотаций Машинного, Ассемблерного или Байт-кода, и получаемых синтаксисов Исходного кода) при крайней потребности в подобных средствах реверс-инжиниринга, переводят данную задачу в разряд проблемы, остро стоящей для области ИБ в части поиска уязвимостей в программах. В подтверждение этого приведем здесь ссылку на страницу авторского сайта с указанием некоторых наиболее известных на данный момент декомпиляторов: http://demono.ru/links/decompilers.

Особенность Представления заключается в хранении древовидной структуры кода (с более сложными связями между узлами), что позволяет решать определенные задачи более эффективно – например, производить различные типы оптимизации. Также Представление позволяет осуществлять более глубокий анализ программ непосредственно перед и во время выполнения. Уязвимости и их поиск совпадают с аналогичными у Ассемблерного кода, хотя сами способы будут более эффективным. Так, для Байт-кода языка Python существует инструмент Bandit [35], который использует плагины для обработки Дерева абстрактного синтаксиса на предмет поиска проблем в безопасности.

Примерами Представления, в общем случае, могут служить любые описания Исходного кода с помощью Дерева абстрактного синтаксиса в подходящем для этого формате. Как правило, внутренняя структура такого Дерева в каждом программном средстве обработки Исходного кода собственная.

XIV. Машинный код (№ 14 на Схеме)

Представление является наиболее «дальним» от Идеи и представляет собой нечитабельные данные, строго задающие выполнение инструкций процессора.

Форма Представления является бинарной. Содержание Представления, аналогично Ассемблерному коду, задается описанием логики работы аппаратной части, но уже в базисе кодирования инструкций процессора.

Представление генерируется по Ассемблерному коду с применением специальных программных средств – *ассемблеров*, полностью опуская всю метаинформацию, ненужную для выполнения процессором (например, имена переменных). Распространенной практикой является генерация Представления напрямую по Исходному или Графическому коду путем компиляции. Также для ускорения работы Представление может быть получено из Байт-кода (в том числе, в процессе работы виртуальной машины – с помощью т. н. механизма Just-In-Time [36]). Способ восстановления Машинного кода отсутствует, поскольку оно является конечным в Схеме.

В данном Представлении могут появляться АУ вследствие неверной работы ассемблеров кода или Just-In-Time-утилит; однако вероятность этого крайне мала. Из-за своей полной формализации Представление подходит для автоматического анализа на предмет НУ и АУ; впрочем, по частой причине отсутствия Исходный кодов программ, а также технических сложностей преобразования Представления в предыдущие, Машинный код также вынужденно анализируется и вручную. Примером Представления может быть любая любой бинарный образ для встроенного устройства или UEFI-прошивка [37].

Код данного Представления готов для непосредственного выполнения на Центральном процессоре.

XV. Байт-код (№ 15 на Схеме)

Байт-код является Представлением, аналогичным Машинному, но предназначенному для выполнения на виртуальной машине. Байт-код считается прошедшим синтаксический и семантический анализ и готовым для непосредственного выполнения. Его особенностью является независимость от конечного аппаратного обеспечения, на котором он должен работать, поскольку код не имеет платформенно-зависимых особенностей;

они полностью перенесены на сторону реализации виртуальной машины.

Представление имеет бинарную форму и содержание, аналогичные Ассемблерному и Машинному коду, но с учетом базиса инструкций виртуальной машине.

Отличительной чертой Байт-кода является минимизация в нем уязвимостей из-за особенностей (с позиции ИБ) языков программирования, для которых он генерируется; в частности, в них отсутствует работа с памятью, ограничен доступ к файловой системе, применяется встроенная верификация и автоматическая «сборка мусора» (т. н. Garbage Collection) и т. д. В данном Представлении могут появляться АУ, а поиск их осуществляется автоматическими и трудоемкими ручными способами (по аналогичным с Машинным кодом причинам). Примером Представления может быть высокоуровневый Ассемблерный код для виртуальных машин: CIL (*аббр. от англ.* Common Intermediate Language – общий промежуточный язык) для .Net [38] и JBC (*аббр. от англ.* Java Byte Code) для Java [39].

Код данного Представления готов для непосредственного выполнения на виртуальной машине.

Исходя из Схемы (см. рисунок 1), можно сделать важнейший вывод – за исключением Машинного и Байт-кода, уязвимости обнаруживаются исключительно в тех Представлениях, в которых они были внесены. Так, например, КУ, внесенная при создании Концептуальной модели, обнаруживаемая именно там (ручным способом), а более «дальние» Представления, такие, как Исходный, для этого не подходят. В случае же Машинного и Байт-кода, хотя при создании могут вноситься только АУ, однако по ним также возможен поиск НУ – что обосновывается близостью этих классов уязвимостей и тем, что для генерации Представлений из Дерева абстрактного синтаксиса применяются достаточно «прямолинейных» правил.

7. ЗАКЛЮЧЕНИЕ

Проведенное в работе схематическое моделирование программы с уязвимостями с позиции эволюции ее Представлений, основанное на богатой авторской практике и подкрепленное реальными примерами, позволяет более глубоко взглянуть на динамику процесса создания программ, учитывая появление и обнаружение уязвимостей, что приводит к ряду следующих умозаключений.

Во-первых, проведенный обзор современных исследований в области моделирования ЖЦ показал, что достаточно абстрактные (но без потери сущности процессов) и системные модели предметной области отсутствуют, что свидетельствует о новизне представленного авторского подхода. Ближайшие аналоги «грешат» фрагментарностью, отсутствием причинно-следственных связей и конструктивности выводов. Автором впервые введено четкое и не противоречащее практике описание механизма создания программ с делением на стадии и процессы, в которых непрерывно «зарождаются» и «живут» стратифицированные уязвимости.

Во-вторых, было установлено, что после каждой стадии и процесса создания программы, типы возникающих и обнаруживаемых уязвимостей в большинстве случаев совпадают, что позволяет сделать вывод о необходимости использования для каждого Представления собственных методов поиска, оперирующих соответствующими элементами абстракции. Также возможным решением может быть трансформация Представлений из более поздних в более ранние – т. е. их восстановление, поскольку связанное с этим повышение уровня абстракции также и поднимет структурный уровень обнаруживаемых уязвимостей. Часть из Представлений автоматически преобразуемы в предыдущие (например, Машинный код в Ассемблерный, Байт-код в Исходный), часть – с применением ручного анализа (например, Алгоритмы в Архитектуру), а часть – с применением специализированных программные средств (например, Машинный код в Алгоритмы [40]).

И, в-третьих, очевидная стадийность при создании программ и возможность прямого выполнения ее некоторых конечных Представлений (от бинарного Машинного кода до Сценарного текстового) позволяют предположить следующий вариант описания процессе программного инжиниринга, объединяя и полностью структурируя все существующие Представления. Так, полный путь создания программы для решения поставленной задачи может представлять собой пирамидообразный и конкретизирующий пошаговый процесс преобразования ее Представлений, каждое из которых является в некотором смысле завершенным и может быть выполнено реально или ментально. Например, если упрощенный путь создания программы для вычисления операций над входными значениями (простейший калькулятор) будет проходить через Представления Архитектуры, Алгоритмов и Машинного кода, то каждое из них гипотетически выполнимо в собственной среде с получением собственных (иногда обобщенных) результатов, построенных на собственном базисе, характерном для содержания Представления: для Архитектуры это будет взаимосвязь значений, для Алгоритмов – множество вычисляемых значений, а для Машинного кода – конкретные значения для заданных входных. Такой подход к описанию создания программ позволит на каждом шаге как локализовать появление в нем уязвимостей, так и применять специализированные и максимально эффективные способы их поиска.

Продолжение следует…

Список источников

1. Благодаренко А.В., Разработка метода, алгоритмов и программ для автоматического поиска уязвимостей программного обеспечения в условиях отсутствия исходного кода. Дис. ... канд. техн. наук. Таганрог: Южный федеральный университет, 2011. 140 с.

2. Марков А.С., Фадин А.А. Систематика уязвимостей и дефектов безопасности программных ресурсов // Защита информации. Инсайд. 2013. № 3(51). С. 56–61.

3. Баев Р.В., Скворцов Л.В., Кудряшов Е.А., Бучацкий Р.А., Жуйков Р.А. Предотвращение уязвимостей, возникающих в результате оптимизации кода с неопределенным поведением // Труды Института системного программирования РАН. 2021. Т. 33. № 4. С. 195–210.

4. Буйневич М.В., Израилов К.Е. Антропоморфический подход к описанию взаимодействия уязвимостей в программном коде. Часть 1. Типы взаимодействий // Защита информации. Инсайд. 2019. № 5(89). С. 78–85.

5. Буйневич М.В., Израилов К.Е. Антропоморфический подход к описанию взаимодействия уязвимостей в программном коде. Часть 2. Метрика уязвимостей // Защита информации. Инсайд. 2019. № 6(90). С. 61–65.

6. Максимова Е.А. Методы выявления и идентификации источников деструктивных воздействий инфраструктурного генеза // Электронный сетевой политематический журнал "Научные труды КубГТУ". 2022. № 2. С. 86–99.

7. Максимова Е.А. Аксиоматика инфраструктурного деструктивизма субъекта критической информационной инфраструктуры // Информатизация и связь. 2022. № 1. С. 68–74. DOI:10.34219/2078-8320-2022-13-1-68-74

8. Максимова Е.А., Буйневич М.В. Метод оценки инфраструктурной устойчивости субъектов критической информационной инфраструктуры // Вестник УрФО. Безопасность в информационной сфере. 2022. № 1(43). С. 50–63. DOI:10.14529/secur220107

9. Максимова Е.А. Инфраструктурный деструктивизм субъектов критической информационной инфраструктуры. Москва – Волгоград: Волгоградский государственный университет, 2021. 181 с.

10. Вихрев В.В. О механизме реализации коэволюционной модели жизненного цикла разработки компьютерных программ для обучения // Системы и средства информатики. 2014. Т. 24. № 4. С. 168–185. DOI:10.14357/08696527140411

11. Галимянов А.Ф., Аль-Саффар Н.М.Ф. Жизненный цикл программного продукта с большим количеством пользователей: на примере обучающих программ // VI Международные Махмутовские чтения. Проблемное обучение в современном мире (Казань, Елабуга, 12–14 апреля 2016): сборник статей. Елабуга: Казанский (Приволжский) федеральный университет, Елабужский филиал: 2016. С. 129–133.

12. Слепов В.А. Проектирование и разработка программного продукта "личный блокнот для записи мыслей" // Научное обозрение. Технические науки. 2020. № 4. С. 58–63.

13. Гишлакаев С.У., Минаев О.М. Базовые основы и процессы программной инженерии // XXVII Всероссийская научно-практическая конференция. Цифровизация образования: теоретические и прикладные исследования современной науки (Ростов-на-Дону, Россия, 25.01.2021). Ростов-на-Дону: Южный университет (ИУБиП), ООО "Издательство ВВМ", 2021. Ч. 1. С. 18–22.

14. Iannone E., Guadagni R., Ferrucci F., De Lucia A., Palomba F. The Secret Life of Software Vulnerabilities: A Large-Scale Empirical Study // IEEE Transactions on Software Engineering. 2023, Vol. 49. Iss. 1. PP. 44–63. DOI:10.1109/TSE.2022.3140868

15. Buinevich M., Izrailov K., Vladyko A. The life cycle of vulnerabilities in the representations of software for telecommunication devices // The Proceedings of 18th International Conference on Advanced Communication Technology (ICACT, Pyeongchang, South Korea, 31 January–3 February 2016). IEEE, 2016. PP. 430–435. DOI:10.1109/ICACT.2016.7423420

16. Buinevich M., Izrailov K., Vladyko A. Metric of vulnerability at the base of the life cycle of software representations // The Proceedings of 20th International Conference on Advanced Communication Technology (ICACT, Pyeongchang, South Korea, 11–14 February 2018). IEEE, 2018. PP. 1–8. DOI:10.1109/ICACT.2018.8323940

17. Израилов К.Е. Метод алгоритмизации машинного кода для поиска уязвимостей в телекоммуникационных устройствах. Автореф. дис. ... канд. техн. наук. СПб.: СПбГУТ, 2017. 22 с.

18. Израилов К.Е. Метод алгоритмизации машинного кода для поиска уязвимостей в телекоммуникационных устройствах. Дис. ... канд. техн. Наук. СПб.: СПбГУТ, 2017. 261 с.

19. Буйневич М.В., Израилов К.Е. Аналитическое моделирование работы программного кода с уязвимостями // Вопросы кибербезопасности. 2020. № 3(37). С. 2–12. DOI:10.21681/2311-3456-2020-03-02-12

20. Монастырная В.С., Фролов В.В. Визуальный язык дракон и его применение // Актуальные проблемы авиации и космонавтики. 2016. Т. 2. № 12. С. 78–79.

21. Пароннджанов В.Д. Алгоритмические языки и программирование: ДРАКОН: учебное пособие для среднего профессионального образования. Москва: Издательство Юрайт, 2023. 436 с.

22. Лапшова А.А. Разработка графического описания программного обеспечения с помощью языка UML // Теория и практика современной науки. 2018. № 6(36). С. 894–896.

23. Долидзе А.Н. Обзор специфических функций языка FBD на примере программируемых реле Logo! // Инженерный вестник Дона. 2022. № 11(95). С. 1–10.

24. Pardo M.X.C., Ferreiro G.R. SFC++: A Tool for Developing Distributed Real-Time Control Software // Microprocessors and Microsystems. 1999. Vol. 23. Iss. 2. PP. 75–84. DOI:10.1016/S0141-9331(99)00015-0

25. Ахмерова А.Н. Языки программирования контроллеров. особенности применения языков FBD, LD // Научный аспект. 2019. Т. 3. № 3. С. 340–345.

26. Туренко Д.Л., Кирьянов К.Г. Исследование подходов к идентификации и восстановлению алгоритмов программ // Вестник Нижегородского университета им. Н.И. Лобачевского. Серия: Радиофизика. 2004. № 1. С. 37–46.

27. Nassi I., Shneiderman B. Flowchart techniques for structured programming // SIGPLAN Notices. Vol. 8. Iss. 8. PP. 12–26. DOI:10.1145/953349.953350

28. Басов А.С. Классификация языков программирования и их особенности // Вестник науки. 2020. Т. 2. № 8(29). С. 95–101.
29. Буйневич М.В., Израилов К.Е., Покусов В.В., Тайлаков В.А., Федулина И.Н. Интеллектуальный метод алгоритмизации машинного кода в интересах поиска в нем уязвимостей // Защита информации. Инсайд. 2020. № 5(95). С. 57–63.
30. Кизянов А.О., Глаголев В.А. Концепция универсального языка программирования // Постулат. 2022. № 1(75).
31. Морозов Д.П., Слепнев А.В. Разработка анализатора кода C, C++ на языке Python с использованием Lex, Yacc // 74-я региональная научно-техническая конференция студентов, аспирантов и молодых ученых. Студенческая весна – 2020 (Санкт-Петербург, Россия, 26–27 мая 2020). СПб.: СПбГУТ, 2020. С. 28–32.
32. Буйневич М.В., Израилов К.Е. Основы кибербезопасности: способы анализа программ: учебное пособие. СПб.: Санкт-Петербургский университет ГПС МЧС России, 2022. 92 с.
33. Lee W.I., Lee G. From natural language to Shell Script: A case-based reasoning system for automatic UNIX programming // Expert Systems with Applications. 1995. Vol. 9. Iss. 1. PP. 71–79. DOI:10.1016/0957-4174(94)00050-6
34. Пирогов В. Ассемблер для Windows. Санкт-Петербург: БХВ-Петербург, 2012. 896 с.
35. Капустин Д.А., Швыров В.В., Шулика Т.И. Статический анализ корпуса исходных кодов Python-приложений // Программная инженерия. 2022. Т. 13. № 8. С. 394–403. DOI:10.17587/prin.13.394-403
36. Suganuma T., Ogasawara T., Kawachiya K., Takeuchi M., Ishizaki K., Koseki A., et al. Evolution of a Java just-in-time compiler for IA-32 platforms // IBM Journal of Research and Development. 2004. Vol. 48. Iss. 5.6. PP. 767–795. DOI:10.1147/rd.485.0767
37. Кричанов М.Ю., Чепцов В.Ю. Защищенная UEFI-прошивка для виртуальных машин // Системный администратор. 2021. № 11(228). С. 75–81.
38. Макаров А.В., Скоробогатов С.Ю., Чеповский А.М. Common Intermediate Language и системное программирование в Microsoft.NET: учебное пособие. Москва, Саратов: Интернет-Университет Информационных Технологий (ИНТУИТ), Ай Пи Ар Медиа, 2020. 397 с.
39. Красов А.В., Шариков П.И. Методика защиты байт-кода Java-программы от декомпиляции и хищения исходного кода злоумышленником // Вестник Санкт-Петербургского государственного университета технологии и дизайна. Серия 1: Естественные и технические науки. 2017. № 1. С. 47–0.
40. Buinevich M.V., Izrailov K.E. Method and Utility for Recovering Code Algorithms of Telecommunication Devices for Vulnerability Search // Proceedings of the 16th International Conference on Advanced Communications Technology (ICACT, Pyeongchang, South Korea, 16–19 February 2014). IEEE, 2014. PP. 172–176. DOI:10.1109/ICACT.2014.6778943

References

1. Blagodarenko A.V. *Development of a Method, Algorithms and Programs for Automatic Search for Software Vulnerabilities in the Absence of Source Code*. PhD Thesis. Taganrog: Southern Federal University Publ.; 2011. 140 p. (in Russ.)
2. Markov A.S., Fadin A.A. System of vulnerabilities and security defects of software resources. *Zaŝita informacii. Inside.* 2013;3(51):56–61. (in Russ.)
3. Baev R.V., Skvortsov L.V., Kudriashov E.A., Buchatskii R.A., Zhuikov R.A. Prevention of vulnerabilities arising from optimization of code with undefined behavior. *Trudy ISP RAN/Proc. ISP RAS*. 2021;4(33):195–210 (in Russ.)
4. Buinevich M.V., Izrailov K.E. Anthropomorphic approach to describing the interaction of vulnerabilities in program code. Part 1. Types of interactions. *Zaŝita informacii. Inside*. 2019;5(89):78–85. (in Russ.)
5. Buinevich M.V., Izrailov K.E. Anthropomorphic approach to describing the interaction of vulnerabilities in program code. Part 2. Vulnerability metric. *Zaŝita informacii. Inside*. 2019;6(90):61–65. (in Russ.)
6. Maksimova E.A. Methods for identifying and identifying sources of destructive impacts of infrastructural genesis. *Electronic network polythematic journal "Nauchnye trudy KubGTU*". 2022;2:86–99. (in Russ.)
7. Maksimova E.A. Axiomatics of infrastructure destruction of the subject of critical information infrastructure. *Informatization and communication*. 2022;1:68–74. (in Russ.) DOI:10.34219/2078-8320-2022-13-1-68-74
8. Maksimova E.A., Buynevich M.B. The method of assessing the infrastructural stability of the subjects of critical information infrastructure. *Vestnik UrFO. Security in the Information Sphere.* 2022;1(43):50–63. (in Russ.) DOI:10.14529/secur220107
9. Maksimova E.A. *Infrastructural Destructivism of Critical Information Infrastructure Subjects*. Moscow – Volgograd: Volgograd State University Publ.; 2021. 181 p. (in Russ.)
10. Vikhrev V.V. On the mechanism for implementing the co-evolutionary model of the life cycle of developing computer programs for learning. *Systems and Means of Informatics.* 2014;24(4):168–185. (in Russ.). DOI:10.14357/08696527140411
11. Galimianov A.F., Al-Saffar N.M.F. The Software Product's Life Cycle with Large Numbers of Users: The Case of Training Programs*. VIth International Makhmutov Readings on Problem-Based Learning in the Modern World, 12–14 April 2016, Kazan, Yelabuga, Russia*. Yelabuga: Kazan (Volga region) Federal University Publ.; 2016. p.129–133. (in Russ.)
12. Slepov V.A. Design and development of a software product "personal notepad for writing thoughts". *Scientific Review. Technical science*. 2020;4:58–63. (in Russ.)
13. Gishlakaev S.U., Minaev O.M. Basic foundations and processes of software engineering. *Proceedings of the XXVII All-Russian Scientific and Practical Conference on Digitalization of Education: Theoretical and Applied Research of Modern Science, 25th January 2021, Rostov-on-Don, Russia*. Rostov-on-Don: Southern University Publ.; VVM Publ.; 2021. Part 1. p.18–22. (in Russ.)
14. Iannone E., Guadagni R., Ferrucci F., De Lucia A., Palomba F. The Secret Life of Software Vulnerabilities: A Large-Scale Empirical Study. *IEEE Transactions on Software Engineering*. 2023;49(1):44–63. DOI:10.1109/TSE.2022.3140868
15. Buinevich M., Izrailov K., Vladyko A. The life cycle of vulnerabilities in the representations of software for telecommunication devices. *The Proceedings of 18th International Conference on Advanced Communication Technology, ICACT, 31 January–3 February 2016, Pyeongchang, South Korea*. IEEE; 2016. p.430–435. DOI:10.1109/ICACT.2016.7423420

16. Buinevich M., Izrailov K., Vladyko A. Metric of vulnerability at the base of the life cycle of software representations. *The Proceedings of 20th International Conference on Advanced Communication Technology, ICACT, 11–14 February 2018, Pyeongchang, South Korea*. IEEE; 2018. p.1–8. DOI:10.1109/ICACT.2018.8323940
17. Izrailov K.E. *Machine Code algorithmization method for searching for vulnerabilities in telecommunication devices*. PhD Thesis. St. Petersburg: The Bonch-Bruevich Saint Petersburg State University of Telecommunications Publ.; 2017. 22 p. (in Russ.)
18. Izrailov K.E. *Machine Code algorithmization method for searching for vulnerabilities in telecommunication devices.* PhD Thesis. St. Petersburg: The Bonch-Bruevich Saint Petersburg State University of Telecommunications Publ.; 2017. 22 p. (in Russ.)
19. Buinevich M., Izrailov K. Analytical modeling of the vulnerable program code execution. *Voprosy kiberbezopasnosti*. 2020;3(37):2–12. (in Russ.) DOI:10.21681/2311-3456-2020-03-02-12.
20. Monastyrnaya V.S., Frolov V.V. Visual language dragon and it is application. *Aktual'nye problemy aviacii i kosmonavtiki*. 2016;2(12):78–79. (in Russ.)
21. Parondzhanov V.D. *Algorithmic Languages and Programming: DRAGON*. Moscow: Yurajt Publ.; 2023. 436 p. (in Russ.)
22. Lapshova A.A. Development of a graphic description of software using the UML language. *Teoriya i praktika sovremennoj nauki.* 2018;6(36):894–896. (in Russ.)
23. Dolidze A.N. Overview of specific functions of the FBD language using the example of Logo! *Engineering journal of Don*. 2022;11(95):1–10. (in Russ.)
24. Pardo M.X.C., Ferreiro G.R. SFC++: A Tool for Developing Distributed Real-Time Control Software. *Microprocessors and Microsystems*. 1999;23(2):75–84. DOI:10.1016/S0141-9331(99)00015-0
25. Akhmerova A.N. Controller programming languages. Features of the application of the languages. *Nauchnyj aspekt.* 2019;3(3):340–345. (in Russ.)
26. Turenko D.L. Kirianov K.G. Research of approaches to identification and recovery of program algorithms. *Vestnik of Lobachevsky University of Nizhni Novgorod*. 2004;(1):37–46. (in Russ)
27. Nassi I., Shneiderman B. Flowchart techniques for structured programming. *SIGPLAN Notices*. 8(8):12–26. DOI:10.1145/ 953349.953350
28. Basov A.S. Classification of programming languages and their features. *Vestnik nauki.* 2020;2(8):95–101. (in Russ.)
29. Buinevich M.V., Izrailov K.E., Pokusov V.V., Tailakov V.A., Fedulina I.N. An intelligent method of machine code algorithmization in the interests of finding vulnerabilities in it. *Zašita informacii. Inside.* 2020;5(95):57–63. (in Russ.)
30. Kizianov A.O., Glagolev V.A. Concept of a universal programming language. *Postulat*. 2022;1(75). (in Russ.)
31. Morozov D.P., Slepnev A.V. Development of C, C++ code analyzer in Python using Lex, Yacc. *Proceedings of the 74th Regional Scientific and Technical Conference of Students, Graduate Students and Young Scientists "Student Spring – 2020", 26–27 May 2020, St. Petersburg, Russia.* St. Petersburg: The Bonch-Bruevich Saint Petersburg State University of Telecommunications Publ.; 2020. p.28–32. (in Russ.)
32. Buinevich M.V., Izrailov K.E. *Cybersecurity Fundamentals: Ways to Analyze Programs.* St. Petersburg: St. Petersburg University of the State Fire Service of the Ministry of Emergency Situations of Russia Publ.; 2022. 92 p. (in Russ.)
33. Lee W.I., Lee G. From natural language to Shell Script: A case-based reasoning system for automatic UNIX programming. *Expert Systems with Applications*. 1995;9(1):71–79. DOI:10.1016/0957-4174(94)00050-6
34. Pirogov V. *Assembler for Windows*. BHV-Petersburg Publ.; 2012. 896 p. (in Russ.)
35. Kapustin D.A., Shvyrov V.V., Shulika T.I. Static analysis of the source code of python applications. *Software Engineering*. 2022;13(8):394–403. (in Russ.) DOI:10.17587/prin.13.394-403
36. Suganuma T., Ogasawara T., Kawachiya K., Takeuchi M., Ishizaki K., Koseki A., et al. Evolution of a Java just-in-time compiler for IA-32 platforms. *IBM Journal of Research and Development.* 2004;48(5.6):767–795. DOI:10.1147/rd.485.0767
37. Krichanov M.Y., Cheptsov V.Y. Secure UEFI firmware for virtual machines. *Sistemnyj administrator*. 2021;11(228): 75–81. (in Russ.)
38. Makarov A.V., Skorobogatov S.Y., Chepovskii A.M. Common Intermediate Language and system programming in Microsoft.NET. Moscow, Saratov: Internet University of Information Technologies Publ.; Ai Pi Ar Media Publ.; 2020. 397 p. (in Russ.)
39. Krasov A.V., Sharikov P.I. Methods of protection byte code java-programs from decompilation and theft of source code by an attacker. *Vestnik of St. Petersburg State University of Technology and Design. Series 1: Natural and technical Sciences.* 2017;(1):47–50. (in Russ.)
40. Buinevich M.V., Izrailov K.E. Method and Utility for Recovering Code Algorithms of Telecommunication Devices for Vulnerability Search. *Proceedings of the 16th International Conference on Advanced Communications Technology, ICACT, 16–19 February 2014, Pyeongchang, South Korea.* IEEE; 2014. p.172–176. DOI:10.1109/ICACT.2014.6778943

Статья поступила в редакцию 20.02.2023; одобрена после рецензирования 27.02.2023; принята к публикации 28.02.2023.
The article was submitted 20.02.2023; approved after reviewing 27.02.2023; accepted for publication 28.02.2023..

Информация об авторе:

ИЗРАИЛОВ Константин Евгеньевич — кандидат технических наук, старший научный сотрудник Санкт-Петербургского Федерального исследовательского центра Российской академии наук, https://orcid.org/0000-0002-9412-5693