Научная статья
УДК 004.42
DOI:10.31854/1813-324X-2023-9-2-95-111

Моделирование программы с уязвимостями с позиции эволюции ее представлений. Часть 2. Аналитическая модель и эксперимент

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. Во второй части цикла статей предлагается обобщенная аналитическая модель жизненного цикла программы на базе ее представлений, учитывающая способы прямого и обратного преобразования последних. Также в модели отражается возникновение и обнаружения уязвимостей и их классификация. Из нее синтезируется частная модель, отражающая текущее состояние эволюции программных представлений, и исходя из которой был выведен ряд основополагающих утверждений, записанных в аналитическом виде. Для основания работоспособности моделей в части отражения уязвимостей проводятся два следующих эксперимента: ретроспективно-фактологический, сопоставляющий реально существующие уязвимости с частной моделью; и практический, демонстрирующий эволюцию уязвимости в представлениях в процессе эволюции простейшей программы. В результате второго эксперимента наглядно показан рост охвата уязвимостью представлений в процессе разработки.*

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

Ссылка для цитирования: Израилов К.Е. Моделирование программы с уязвимостями с позиции эволюции ее представлений. Часть 2. Аналитическая модель и эксперимент // Труды учебных заведений связи. 2023. Т. 9. № 2. С. 95–111. DOI:10.31854/1813-324X-2023-9-2-95-111

Modeling a Program with Vulnerabilities in the Terms of the Its Representations Evolution. Part 2. Analytical Model and Experiment

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. In the second part of the articles series, a program life cycle generalized analytical model of the based on its representations is proposed, taking into account the direct and reverse transformation methods. Also, the model reflects the occurrence and detection of vulnerabilities and their classification. A particularly model is synthesized from it, reflecting the current state of the program representations evolution, and on the basis of which a number of fundamental statements were derived, written in an analytical form. To base the performance of models in*

reflecting vulnerabilities terms, the following two experiments are carried out: retrospective-factual, comparing real-life vulnerabilities with a particularly model; and practical demonstration the evolution of vulnerability in representations in the process of the simplest program evolution. As a result of the second experiment, the increase in coverage by the vulnerability of representations in the development process is clearly shown.

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

For citation: Izrailov K. Modeling a Program with Vulnerabilities in the Terms of Its Representations Evolution. Part 2. Analytical Model and Experiment. *Proc. of Telecom. Universities.* 2023;9(2):95–111. (in Russ.) DOI:10.31854/1813-324X-2023-9-2-95-111

1. ВВЕДЕНИЕ

Поиск уязвимостей в программном обеспечении является актуальнейшей задачей в сфере информационной безопасности (далее – ИБ). Наличие уязвимостей в зависимости от области функционирования программ может приводить к угрозам различной степени тяжести, вплоть до человеческих жертв (что особенно актуально для киберфизических устройств [1–3]). Однако, несмотря на достаточно долгую историю решения данной задачи, многие успехи в обнаружении и нейтрализации уязвимостей нивелируются новыми способами их внедрения злоумышленниками. Причина этого, в том числе, может заключаться в слабой формализации не только самого понятия уязвимости, но и всего жизненного цикла программы – от появления самой ее задумки до получения непосредственно выполняемого образца.

Часть важнейших и используемых далее результатов была получена в предыдущей части данного цикла статей [4]. Ранее автором были обоснованы и введены *представления* (далее – Представление), через которые проходит любая программа, а именно, следующие (с их краткой сутью):

1) Идея – изначальный замысел программы;

2) Концептуальная модель – основные понятия, их взаимосвязи и особенности программы;

3) Архитектура – структура программы с делением на физические и логические модули;

4) Двухмерная структурная схема – совмещение Представления алгоритмов в графическом и текстовом виде (например, язык ДРАКОН [5, 6]);

5) Функциональная диаграмма – аналог Двухмерной структурной схемы, но с некоторыми особенностями (например, FBD [7], SFC [8] или LD [9]);

6) Блок-схема – классическое блочное Представление алгоритма, как правило, в соответствии с ГОСТ 19.701-90 Единая система программной документации. Схемы алгоритмов, программ, данных и систем. Обозначения условные и правила выполнения;

7) Структурограмма – аналог Блок-схем, но для программ согласно структурной парадигме программирования; как следствие, без операций безусловного перехода (например, диаграммы Насси – Шнейдермана [10]);

8) Псевдокод – запись алгоритма в специальной текстовой нотации (например, семейство языков Algol [11]);

9) Классический исходный код – код на классических языках программирования (Pascal, Java и C/C++/C# и т. п.);

10) Метакод генерации – промежуточный код для генерации Классического исходного (например, формальные грамматики для генератора синтаксического анализатора Yacc/Bison [12]);

11) Сценарный код – развитие Классического кода в сторону повышения абстракции, упрощения разработки и укрупнения используемых элементов (сценарий командной строки Shell Script [13]);

12) Ассемблерный код – низкоуровневое Представление кода, отражающее специфику среды выполнения (например, согласно синтаксису для утилиты-ассемблера TASM [14]);

13) Дерево абстрактного синтаксиса – промежуточное Представление между человеко- и машиноориентированным кодом (так, Представление используется инструментом Bandit [15] для поиска уязвимостей коде Python-программ);

14) Машинный код – код программы, готовый для выполнения на Центральном процессоре (например, бинарный образ для UEFI-прошивка персонального или серверного компьютера [16]);

15) Байт-код – аналогичный Машинному коду, но предназначенный для выполнения на виртуальной машине (например, CIL (*аббр. от англ.* Common Intermediate Language – общий промежуточный язык) для .Net [17] и JBC (*аббр. от англ.* Java Byte Code) для Java [18]).

Для определения Представлений использовались понятия из категориальный пары – *форма* и *содержания* [19]. Первый элемент пары отвечает за внешнее отражение Представления; второй же соответствует внутреннему функционалу программы в данной форме (отражая тем самым первоначальную Идею программы); при этом, содержание само состоит из двух компонент – своей функциональной логики и базиса, на котором эта логика построена.

На базе Представлений была предложена схема жизненного цикла программы (далее – Схема), приведенная на рисунке 1.

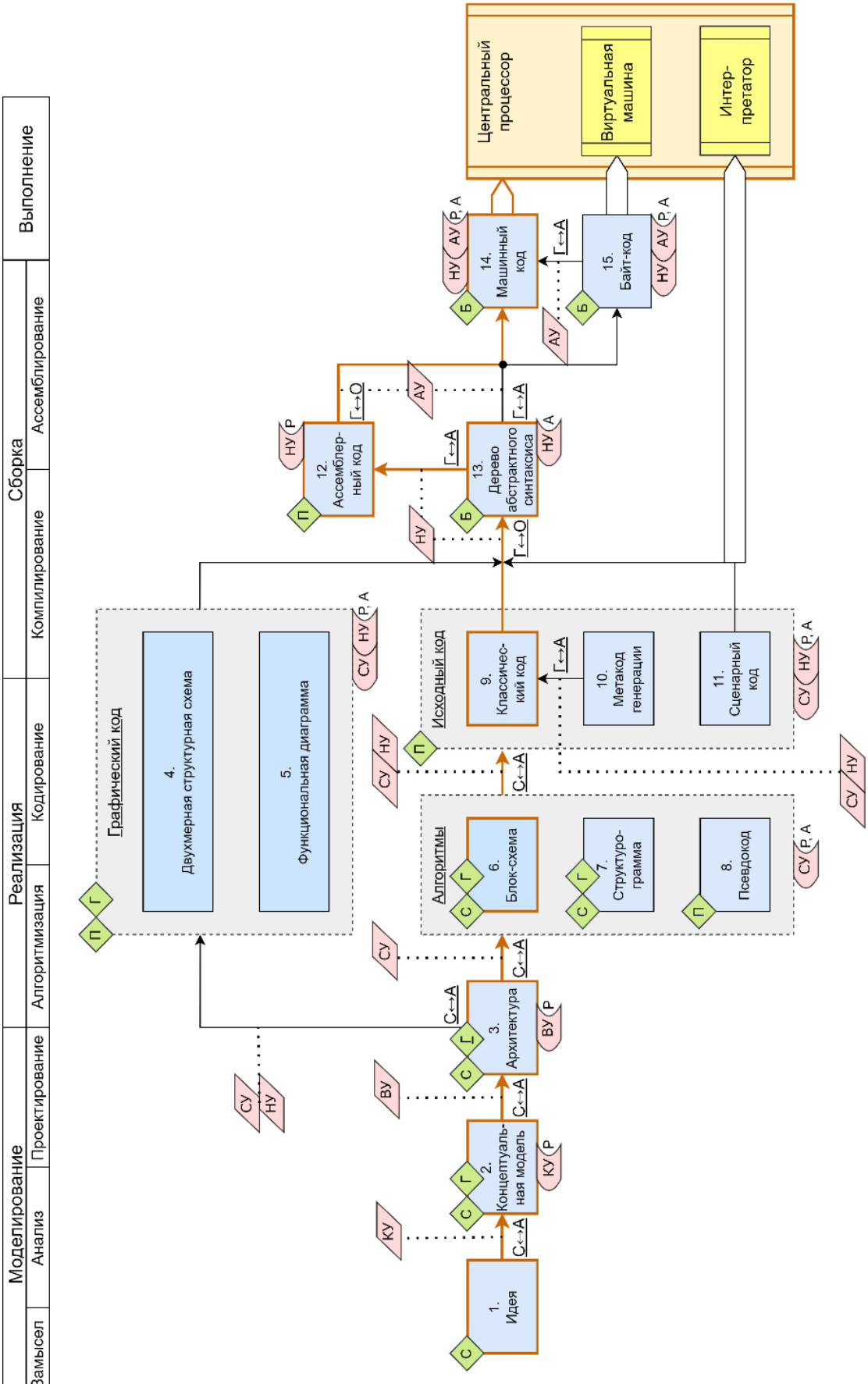

Рис. 1. Схема жизненного цикла Представлений программы с уязвимостями

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

На Схеме используются следующие обозначения: таблицы сверху – стадии разработки программы и их состав; прямоугольник с синим фоном – само Представление; ромб с зеленым фоном – его форма (по первой букве): Словесная, Графическая, Программно-языковая и Бинарная; стрелки – способ получения Представления из предыдущего; надпись под стрелкой в формате «$X \leftrightarrow Y$» – тип прямого и обратного преобразования (по первой букве): Анализ, Синтез, Генерация, Обратная генерация; параллелограмм с красным фоном, соединенный пунктирной линией со способом получения – класс возникающей уязвимости согласно структурному уровню (по первой букве): Концептуальные, Высокоуровневые, Среднеуровневые, Низкоуровневые, Атомарные; прямоугольник с розовым фоном и гнутыми боками – класс уязвимости, обнаруживаемый в Представлении; буква внутри вогнутой части прямоугольника – тип способа обнаружения (по первой букве): Ручной, Автоматический; прямоугольник с боковыми линиями и желтым фоном – механизм выполнения конечного Представления программы.

Как результат, удалось ввести более формализованное и точное (с точки зрения автора) определение понятия «уязвимость», как *различие содержания программы в ее конечном и изначальном Представлениях*.

Классификация уязвимостей детализирована: по структурному уровню – согласно уровню абстракции, на котором внесена уязвимость (что отображается на Схеме); по изменению содержания – потеря, внесение и модификация функционала; по воздействию на внутренние информационные потоки (по аналогии с триадой нарушения конфиденциальности, целостности и доступности): появление, изменение и потеря потока.

А поскольку визуально Схема обладает признаками формализации (строго выделенные элементы, логика их связей, перечислимый набор свойств и т. п.), то целесообразно ее записать в виде соответствующей аналитической модели, а именно – жизненного цикла программы (далее – Модель); это позволит применять соответствующие математические аппараты, что будет иметь как теоретическую значимость – например, для доказательства решения различных задач по преобразованию Представлений, так и практическую значимость – для реализации программных средств управления комплексным методом (де)эволюции Представлений. Учет же в аналитической Модели классификации уязвимостей позволит ее применить для сферы ИБ программной инженерии.

2. ОБОБЩЕННАЯ АНАЛИТИЧЕСКАЯ МОДЕЛЬ

В интересах строгой записи аналитической Модели введем следующие обозначения (строчными буквами) упоминаемых ранее основополагающих объектов предметной области:

– p (*от англ.* Presentation) – Представление;
– f (*от англ.* Form) – форма;
– c (*от англ.* Content) – содержание;
– v (*от англ.* Vulnerability) – уязвимость.

Подчеркнем, что все объекты предметной области так или иначе связаны с сущностями информационного мира.

Следуя той же логике, введем обозначения (прописными буквами) операций над объектами, детально описанные в предыдущей части цикла статей:

– T (*от англ.* Transformation) – преобразование Представления некоторым способом;
– $\overline{T}$ (верхняя черта соответствует отрицанию) – обратное преобразование Представления некоторым способом;
– I (*от англ.* Infection) – внесение уязвимости некоторым способом;
– D (*от англ.* Detection) – обнаружение уязвимости некоторым способом;
– E – пустое действие (т. е. отсутствие операции, как таковой).

В результате действий получаются новые объекты (в том числе, тождественные исходным).

Также будем использовать следующие достаточно распространенные (и, что важно, интуитивно понятные и имеющие лаконичную запись) обозначения математического аппарата:

1) $\circ$ – действие над объектом (аналог функции от одного операнда, что позволяет записывать последовательность операций в хорошо читаемом виде);

2) $\langle x|y\rangle$ – совокупность объектов или операций x и y (используется для разделения на форм-содержательные аспекты более сложной сущности);

3) правый верхний индекс – обозначение номера Представления (индекс будет отсутствовать тогда, когда отнесение объекта или процесса к конкретному Представлению не существенно); индексы тождественны номерам Представлений на Схеме; исключением является формат индекса «$x \to y$», который подчеркивает, что идет преобразование Представления с индексом x к индексу y.

4) используются следующие наименования индексов: k – для Представлений; i – для перечисления объектов произвольного рода; x и y – произвольные для дополнительных целей; a, n и m – для примеров;

5) $\equiv$ – тождественность выражений (т. е. верность при любых значениях их переменных);

6) $\setminus$ – отличие между объектами (аналог вычитания множеств);

7) $x \to y$ – некоторый алгоритм по преобразования объекта x к объекту y;

8) $\triangleright$ – специальное преобразование, которое согласованно изменяет логику и базис содержания (т. е. после применения изменяется не все содержание, а его компоненты);

9) $\emptyset$ – несуществующий объект (может относиться к Представлению или уязвимости);

10) $\mathbb{C}$ – получение класса объекта (может состоять из подклассов, обозначаемых в виде подписей в нижней правой части символа).

11) $\cdot$ – оператор составления нового объекта из подобъектов (используется для повышения читаемость записи);

12) $\hat{x}$ («шапка» над символом) – пометка того, что объект содержит уязвимость (используется контекстно для улучшения восприятия записи).

Далее для упрощения будем Представление с индексом k называть просто – Представление k.

Исходя из введенных обозначений и используя сделанные ранее утверждения, а также продуцируя новые, Схема в аналитическом виде может быть записана следующим образом.

Утверждение 1. Представление является совокупностью формы и содержания:

$$p^k \equiv \langle f|c\rangle^k \equiv \langle f^k|c^k\rangle.$$

Иная, более интуитивная, совокупность формы и содержания следующая:

$$\langle f|c\rangle \equiv \boxed{f\boxed{c}} \equiv c \underline{by} f,$$

где $\boxed{\quad}$ и $\underline{by}$ – «представимость» (содержания с в форме f); впрочем, данная форма далее не применяется, а указана лишь для расширения визуальной части предлагаемого математического аппарата.

Введем также класс формы, а именно следующий:

$$\mathbb{C}_{form}, form \in \{VF, GF, PF, BF\},$$

где $form$ соответствует одной из введенных ранее форм: словесной (VF), графической (GF), программной (PF) и бинарной (BF), где первая буква является *сокр. от англ.* названия (Verbal, Graphic, Program, Binary), а вторая – *сокр. от англ.* Form.

Утверждение 2. Исходным Представлением любой программы является идея:

$$p^0 \equiv \langle f|c\rangle^0 \equiv \langle f^0|c^0\rangle,$$

где 0 – индекс Представление Идеи.

Утверждение 3. Форма и содержания отражают программу в некотором одном Представлении и не могут составлять совокупность, находясь в разных Представлениях:

$$(\forall x, y: x \neq y) \Rightarrow \langle f^x|c^y\rangle \in \emptyset,$$

где $\in \emptyset$ означает, что объект (т. е. Представление $\langle f^x|c^y\rangle$) не существует.

Другими словами, содержание Представления может сосуществовать только с формой этого же Представления. Для обоснования этого можно привести следующий пример – логика работы Машинного кода не может быть отражена в форме Исходного кода, поскольку первый оперирует инструкциями процессора (ориентированными на автомат), а второй – конструкциями языка программирования (ориентированными на человека).

Утверждение 4. Жизненный цикл программы состоит из преобразования Представлений:

$$p^{k+1} = T^{k\to k+1} \circ p^k,$$

где $k+1$ – индекс нового Представления, получаемого из Представления k. Индекс $k \to k+1$ означает, что способ преобразования предназначен для оперирования с формой f^k и содержанием c^k с получением аналогичных для индекса $k+1$.

Саму операцию преобразования (как и любую другую, применяемую к Представлению) можно разбить на 2 части, алгоритмы которых работают с формой и содержанием:

$$\begin{cases} T^{k\to k+1} \equiv \langle (T^{k\to k+1})_f | (T^{k\to k+1})_c \rangle \\ (T^{k\to k+1})_f \equiv f^k \to f^{k+1} \\ (T^{k\to k+1})_c \equiv c^k \triangleright c^{k+1} \end{cases},$$

где $(T^{k\to k+1})_f$ – изменение формы Представления k к $k+1$; $(T^{k\to k+1})_c$ – перевод логики содержания на базисе Представления k к логике Представления на базисе $k+1$; $\to$ – алгоритм изменения формы; $\triangleright$ – алгоритм перевода логики на другой базис.

Подчеркнем, что поскольку преобразование происходит без внесения уязвимостей, то само содержание не меняется, а осуществляется лишь согласованное изменение его логики и базиса; например, при кодировании Алгоритмов с помощью Исходного кода меняется не само функционирование, а логика и объекты, на которых она строится – например, с элементов блок-схем к конструкциям языка программирования.

Введем классификацию преобразований согласно веденной ранее:

$$\mathbb{C}_{transform}, transform \in \{ST, GT\},$$

где $transform$ соответствует одному из введенных ранее типов прямого преобразования: синтез (ST) и генерация (GT); первая буква названия является *сокр. от англ.* названия (Synthes, Generation), а вторая – *сокр. от англ.* Transform, *перев. на рус.* Трансформировать.

Утверждение 5. Уязвимость Представления есть отличие между содержаниями Представлений Идеи и текущего (независимо от их формы):

$$v \equiv c^0 \setminus c^k,$$

где k – индекс текущего Представления.

Утверждение 6. Отсутствие уязвимости соответствует тождественности Представлений Идеи и текущего:

$$c^0 \equiv c^k \Leftrightarrow c^0 \setminus c^k = v \in \emptyset,$$

где $v \in \emptyset$ означает, что какая-либо уязвимость не существует.

Утверждение 7. Классификация уязвимости возможна по ее структурному уровню, изменению содержаний Представлений и воздействию на информационные потоки:

$$\begin{cases} \mathbb{C}_v \equiv (\mathbb{C}_v)_{level} \cdot (\mathbb{C}_v)_{difference} \cdot (\mathbb{C}_v)_{infoflow} \\ level \in \{CL, HL, ML, LL, AL\} \\ difference \in \{-, +, \triangle\} \\ infoflow \in \{-, +, \triangle\} \end{cases},$$

где $\mathbb{C}_v$ – класс уязвимости (составной); $(\mathbb{C}_v)_{level}$ – подклассы согласно структурному уровню $level$: КУ (CL), ВУ (HL), СУ (ML), НУ (LL), АУ (AL), где первая буква названия является *сокр. от англ.* аналогов названий уязвимостей (Conceptual, High, Middle, Low, Atomic), а вторая – сокращение переведенного *на англ. яз.* слова «уровень» (Level); $(\mathbb{C}_v)_{difference}$ – подклассы согласно изменению в содержании $difference$: потеря функционала ($-$), внесение функционала ($+$), модификация функционала ($\triangle$); $(\mathbb{C}_v)_{infoflow}$ – подклассы согласно изменению информационных потоков $infoflow$ (*сокр. от* Information Flow): потеря информационного потока ($-$), возникновение информационного потока ($+$), изменение информационного потока ($\triangle$).

Так, если обнаружена уязвимость в Алгоритме системы аутентификации (что соответствует СУ), реализованная с помощью добавление пароля по умолчанию (соответствует внесению функционала) и позволяющая проходить аутентификацию несанкционированным для этого объектам-пользователям (соответствует появлению информационного потока), то ее составной класс будет записан следующим образом:

$$\mathbb{C}_v = \mathbb{C}_{ML++}.$$

Утверждение 8. Уязвимость может быть внесена операцией, которая меняет содержания в некотором Представлении:

$$\hat{p} \equiv \widehat{\langle f|c\rangle} \equiv \langle f|\hat{c}\rangle = I \circ p,$$

где $\hat{p}$ или тождественное ему $\widehat{\langle f|c\rangle}$ – Представление с внесенной уязвимостью (т. е. уже небезопасное); $\hat{c}$ – содержание Представления с этой же уязвимостью (форма, очевидно, сама по себе из-за внесения уязвимости не меняется, поскольку также используется для отображения уязвимости).

Утверждение 9. Операция внесения уязвимости может быть записана, как способ изменения содержания, при котором форма Представления является лишь связывающим звеном (программы и способа) и на уязвимость никак не влияет:

$$\begin{cases} \mathrm{I}^k \equiv \langle (E)_f | (I^k)_c \rangle \\ (I^k)_c \equiv c^k \to \hat{c}^k \end{cases},$$

где $(E)_f$ – отсутствие изменений в конкретной форме Представления; $(I^k)_c$ – изменение содержания Представления k при внесении уязвимостей; $\hat{c}^k$ – содержание с внесенной уязвимостью; $\to$ – некий алгоритм изменения содержания (также отразится на подклассе уязвимости $(\mathbb{C}_v)_{difference}$).

Утверждение 10. Уязвимость может быть найдена путем применения операции обнаружения к Представлению:

$$v = D \circ p$$

Естественно, не каждое Представление подходит для обнаружения всех классов уязвимостей (например, «просчеты» в концептуальной модели практически невозможно найти по Машинному коду), о чем будет дано следующее утверждение. При этом, далеко не все уязвимости могут быть найдены автоматически и требуют применения ручного труда эксперта; для учета этого можно ввести классификацию способов поиска:

$$\mathbb{C}_{detect}, detect \in \{HD, AD\},$$

где $detect$ соответствует ручной (HD) или автоматической (AD) форме, где первая буква названия является *сокр. от англ.* названия (Hand, Auto), а вторая – *сокр. от англ.* Detect, *перев. на рус.* Обнаружить.

Утверждение 11. Способы обнаружения уязвимостей взаимодействуют с содержанием посредством формы, и, следовательно, могут применяться (а точнее иметь не 0-й эффект) только на соответствующем Представлении:

$$(\forall x, y: x \neq y) \Rightarrow D^x \circ p^y \in \emptyset,$$

где $\in \emptyset$ означает, что по данному Представлению данным способом не будет найдено ни одной уязвимости.

А поскольку (согласно Утверждению 5) суть операции обнаружения состоит в определении отличий между содержаниями двух Представлений, и при этом (согласно текущему Утверждению) обнаружение действует только для того Представления, для которого был создан его способ, то логично возникает следующее (производное) утверждение.

Утверждение 12. Операция обнаружения уязвимости может быть записана, как способ нахождения отличия содержания Представлений Идеи от содержания текущего Представления, при котором форма Представления является лишь связывающим звеном (программы и способа) и на отличие не влияет:

$$\begin{cases} \mathrm{D}^k \equiv \langle (E)_f | (D^k)_c \rangle \\ (D^k)_c \equiv c^0 \setminus c^k \end{cases},$$

где $(E)_f$ – не учет способом обнаружения конкретной формы Представления; $(D^k)_c$ – применение способа обнаружения к Представлению только в части содержания.

Утверждение 13. Восстановление Предыдущих Представлений из текущего согласно жизненному циклу программы состоит из обратных преобразований Представлений:

$$p^{k-1} = \bar{T}^{k \to k-1} \circ p^k,$$

где $k-1$ – индекс предыдущего Представления, получаемого из текущего Представления k; индекс $k \to k-1$ означает, что способ обратного преобразования предназначен для оперирования с формой f^k и содержанием c^k, получая аналогичные для индекса $k-1$.

По аналогии с прямым преобразованием (согласно Утверждению 4) операция обратного преобразования разбивается на 2 части, алгоритмы которых работают с формой и содержанием:

$$\begin{cases} \bar{\mathrm{T}}^{k \to k-1} \equiv \langle (T^{k \to k-1})_f | (T^{k \to k-1})_c \rangle \\ (T^{k \to k-1})_f \equiv f^k \to f^{k-1} \\ (T^{k \to k-1})_c \equiv c^k \triangleright c^{k-1} \end{cases},$$

где $(T^{k \to k-1})_f$ – изменение формы Представления k к Представлению $k-1$; $(T^{k \to k-1})_c$ – перевод логики содержания на базисе Представления k к логике на базисе Представления $k-1$.

Аналогично прямому преобразованию, в процессе операции уязвимости не появляются.

Введем классификацию преобразований согласно указанной ранее:

$$\mathbb{C}_{\overline{transform}}, \overline{transform} \in \{AT, RT\},$$

где $\overline{transform}$ соответствует одному из введенных ранее типов обратного преобразования: анализ (AT) и обратная генерация (RT); первая буква названия является *сокр. от англ.* Analysis, Reverse generation, а вторая – *сокр. от англ.* Transform.

Данный процесс, называемый *реверс-инжинирингом*, как раз и призван частично решить основную проблему поиска уязвимостей, внесенных на более ранних Представлениях, чем исследуемое.

Утверждение 14. Весь жизненный цикл программы без внесения уязвимостей может быть представлен, как множество цепочек преобразований Представлений:

$$\langle f^n | c^n \rangle = p^n = T^{n-1 \to n} \circ p^{n-1} = \\ = T^{n-1 \to n} \circ \ldots \circ T^{0 \to 1} \circ \langle f^0 | c^0 \rangle.$$

Проверим корректность действия операции обнаружения уязвимостей для такого случая безопасной (с позиции ИБ) программной инженерии. Поскольку в процессе создания программы уязвимости не вносились, то (согласно Утверждению 6) $c^0 \equiv c^n$. Тогда результат попытки обнаружения уязвимостей в конечном Представлении (согласно Утверждениям 10 и 12) будет следующим:

$$(c^0 \equiv c^n) \Longrightarrow v^n = D^k \circ \langle f^n | c^n \rangle = \langle f^n | c^0 \setminus c^n \rangle = \\ = \langle f^n | \emptyset \rangle \in \emptyset, \text{ т. е. } v \in \emptyset.$$

Таким образом, формальное применение операции обнаружения уязвимостей для безопасного Представления программы не позволило получить какую-либо уязвимость, что подтверждает тождественность сделанных утверждений и их аналитических записей.

Расширим аналитическую запись цепочки жизненного цикла программы для случая наличия уязвимостей.

Утверждение 15. Весь жизненный цикл программы с внесением уязвимостей (для общности, на каждом преобразовании) может быть представлен, как множество цепочек преобразований Представлений:

$$\langle \widehat{f^n | c^n} \rangle = I^n \circ \hat{p}^n = I^n \circ T^{n-1 \to n} \circ \hat{p}^{n-1} = \\ = I^n \circ T^{n-1 \to n} \circ I^{n-1} \circ \ldots \circ I^1 \circ T^{0 \to 1} \circ \langle f^0 | c^0 \rangle.$$

Применим описанные элементы аналитической Модели к реальным ситуациям, соответствующим следующим классическим задачам области ИБ программ.

Задача 1. Обнаружить уязвимость, которая внесена в Представлении a, если для анализа есть более позднее Представление m.

В качестве примера можно взять уже упоминаемую ранее уязвимость – пароль по умолчанию в Алгоритме программы (индекс «a» как раз и соответствует *сокр. от англ.* Algorithm, *перев. на рус.* алгоритм) внесенный туда с помощью дополнительной проверки, когда в наличии есть Машинный код этой программы (индекс «m» соответствует *сокр. от англ.* Machine, *перев. на рус.* машинный).

Запись такого Представления m с уязвимостью (внесенной операцией I^a) имеет следующий вид:

$$a < m: \langle \widehat{f^m | c^m} \rangle = T^{m-1 \to m} \ldots T^{a \to a+1} \circ I^a \circ T^{a-1 \to a} \circ \ldots \\ \circ T^{0 \to 1} \circ \langle f^0 | c^0 \rangle.$$

Попытка обнаружения указанной уязвимости с учетом того, что такая операция действует только на содержание (согласно Утверждению 12), которое меняется только при внесении уязвимостей (согласно Утверждению 5), а также следуя способу преобразования Представлений в рамках жизненного цикла (согласно Утверждению 4), будет иметь следующую запись:

$$v^a = D^a \circ \langle \widehat{f^m|c^m} \rangle =$$
$$= \left\langle (E)^a{}_f \middle| c^0 \setminus (c^0 \rhd \cdots \rhd c^a \to \hat{c}^a \rhd \cdots \rhd \hat{c}^m) \right\rangle,$$

где факт внесения уязвимости операцией I^a записан, как изменения содержания в Представлении a (согласно Утверждению 9): $c^a \to \hat{c}^a$.

Очевидно, что содержание меняется только в Представлении a и, следовательно, операция D (которая как раз и находит отличие между содержаниями) может применяться (согласно Утверждению 11) только для этого Представления:

$$v^a = D^a \circ \langle \widehat{f^m|c^m} \rangle = \left\langle (E)^a{}_f \middle| D^a \circ \hat{c}^m \right\rangle \in \emptyset,$$

где последняя часть (согласно Утверждению 11) постулирует о невозможности получения уязвимости таким образом – способом, работающем на Представлении a (в примере – для Алгоритмов), но применяемым для m (в примере – для Машинного кода).

Единственно возможным способом поиска уязвимости v^a является применение операции $(D^a)_c$ к содержанию $\hat{c}^a$, для чего в аналитической Модели существует операция обратного преобразования, применение которой для Представления m (согласно Утверждению 13) будет иметь следующий вид:

$$\hat{p}^a = \bar{T}^{a+1 \to a} \circ \ldots \circ \bar{T}^{m \to m-1} \circ \hat{p}^m.$$

В этом случае обнаружение уязвимости будет следующим:

$$v^a = D^a \circ \hat{p}^a =$$
$$\left\langle (E)^a{}_f \middle| c^0 \setminus (c^0 \rhd \cdots \rhd c^a \to \hat{c}^a \rhd \cdots \rhd \hat{c}^m \rhd \cdots \rhd \hat{c}^a) \right\rangle \notin$$
$$\notin \emptyset,$$

где $\notin \emptyset$ означает, что по данному Представлению данным способом возможно обнаружение уязвимостей.

Соответственно, класс уязвимости для примера будет следующим (согласно Утверждению 7):

$$\mathbb{C}_{v^a} = \mathbb{C}_{ML++}.$$

Описанный в задаче подход к обнаружению уязвимостей дословно звучит, как преобразование программы из Представления m к более раннему Представлению a для непосредственного поиска уязвимостей именно в нем. Такой подход является хорошо известным под общим названием «Реверс-инжиниринг программ в интересах поиска в них уязвимостей».

Далее кратко рассмотрим более сложную и реалистичную задачу.

Задача 2. Обнаружить несколько уязвимостей, которые внесены в Представления x и y, если для анализа есть более позднее Представление m.

Пример можно считать расширенной версией аналогичного для рассмотренной Задачи 1, когда имея в наличие Машинный код (Представление m), необходимо найти уязвимость Архитектуры (Представление x) с потерей функционала – отсутствие механизма шифрования (что позволяет получать несанкционированный доступ к базам данным), и Алгоритма (Представление y) с внесением функционала – вход с помощью пароля по умолчанию (что позволяет получать несанкционированный вход в систему).

Запись такого Представления имеет следующий вид:

$$x < y < m:$$
$$\hat{p}^m = \langle \widehat{f^m|c^m} \rangle = T^{m-1 \to m} \ldots T^{y \to y+1} \circ I^y \circ T^{y-1 \to y} \circ \ldots$$
$$\circ \ldots T^{x \to x+1} \circ I^x \circ T^{x-1 \to x} \circ \ldots \circ T^{0 \to 1} \circ \langle f^0|c^0 \rangle,$$

а Представления после раздельного внесения уязвимостей будут следующими:

$$\begin{cases} \hat{p}^x = I^x \circ p^x \\ \hat{p}^y = I^y \circ p^y \end{cases}.$$

Аналогичным с решением Задачи 1 образом обнаружение уязвимостей необходимо будет произвести на каждом из Представлений x и y, поскольку только на них будет работать соответствующий способ. Для этого, соответственно, потребуется обратное преобразование Представления m (в примере, Машинный код) сначала к y (в примере, к Алгоритмам), а затем к x (в примере, к Архитектуре):

$$\begin{cases} v^y = D^y \circ \hat{p}^y = \bar{T}^{y+1 \to y} \circ \ldots \circ \bar{T}^{m \to m-1} \circ \hat{p}^m \\ v^x = D^x \circ \hat{p}^x = \bar{T}^{x+1 \to x} \circ \ldots \circ \bar{T}^{y \to y-1} \circ \hat{p}^y \end{cases}.$$

Соответственно, классы уязвимости для примера будут следующими:

$$\begin{cases} \mathbb{C}_{v^x} = \mathbb{C}_{HL-+} \\ \mathbb{C}_{v^y} = \mathbb{C}_{ML++} \end{cases}.$$

Необходимо уточнить, что приведенный подход будет иметь некоторую погрешность в корректности обнаружения, поскольку уязвимости из более ранних Представлений (здесь, с индексом x) будут примешивать свое содержание к уязвимостям более поздних Представлений (здесь, с индексом y). Так, следуя примеру – в Архитектуре, полученной обратным преобразованием, помимо отсутствия механизма шифрования, будет упоминание системы аутентификации с дополнительным параметром в виде некоего пароля; а в Алгоритмах – помимо входа с помощью пароля по умолчанию будет отсутствовать целый «пласт» функционала по шифрованию данных, который потенциально должен использоваться и при хранении и проверке пароля. При этом, каждая уязвимость из другого Представления, скорее всего, не будет иметь явных признаков для обнаружения соответствующим способом.

Как результат, Представление $\hat{p}^m$ имеет сразу две уязвимости – x и y, а, следовательно, $\hat{p}^x$ и $\hat{p}^y$

после восстановления из $\hat{p}^m$, имеет такое же их количество; это в конечном итоге может привести к ошибкам в работе D^x и D^y, поскольку способы их работы ориентированы и, следовательно, наиболее результативны только для своего Представления. Для частичного нивелирования данного негативного эффекта при обнаружении уязвимостей на различных Представлениях можно применить следующий «реверс-инженерный финт» – преобразовать $\hat{p}^m$ сначала в $\hat{p}^x$, затем обнаружить уязвимость в нем, а после этого получить $\hat{p}^y$ (или прямым преобразованием из $\hat{p}^y$ или обратным их $\hat{p}^m$) для обнаружения уязвимости в нем, но уже с учетом уязвимости $\hat{p}^x$. Основная идея заключается в том, что более раннее Представление x оперирует более абстрактной базой содержания (в примере – Архитектура не использует элементы блок-схем, и, следовательно, небезопасная проверка пароля по умолчанию практически не будет влиять на обнаружение уязвимости этого Представления). Таким образом, обнаружив уязвимость в более раннем Представлении, можно минимизировать ее влияние на обнаружение уязвимостей во всех последующих Представлениях (в примере – учет того, что в Архитектуре достоверно присутствует уязвимость в виде отсутствия системы шифрования, позволит при обнаружении встроенного пароля считать, что хранение пароля не в зашифрованном виде является отдельно стоящей и уже детектированной проблемой безопасности).

Еще более реалистичной ситуацией является наличие сразу нескольких уязвимостей в одном Представлении – о чем далее.

Задача 3. Обнаружить несколько уязвимостей, которые внесены в одно Представление a, если для анализа есть более позднее Представление m.

Пример аналогичен предыдущему за исключением того, что в Алгоритмах (Представление a) присутствуют сразу две следующих уязвимости: (v_1^a) – внесение функционала в виде дополнительных проверок для входа с помощью пароля по умолчанию; (v_2^a) – изменение функционала путем изменения логина для существующей учетной записи для смены текущего администратора.

Запись такого Представления m с уязвимостью (внесенной операцией I^a) имеет следующий вид:

$$m > a: \hat{p}^m = T^{m-1\to m} \dots T^{a\to a+1} \circ I_1^a \circ I_2^a \circ T^{a-1\to a} \circ \dots \circ \circ \dots \circ T^{0\to 1} \circ p^0.$$

Применение для обнаружения D^a будет нести вероятность того, что две уязвимости выделятся, как единая – композитная, поскольку операции $I^{a1} \circ I^{a2}$ получат содержание, отличное от c^0 и трактуемое, как одна уязвимость:

$$(I_1{}^a \circ I_2^a)_c \circ c^a = c^a \to \hat{c}_2{}^a \to \hat{c}_1{}^a = c^a \to \hat{c}_{12}{}^a = = (I_{12}{}^a)_c \circ c^a,$$

где $\hat{c}_2{}^a$ и $\hat{c}_1{}^a$ – содержания, получаемые после каждого внедрения уязвимостей; $\hat{c}_{12}{}^a$ – содержание, полученное после совместного применения $I_1{}^a$ и I_2^a, которым можно сопоставить некое единое внедрение $I_{12}{}^a$ новой крупной уязвимости.

Для разрешения этой ситуации можно предположить более «умный» способ обнаружения (вместо корректного, но в данном случае тривиального нахождения отличий в содержании с помощью оператора «\»), который бы производил некоторую группировку отличий в содержании. Последнее обосновано тем, что с большой вероятностью уязвимость (в классическом понимании) является некоторым законченным и обособленным изменением в программе, а, следовательно, ее логика в базисе содержания будет строиться на близких элементах и, так или иначе, локализована в одной области. Графическая интерпретация содержания с двумя уязвимостями, поясняющая и указанный «умный» способ, в манере, аналогичной интерпретации подклассификаций уязвимостей, представлена на рисунке 2.

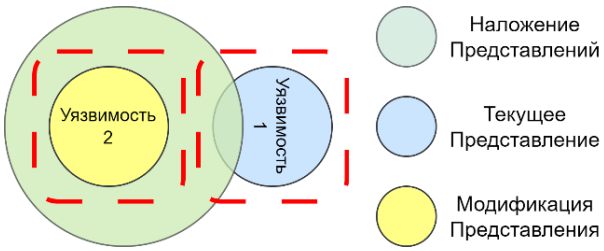

Рис. 2. Графическая интерпретация примера обнаружения двух уязвимостей в одном Представлении

Fig. 2. Graphical Interpretation of a Two Vulnerabilities Detection Example in One Representation

Согласно рисунку 2, хотя новое содержание и было получено единым изменением текущего, однако в нем все же можно выделить две группы полученных отличий – добавление (Уязвимость 1) и модификация (Уязвимость 2) элементов.

Классы же уязвимости для примера будут следующими:

$$\begin{cases} \mathbb{C}_{v_1^a} = \mathbb{C}_{ML++} \\ \mathbb{C}_{v_2^a} = \mathbb{C}_{ML\triangle\triangle} \end{cases}.$$

В заключение к приведенной аналитической Модели нужно отметить, что, хотя некоторые Утверждения (такие, как 6, 8 и 10) и примеры задач могут показаться очевидными (кстати, не без лишних оснований), однако это лишь подтверждает корректность их аналитической записи и применимость предложенного аппарата, а также обосновывает саму концепцию и эволюционные процессы в жизненном цикле программ, механизмы появления и способы обнаружения уязвимостей, а также их авторскую классификацию.

3. ЧАСТНАЯ АНАЛИТИЧЕСКАЯ МОДЕЛЬ

Сформированная аналитическая Модель жизненного цикла программы (с возможными уязвимостями) является обобщенной, поскольку предназначена для описания процессов, не зависящих от конкретных (т. е. существующих на данный момент) вариаций Представлений: их формы, содержания, способов получения и восстановления, классов уязвимостей, способов их поиска и т. п. Так, например, согласно Схеме, из Архитектуры могут быть получены Алгоритмы и Графический код, но не Дерево абстрактного синтаксиса код, а Исходный код разделяется не только на Классический, но и Метакод генерации и Сценарный. Таким образом, за обобщенной Моделью жизненного цикла, несущей более теоретический смысл, логично должно последовать создание частной Модели – которая бы отражала (т. е. моделировала) текущее состояние области программной и реверс-инженерии.

Для этого кратко запишем частную Модель на основании обобщенной Модели, указывая как формальную запись Представления, так способ и свойства его получения и восстановления, вносимые и обнаруживаемые уязвимости, а также способы их поиска (таблица 1); для сокращения записи будем, в случае необходимости, указывать в качестве номера Представления (правый верхний индекс) их перечисление. В качестве же самого номера Представления будем использовать номера на Схеме.

ТАБЛИЦА 1. Частная модель жизненного цикла программы с уязвимостями на базе эволюции ее Представлений

TABLE 1. A Life Cycle Particularly Model of the Program with Vulnerabilities Based on the Evolution of its Representations

Представления					Уязвимости		
№	Название	Формальная запись	Получение	Восстановление	Вносимые, $(\mathbb{C}_v)_{level}$	Обнаруживаемые, $(\mathbb{C}_v)_{level}$	Способ поиска, $\mathbb{C}_{detect}$
1	Идея	$p^1 \equiv \langle f^1 \mid c^1 \rangle$ $\mathbb{C}^1_{form} = \mathbb{C}_V$	Отсутствует (исходное)	$p^1 = \bar{T}^{2\to1} \circ p^2$ $\mathbb{C}^{2\to1}_{transform} = \mathbb{C}_{AT}$	Нет	Нет	Нет
2	Концептуальная модель	$p^2 \equiv \langle f^2 \mid c^2 \rangle$ $\mathbb{C}^2_{form} \in \{\mathbb{C}_V, \mathbb{C}_G\}$	$p^2 = T^{1\to2} \circ p^1$ $\mathbb{C}^{1\to2}_{transform} = \mathbb{C}_{ST}$	$p^2 = \bar{T}^{3\to2} \circ p^3$ $\mathbb{C}^{3\to2}_{transform} = \mathbb{C}_{AT}$	CL	CL	HD
3	Архитектура	$p^3 \equiv \langle f^3 \mid c^3 \rangle$ $\mathbb{C}^3_{form} \in \{\mathbb{C}_V, \mathbb{C}_G\}$	$p^3 = T^{2\to3} \circ p^2$ $\mathbb{C}^{2\to3}_{transform} = \mathbb{C}_{ST}$	$p^3 = \bar{T}^{4\to3} \circ p^4$ $p^3 = \bar{T}^{5\to3} \circ p^5$ $p^3 = \bar{T}^{6\to3} \circ p^6$ $p^3 = \bar{T}^{7\to3} \circ p^7$ $p^3 = \bar{T}^{8\to3} \circ p^8$ $\mathbb{C}^{4,5,6,7,8\to3}_{transform} = \mathbb{C}_{AT}$	HL	HL	HD
4	Двухмерная структурная схема	$p^4 \equiv \langle f^4 \mid c^4 \rangle$ $\mathbb{C}^4_{form} \in \{\mathbb{C}_P, \mathbb{C}_G\}$	$p^4 = T^{3\to4} \circ p^3$ $\mathbb{C}^{3\to4}_{transform} = \mathbb{C}_{ST}$	$p^4 = \bar{T}^{13\to4} \circ p^4$ $\mathbb{C}^{13\to4}_{transform} = \mathbb{C}_{RT}$	HL, ML	HL, ML	HD, AD
5	Функциональная диаграмма	$p^5 \equiv \langle f^5 \mid c^5 \rangle$ $\mathbb{C}^5_{form} \in \{\mathbb{C}_P, \mathbb{C}_G\}$	$p^5 = T^{3\to5} \circ p^3$ $\mathbb{C}^{3\to5}_{transform} = \mathbb{C}_{ST}$	$p^5 = \bar{T}^{13\to5} \circ p^5$ $\mathbb{C}^{13\to5}_{transform} = \mathbb{C}_{RT}$	HL, ML	HL, ML	HD, AD
6	Блок-схема	$p^6 \equiv \langle f^6 \mid c^6 \rangle$ $\mathbb{C}^6_{form} \in \{\mathbb{C}_V, \mathbb{C}_G\}$	$p^6 = T^{3\to6} \circ p^3$ $\mathbb{C}^{3\to6}_{transform} = \mathbb{C}_{ST}$	$p^6 = \bar{T}^{9\to6} \circ p^9$ $p^6 = \bar{T}^{10\to6} \circ p^{10}$ $p^6 = \bar{T}^{11\to6} \circ p^{11}$ $\mathbb{C}^{9,10,11\to6}_{transform} = \mathbb{C}_{AT}$	ML	ML	HD, AD
7	Структурограмма	$p^7 \equiv \langle f^7 \mid c^7 \rangle$ $\mathbb{C}^7_{form} \in \{\mathbb{C}_V, \mathbb{C}_G\}$	$p^7 = T^{3\to7} \circ p^3$ $\mathbb{C}^{3\to7}_{transform} = \mathbb{C}_{ST}$	$p^7 = \bar{T}^{9\to7} \circ p^9$ $p^7 = \bar{T}^{10\to7} \circ p^{10}$ $p^7 = \bar{T}^{11\to7} \circ p^{11}$ $\mathbb{C}^{9,10,11\to7}_{transform} = \mathbb{C}_{AT}$	ML	ML	HD, AD
8	Псевдокод	$p^8 \equiv \langle f^8 \mid c^8 \rangle$ $\mathbb{C}^8_{form} = \mathbb{C}_P$	$p^8 = T^{3\to8} \circ p^3$ $\mathbb{C}^{3\to8}_{transform} = \mathbb{C}_{ST}$	$p^8 = \bar{T}^{9\to8} \circ p^9$ $p^8 = \bar{T}^{10\to8} \circ p^{10}$ $p^8 = \bar{T}^{11\to8} \circ p^{11}$ $\mathbb{C}^{9,10,11\to8}_{transform} = \mathbb{C}_{AT}$	ML	ML	HD, AD
9	Классический код	$p^9 \equiv \langle f^9 \mid c^9 \rangle$ $\mathbb{C}^9_{form} = \mathbb{C}_P$	$p^9 = T^{6\to9} \circ p^6$ $p^9 = T^{7\to9} \circ p^7$ $p^9 = T^{8\to9} \circ p^8$ $p^9 = T^{10\to9} \circ p^{10}$ $\mathbb{C}^{6,7,8\to9}_{transform} = \mathbb{C}_{ST}$ $\mathbb{C}^{10\to9}_{transform} = \mathbb{C}_{GT}$	$p^9 = \bar{T}^{13\to9} \circ p^{13}$ $\mathbb{C}^{13\to9}_{transform} = \mathbb{C}_{RT}$	ML, LL	ML, LL	HD, AD
10	Метакод генерации	$p^{10} \equiv \langle f^{10} \mid c^{10} \rangle$ $\mathbb{C}^{10}_{form} = \mathbb{C}_P$	$p^{10} = T^{6\to10} \circ p^6$ $p^{10} = T^{7\to10} \circ p^7$ $p^{10} = T^{8\to10} \circ p^8$ $\mathbb{C}^{6,7,8\to10}_{transform} = \mathbb{C}_{ST}$	$p^{10} = \bar{T}^{9\to10} \circ p^9$ $\mathbb{C}^{9\to10}_{transform} = \mathbb{C}_{AT}$	ML, LL	ML, LL	HD, AD

Представления					Уязвимости		
№	Название	Формальная запись	Получение	Восстановление	Вносимые, $(\mathbb{C}_v)_{level}$	Обнаруживаемые, $(\mathbb{C}_v)_{level}$	Способ поиска, $\mathbb{C}_{detect}$
11	Сценарный код	$p^{11} \equiv \langle f^{11} \mid c^{11} \rangle$ $\mathbb{C}^{11}_{form} = \mathbb{C}_P$	$p^{11} = T^{6\to11} \circ p^6$ $p^{11} = T^{7\to11} \circ p^7$ $p^{11} = T^{8\to11} \circ p^8$ $\mathbb{C}^{6,7,8\to11}_{transform} = \mathbb{C}_{ST}$	$p^{11} = \bar{T}^{13\to11} \circ p^{13}$ $\mathbb{C}^{13\to11}_{\overline{transform}} = \mathbb{C}_{RT}$	ML, LL	ML, LL	HD, AD
12	Ассемблерный код	$p^{12} \equiv \langle f^{12} \mid c^{12} \rangle$ $\mathbb{C}^{12}_{form} = \mathbb{C}_P$	$p^{12} = T^{13\to12} \circ p^{13}$ $\mathbb{C}^{13\to12}_{transform} = \mathbb{C}_{GT}$	$p^{12} = \bar{T}^{14\to12} \circ p^{14}$ $p^{12} = \bar{T}^{15\to12} \circ p^{15}$ $\mathbb{C}^{14,15\to12}_{\overline{transform}} = \mathbb{C}_{RT}$	LL	LL	HD
13	Дерево абстрактного синтаксиса	$p^{13} \equiv \langle f^{13} \mid c^{13} \rangle$ $\mathbb{C}^{13}_{form} = \mathbb{C}_B$	$p^{13} = T^{4\to13} \circ p^4$ $p^{13} = T^{5\to13} \circ p^5$ $p^{13} = T^{9\to13} \circ p^9$ $p^{13} = T^{11\to13} \circ p^{11}$ $\mathbb{C}^{4,5,9,11\to13}_{transform} = \mathbb{C}_{GT}$	$p^{13} = \bar{T}^{12\to13} \circ p^{12}$ $p^{13} = \bar{T}^{14\to13} \circ p^{14}$ $p^{13} = \bar{T}^{15\to13} \circ p^{15}$ $\mathbb{C}^{12,14,15\to13}_{\overline{transform}} = \mathbb{C}_{AT}$	LL	LL	AD
14	Машинный код	$p^{14} \equiv \langle f^{14} \mid c^{14} \rangle$ $\mathbb{C}^{14}_{form} = \mathbb{C}_B$	$p^{14} = T^{12\to14} \circ p^{12}$ $p^{14} = T^{13\to14} \circ p^{13}$ $p^{14} = T^{15\to14} \circ p^{15}$ $\mathbb{C}^{12,13,15\to14}_{transform} = \mathbb{C}_{GT}$	Отсутствует (конечное)	AL	LL, AL	HD, AD
15	Байт-код	$p^{15} \equiv \langle f^{15} \mid c^{15} \rangle$ $\mathbb{C}^{15}_{form} = \mathbb{C}_B$	$p^{15} = T^{12\to15} \circ p^{12}$ $p^{15} = T^{13\to15} \circ p^{13}$ $\mathbb{C}^{12,13\to15}_{transform} = \mathbb{C}_{GT}$	$p^{15} = \bar{T}^{14\to15} \circ p^{14}$ $\mathbb{C}^{14\to15}_{\overline{transform}} = \mathbb{C}_{AT}$	AL	LL, AL	HD, AD

4. ЭКСПЕРИМЕНТ ПО ВОЗНИКНОВЕНИЮ УЯЗВИМОСТЕЙ

В качестве доказательства работоспособности Модели в части отражения уязвимостей приведем результаты проведенного автором ретроспективно-фактологического эксперимента. Суть эксперимента состоит в анализе баз уязвимостей на предмет отнесения моментов их возникновения к соответствующим Представлениям. Как результат, будет обосновано как существование самих Представлений, так и классов уязвимостей в них. В качестве информации об уязвимостях использовалась база CVE (*аббр. от англ.* Common Vulnerabilities and Exposures) и научные статьи из цифровой библиотеки IEEE Xplore.

Идея. Согласно Модели жизненного цикла, уязвимости в Представлении отсутствуют. В подтверждение этого соответствующих уязвимостей в базах данных найдено не было.

Концептуальная модель. Уязвимость CVE-2022-20660 заключается в хранении пароля администратора для IP-телефона Cisco (серии 7800) во флэш-памяти в открытом виде. Уязвимость относится к данному Представлению, поскольку в общей концепции этой модели телефона отсутствует понятие безопасного хранилища такого рода информации и криптографических функций в обеспечение этого. Подтверждение уязвимости: https://seclists.org/fulldisclosure/2022/Jan/34, а ее класс: $\mathbb{C}_v = \mathbb{C}_{CL-+}$.

Архитектура. Уязвимость CVE-2022-32259 заключается в том, что системные образы для установки или обновления продукта SINEMA Remote Connect Server содержат сценарии модульного тестирования, включающие в том числе конфиденциальную информацию. Уязвимость относится к данному Представлению, т. к. в архитектуре версии продукта, предназначенной для распространения и эксплуатации конечными потребителями, была оставлена подсистема тестирования. Подтверждение уязвимости: https://cert-portal.siemens.com/productcert/pdf/ssa-484086.pdf, а ее класс: $\mathbb{C}_v = \mathbb{C}_{\text{HL++}}$.

Группа «Графический код» (Двухмерная структурная схема, Функциональная диаграмма). Поскольку данное Представление, по сути, совмещает в себе два – Алгоритмы и Исходный код, то явного отнесения уязвимостей в CVE к нему найдено не было. Тем не менее, ряд частых ошибок в FBL (переходящих к уязвимостям в Представлении) был описан в статье [20]; а именно следующих: 1) неправильное использование блока таймера из-за схожести их функциональных возможностей (блок TON *аббр. от англ.* ON Delay Timer – откладывает начало события, а блок TOF *аббр. от англ.* OFF Delay Timer – откладывает остановку события); 2) ошибочный порядок входов логических операций (так, в отличие от блока AND для логической операции «И», у блока MUX для мультиплексирования их порядок имеет значение); 3) неправильное использование инвертора (часто графическое изображение операции в виде мелкого кружка приводит к ее ненужному добавлению или ошибочному опусканию на диаграмме); 4) неверная запись переменных, что ведет к некорректным присваиваниям или вычислениям (причины этого можно отнести к особенностям графической разработки алгоритмов и кода). В статье [20], помимо подтверждения указанных уязвимостей, описывается авторский метод структурного тестирования, обнаруживающий их.

Классы указанных уязвимостей (в порядке перечисления ошибок) следующие:

1) $\mathbb{C}_v = \mathbb{C}_{\mathrm{ML}\triangle?}$ (индекс «?» означает, что точный класс воздействия на информационные потоки требует уточнения, поскольку зависит от конкретной FBL-схемы);
2) $\mathbb{C}_v = \mathbb{C}_{ML\triangle\triangle}$;
3) $\mathbb{C}_v = \mathbb{C}_{ML+\triangle}$ или $\mathbb{C}_v = \mathbb{C}_{ML-\triangle}$;
4) $\mathbb{C}_v = \mathbb{C}_{\mathrm{LL}\triangle\triangle}$.

Группа «Алгоритм» (Блок-схема, Структурограмма, Псевдокод). Уязвимость CVE-2021-4021 заключается в неконтролируемом потреблении ресурсов программным продуктом для реверс-инжиниринга *Radare2* (и, как следствие, DoS-последствиям). Условием проявления уязвимости является отображение раздела файла формата ELF (64-битной версии) для процессорной архитектуры MIPS, который имеет большой раздел и заполнен нулевыми значениями. Уязвимость относится к данному Представлению, т. к. исходные алгоритмы продукта не содержали ограничений на максимально возможную анализируемую последовательность нулевых операций (так называемых NOP-инструкций), что приводило к истощению ресурсов. Подтверждение уязвимости: https://seclists.org/fulldisclosure/2022/Jan/34, а ее класс: $\mathbb{C}_v = \mathbb{C}_{ML-+}$. Трудно не отметить парадоксальность и комичность ситуации, которая заключается в том, что средство для анализа кода на предмет уязвимостей низкого уровня (в машинном коде) содержит уязвимости существенно более высокого уровня (в алгоритмах).

Группа «Исходный код» (Классический код, Метакод генерации, Сценарный код). Уязвимость CVE-2022-39288 заключается в возникновении сбоя в Web-фреймворке *fastify* при передаче некорректного Content-Type в заголовке запроса. Уязвимость относится к данному Представлению, т. к. используемый алгоритм получения парсера для определенного типа содержимого корректен, но его реализация использовала в качестве хранилища для таких парсеров JavaScript-объект, что позволяло передать туда любую строку, являющуюся ключом свойства класса Object и вызывать падение Web-сервера. Подтверждение уязвимости: https://hackerone.com/reports/1715536, а ее класс: $\mathbb{C}_v = \mathbb{C}_{LL\triangle+}$.

Ассемблерный код. Уязвимость CVE-2022-38453 заключается в потере конфиденциальности и потенциальной возможности получения полного контроля над CMS8000 (медицинское устройство для мониторинга состояния пациента https://5.imimg.com/data5/SELLER/Doc/2021/7/NM/EK/ZD/89804963/contec-cms-8000-multipara-patient-monitor-1-.pdf) из-за компиляции важных системных файлов с отладочной информацией. Уязвимость относится к данному Представлению, т. к. даже при безопасности всех более ранних Представлений компиляция программы с некорректными флагами делает генерируемый Ассемблерный код уязвимым к информационным атакам, что может привести к критическим угрозам в медицинской сфере. Подтверждение уязвимости: https://www.cisa.gov/uscert/ics/advisories/icsma-22-244-01, а ее класс: $\mathbb{C}_v = \mathbb{C}_{LL++}$.

Дерево абстрактного синтаксиса. Уязвимость CVE-2020-15294 заключается в неверной работе алгоритмов оптимизации, что в результате приводит к генерации кода, который дважды разыменовывает (т. е. получает значение по указателю) один и тот же адрес, потенциально приводя к выполнению произвольного кода. Уязвимость относится к данному Представлению, т. к. оптимизация выполняется на внутреннем Представлении входной программы в компиляторе. Подтверждение уязвимости: https://www.bitdefender.com/support/security-advisories/compiler-optimization-removal-modification-security-critical-code-vulnerability-bitdefender-hypervisor-introspection-va-9339/, а ее класс: $\mathbb{C}_v = \mathbb{C}_{LL++}$.

Машинный код. Уязвимость CVE-2021-38300 заключается в возможности выполнения произвольного кода в kernel space (*перев. на рус.* пространство ядра) на MIPS-архитектуре из-за упущения в JIT-компиляторе байт-кода cBPF (*аббр. от англ.* classic Berkeley Packet Filters – технология некоторых операционных систем, позволяющая выполнять байт-код на уровне ядра для эффективного и безопасного анализа сетевого трафика [21]), которое приводило к генерации машинного кода с недопустимым смещением при использовании условных конструкций cBPF. Уязвимость относится к данному Представлению, т. к. даже при корректности исходного кода, формирующего cBPF байт-код, среда его трансляции в машинный код (т. е. JIT-компилятор) создает угрозу ИБ. Подтверждение уязвимости: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=37cb28ec7d3a36a5bace7063a3dba633ab110f8b, а ее класс: $\mathbb{C}_v = \mathbb{C}_{\mathrm{LL}\triangle+}$.

Байт-код. Уязвимость CVE-2022-31740 заключается в неверной генерации байт-кода кода утилитой компиляции WASM (*сокр. от* WebAssembly – средство для запуска скомпилированного кода в современных Web-браузерах [22]) на процессорной архитектуре Arm64, что приводило к ошибкам в распределении регистров и потенциально опасным сбоям. Уязвимость относится к данному Представлению, поскольку вносится при генерации выполняемой сборки из корректного кода высокоуровневого языка программирования (например, C++). Подтверждение уязвимости: https://bugzilla.mozilla.org/show_bug.cgi?id=1766806, а ее класс: $\mathbb{C}_v = \mathbb{C}_{AL\triangle+}$.

Исходя из приведенных и подтвержденных уязвимостей, реально существующих (или существовавших) в каждом из Представлений (или их группах), можно утверждать, что частная Модель в части уязвимостей является обоснованной.

5. ЭКСПЕРИМЕНТ ПО ЭВОЛЮЦИИ УЯЗВИМОСТЕЙ

Для демонстрации отображения уязвимостей программы в процессе эволюции ее Представлений проведем следующий практический эксперимент. В качестве программы возьмем максимально тривиальную, призванную лишь произвести указанную демонстрацию; суть программы (или другими словами, содержание) будет заключаться в делении двух чисел; естественно, операция деления должна осуществляться по всем правилам арифметики, в которой *деление на ноль неопределено*.

Искусственно заложим уязвимость в Концептуальную модель и покажем, как меняется ее отображение в процессе разработки программы (вплоть до Машинного кода) – т. е. как выглядят все Представления с и без данной уязвимости. Суть уязвимости будет заключаться в том, что пропущена проверка делителя на ноль (правила проведения арифметической операции деления нарушены). Уязвимость имеет класс $\mathbb{C}_v = \mathbb{C}_{CL-+}$, поскольку пропущен функционал, который приводит к обработке некорректных данных.

Невозможность обнаружения уязвимости в Представлениях, отличных от того, в котором она была внесена, также будет считаться подтверждением Модели и всех изложенных ранее идей данной предметной области. В качестве языка программирования выберем классический язык *C*, а в качестве Центрального процессора выполнения – Intel x86-64.

Далее кратко распишем все преобразования между Представлениями данной программы.

Идея. Замысел программы заключается в следующем: «Программа должна корректно осуществлять арифметическую операцию деления над данными пользователя».

Концептуальная модель. Форма Представления приведена на рисунке 3. Согласно Представлению, инициатор программы (Оператор) вызывает Операцию деления, которая на основании Аргументов выдает Результат вычисления. При этом, в соответствии с Идеей, деление должно выполняться согласно Правилам арифметики. Однако, в следствие внесенной уязвимости, учет правила деления на ноль был пропущен и данный элемент в итоговой модели будет отсутствовать – т. е. произошло изменение содержания в виде уязвимости класса «потеря функционала» (здесь и далее область отсутствующего функционала помечена фоном с красными наклонными линиями, которая затем заменится на области кода с рыжим фоном). Доля пропущенного элемента от всех (т. е. некий аналог «площади покрытия» уязвимостью содержания Представления посредством его формы) составляет $\frac{1}{5} = 20$ %.

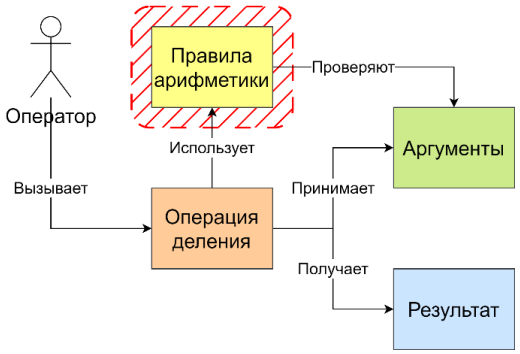

Рис. 3. Концептуальная модель программы (пример)

Fig. 3. Program Conceptual Model (Example)

Архитектура. Форма Представления приведена на рисунке 4.

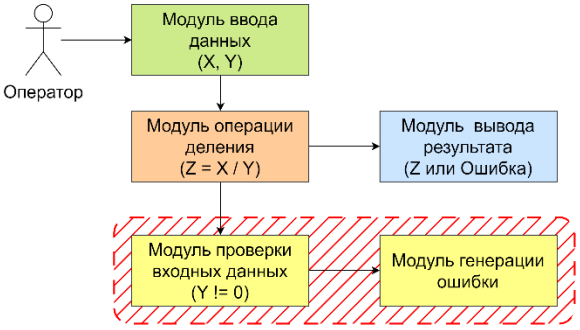

Рис. 4. Архитектура программы (примера)

Fig. 4. Program Architecture (Example)

Согласно сути Представления, полученного из Концептуальной модели, должны присутствовать точка запуска (Оператор) и 5 модулей, отвечающих как за ввод, вычисление и вывод результата деления, так и за проверку делителя на 0 с выводом ошибки. Однако, вследствие уязвимости в предыдущем Представлении, последние два модуля (отвечающие за корректность деления) были пропущены. Отметим, что несмотря на то, что уже на этом Представлении можно предположить потерю функционала, связанную с отсутствием модулей проверки и генерации ошибки, однако исходная уязвимость уже стала «распространяться» по Представлению – пропущены 2 из 6 элементов (т. е. зона охвата составила ~33 %).

Блок-схема алгоритмов. Форма Представления приведена на рисунке 5. Представление имеет вид классической Блок-схемы, по которой (после элемента Начало) производится ввод двух операндов (делимого *X* и делителя *У*), затем второй операнд сравнивается с нулевым значением. Если делитель равен 0, то выводится сообщение об ошибке; в ином случае, вычисляется частное (*Z*), которое возвращается из алгоритма. Блок-схема завершается элементом Конец. Вследствие уязвимости, в Блок-схеме отсутствует проверка делителя на 0 и вывод ошибки, а доля отсутствующего функционала составляет – $\frac{2}{7} = \sim 29$ %.

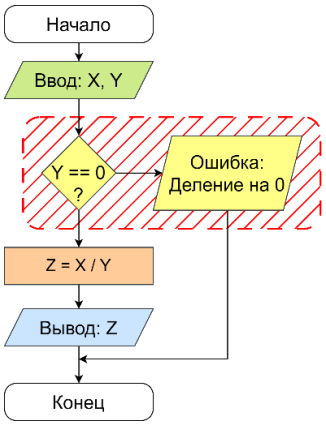

Рис. 5. Алгоритм программы (пример)

Fig. 5. Program Algorithm (Example)

Классический исходный код. В качестве данного Представления приведем одну из типовых реализаций алгоритма на языке *C*, представленную в следующем листинге:

```
#include <stdlib.h>
#include <stdio.h>
int divide (int x, int y)
{
  if (y == 0)
  {
    printf ("Divide by zero\n");
    exit (1);
  }
  int z = x / y;
  return z;
}
```

Код полностью отражает логику работы Блок-схемы программы. Область пропущенного функционала, отмеченная (здесь и далее) рыжим фоном, составляет примерно 3 из 8 строк (т. е. ~38 %), которые не включают несущественные для выполнения конструкции (в данном случае начало и завершение областей видимости переменных – посредством токенов «{» и «}»).

Дерево абстрактного синтаксиса. Представление, как правило, представляет собой совокупность графов (например, со структурой кода и потоком управления), таблиц (например, типов и символов кода) и другой служебной информации; при этом оно достаточно сильно зависит от конкретной реализации компилятора (при том, что может применяться даже для построения логической структуры тестовых документов [23]). Поэтому, без потери корректности хода эксперимента, его можно опустить.

Ассемблерный код. Представление, являющееся результатом компиляции Исходного кода (через Дерево абстрактного синтаксиса) с помощью компилятора из состава Microsoft Visual Studio Community 2019 (версия: Microsoft (R) C/C++ Optimizing Compiler Version 19.29.30145 for x86), имеет вид следующего листинга (привязка ассемблерных инструкций к Исходному коду указана в виде комментариев с префиксом «;»; конструкции ассемблера, не продуцирующие машинных инструкций, отмечены серым фоном).

```
_divide    PROC
; int divide(int x, int y)
; {
            push    ebp
            mov     ebp, esp
            push    ecx
;  if (y == 0)
;  {
            cmp     DWORD PTR _y$[ebp], 0
            jne     SHORT $LN2@divide
;    printf("Y = 0\n");
            push    OFFSET $SG10771
            call    _printf
            add     esp, 4
;    exit(1);
            push    1
            call    _exit
;  }
$LN2@divide:
;  int z = x / y;
            mov     eax, DWORD PTR _x$[ebp]
            cdq
            idiv    DWORD PTR _y$[ebp]
            mov     DWORD PTR _z$[ebp], eax
            mov     eax, DWORD PTR _z$[ebp]
$LN3@divide:
;  return z;
            mov     esp, ebp
            pop     ebp
            ret     0
; }
_divide    ENDP
```

Согласно тому, что рыжая область Ассемблерного кода, отмечающая пропущенный функционал, состоит из 7 ассемблерных строк, а весь код программы (без комментариев и не продуцирующих инструкции строк) – из 18, то область измененного содержания составляет $\frac{7}{18}$ ~39 %. Таким образом, область «распространения» уязвимости увеличилась еще больше.

Машинный код. Представление, полученное ассемблированием предыдущего, будет иметь бинарный вид, который может быть записан с помощью последовательности байт (в 16-ричной форме) следующим образом:

```
55 8B EC 51 83 7D 0C 00 75 14 68 C8 00 00 00 E8
78 00 00 00 83 C4 04 6A 01 E8 C2 00 00 00 8B 45
08 99 F7 7D 0C 89 45 FC 8B 45 FC 8B E5 5D C3
```

Очевидно, что данная форма Представления абсолютно не подходит для анализа экспертом вручную. По аналогии с предыдущими Представлениями, область с рыжим фоном соответствует уязвимости (естественно, в смысле изменения функционала); ее охват составляет $\frac{26}{47} = \sim 55$ %.

Подведем итоги эксперимента с примером данной тривиальной программы. Во-первых, базовая идея о преобразовании Представлений от Идеи до Машинного кода подтверждена. Во-вторых, уязвимость, заложенная в более раннем Представле-

нии, «живет» во всех последующих. И, в-третьих, динамика охвата уязвимостью Представления практически постоянно растет (за исключением Блок-схемы алгоритмов и отсутствующего Дерева абстрактного синтаксиса), о чем свидетельствует гистограмма на рисунке 6.

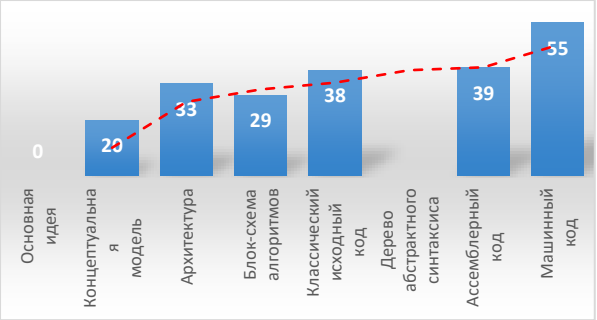

Рис. 6. Охвата уязвимостью Представлений (в процентах)

Fig. 6. Representations Coverage with Vulnerability (in Percentage)

Так, по сравнению с первым Представлением, охват уязвимости на последнем вырос в $\frac{55}{20} = \sim 2{,}75$ раза и составил 55 %. Естественно, такая динамика уязвимостей зависит от свойств конкретной программы. Однако тот факт, что даже для тривиальной программы деления двух чисел, реализуемой достаточно близкими способами, произошло сильнейшее увеличение охвата уязвимости (что в ряде случаев будет приводить к «затерянию» вредоносного кода среди безопасного, существенно затрудняя его обнаружение и «вычленение»), говорит о крайней важности учета не только статических, но и ее динамических свойств (момент возникновения и обнаружения, изменения в процессе преобразования Представлений, динамика охвата, взаимодействие с другими уязвимостями [24, 25] и т. п.).

6. ЗАКЛЮЧЕНИЕ

Проведенное в работе аналитическое моделирование программы с уязвимостями с позиции эволюции ее Представлений, основанное на результатах из предыдущей части цикла статей и подкрепленное реальными примерами, позволяет более полно взглянуть на динамику процесса создания программ и существования уязвимостей с формальной точки зрения, что приводит к ряду следующих умозаключений.

Во-первых, все объекты и процессы, упоминаемые при описании Схемы жизненного цикла, а также производные от них, переведены в разряд Утверждений и записаны в аналитическом виде. Как результат, была построена обобщенная аналитическая Модель, находящаяся в строгом соответствии с предыдущими выкладками.

Во-вторых, учет способов преобразования между Представлениями при создании программ и их восстановлении позволил создать частную аналитическую Модель, отражающую текущее состояние области программной инженерии.

В-третьих, введение в Модели понятия уязвимости и операций, связанных с ними (внесение, обнаружение, классификация) расширил понимание этого достаточно сложного и противоречивого объекта области ИБ программ.

Проведенные эксперименты по возникновению уязвимостей и их эволюции между Представлениями обосновывают сделанные выводы. Продолжение же работы должно быть направлено на теоретико-практический аспект способов обратного преобразования Представлений, что (как уже было многократно сказано) существенно повысит возможности по обнаружению и нейтрализации уязвимостей в программах. В интересах этого автор видит перспективным развитие отдельного направления *интеллектуального реверс-инжиниринга*, например, на базе генетической декомпиляции [26–28]. Основная идея последней заключается в итеративном приближении предыдущих Представлений к таким форме и содержанию, чтобы их прямое преобразование давало бы в точности заданное Представление. Как результат, удастся снизить необходимость в сверхвостребованных экспертах по безопасности программного кода.

Список источников

1. Kotenko I., Izrailov K., Buinevich M. Static Analysis of Information Systems for IoT Cyber Security: A Survey of Machine Learning Approaches // Sensors. 2022. Vol. 22. Iss. 4. P. 1335. DOI:10.3390/s22041335

2. Trevizan R.D., Obert J., De Angelis V., Nguyen Tu.A., Rao V.S., Chalamala B.R. Cyberphysical Security of Grid Battery Energy Storage Systems // IEEE Access. 2022. Vol. 10. PP. 59675–59722. DOI:10.1109/ACCESS.2022.3178987

3. Cho C.-S., Chung W.-H., Kuo S.-Y. Cyberphysical Security and Dependability Analysis of Digital Control Systems in Nuclear Power Plants // IEEE Transactions on Systems, Man, and Cybernetics: Systems. 2016. Vol. 46. Iss. 3. PP. 356–369. DOI:10.1109/TSMC.2015.2452897

4. 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

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

6. Пароджанов В.Д. Алгоритмические языки и программирование: ДРАКОН: учебное пособие для среднего профессионального образования. М.: Издательство Юрайт, 2023. 436 с.

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

8. 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

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

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

11. Басов А.С. Классификация языков программирования и их особенности // Вестник науки. 2020. Т. 2. № 8(29). С. 95–101.

12. 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.

13. 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

14. Пирогов В. Ассемблер для Windows. Санкт-Петербург: БХВ-Петербург, 2012. 896 с.

15. 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

16. Кричанов М.Ю., Чепцов В.Ю. Защищенная UEFI-прошивка для виртуальных машин // Системный администратор. 2021. № 11(228). С. 75–81.

17. 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.

18. 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. 47–50.

19. Израилов К.Е., Татарникова И.М. Подход к анализу безопасности программного кода с позиции его формы и содержания // VIII Международная научно-техническая и научно-методическая конференция «Актуальные проблемы инфотелекоммуникаций в науке и образовании» (Санкт-Петербург, Россия, 27–28 февраля 2019). СПб: СПбГУТ, 2019. С. 462–467.

20. Eunkyoung J., Seungjae J., Hojung B., Sungdeok C., Junbeom Y., Geeyong P., et al. Testing of Timer Function Blocks in FBD // Proceedings of 13th Asia Pacific Software Engineering Conference (APSEC'06, Bangalore, India, 06–08 December 2006). IEEE, 2006. PP. 243–250. DOI:10.1109/APSEC.2006.55

21. McCanne S., Jacobson V. The BSD Packet Filter: A New Architecture for User-Level Packet Capture // Proceedings of the Winter USENIX Technical Conference, San Diego, USA, 25–29 January 1993. USENIX Association, 1993.

22. Kim M., Jang H., Shin Y. Avengers, Assemble! Survey of WebAssembly Security Solutions // Proceedings of 15th International Conference on Cloud Computing (CLOUD, Barcelona, Spain, 10–16 July 2022). IEEE, 2022. PP. 543–553. DOI:10.1109/CLOUD55607.2022.00077

23. Чувилин К.В. Параметрический подход к построению синтаксических деревьев для частично формализованных текстовых документов // Машинное обучение и анализ данных. 2016. Т. 2. № 2. С. 201–217.

24. 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.

25. 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.

26. Израилов К.Е. Концепция генетической декомпиляции машинного кода телекоммуникационных устройств // Труды учебных заведений связи. 2021. Т. 7. № 4. С. 10–17. DOI:10.31854/1813-324X-2021-7-4-95-109

27. Израилов К.Е. Применение генетических алгоритмов для декомпиляции машинного кода // Защита информации. Инсайд. 2020. № 3(93). С. 24–30.

28. Израилов К.Е., Романов Н.Е. Применение генетического алгоритма для реверс-инжиниринга машинного кода // XI Международная научно-техническая и научно-методическая конференция «Актуальные проблемы инфотелекоммуникаций в науке и образовании» (Санкт-Петербург, Россия, 15–16 февраля 2022). СПб: СПбГУТ, 2022. С. 239–243.

References

1. Kotenko I., Izrailov K., Buinevich M. Static Analysis of Information Systems for IoT Cyber Security: A Survey of Machine Learning Approaches. *Sensors*. 2022;22(4):1335. DOI:10.3390/s22041335 (in Russ.)

2. Trevizan R.D., Obert J., De Angelis V., Nguyen Tu.A., Rao V.S., Chalamala B.R. Cyberphysical Security of Grid Battery Energy Storage Systems. *IEEE Access*. 2022;(10):59675–59722. DOI:10.1109/ACCESS.2022.3178987

3. Cho C.-S., Chung W.-H., Kuo S.-Y. Cyberphysical Security and Dependability Analysis of Digital Control Systems in Nuclear Power Plants. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*. 2016;46(3):356–369. DOI:10.1109/TSMC.2015.2452897

4. 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

5. 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.)

6. Parondzhanov V.D. *Algorithmic Languages and Programming: DRAGON*. Moscow: Yurajt Publ.; 2023. 436 p. (in Russ.)

7. 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.)

8. 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

9. Akhmerova A.N. Controller programming languages. Features of the application of the languages. *Nauchnyj aspekt.* 2019;3(3):340–345. (in Russ.)

10. Nassi I., Shneiderman B. Flowchart techniques for structured programming. *SIGPLAN Notices*. 8(8):12–26. DOI:10.1145/953349.953350

11. Basov A.S. Classification of Programming Languages and their Features. *Vestnik nauki*. 2020;2(8):95–101. (in Russ.)

12. 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.)

13. 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

14. Pirogov V. *Assembler for Windows*. BHV-Petersburg Publ.; 2012. 896 p. (in Russ.)

15. 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

16. Krichanov M.Y., Cheptsov V.Y. Secure UEFI firmware for virtual machines. *Sistemnyj administrator*. 2021;11(228): 75–81. (in Russ.)

17. 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.)

18. 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.)

19. Izrailov K., Tatarnikova I. An Approach to Analyzing the Security of a Software Code from the Standpoint of Its Form and Content. *Proceedings of the VIIth International Conference on Infotelecommunications in Science and Education, 27–28 February 2019, St. Petersburg, Russia*. St. Petersburg: The Bonch-Bruevich Saint-Petersburg State University of Telecommunications Publ.; 2019. p.462–467. (in Russ.)

20. Eunkyoung J., Seungjae J., Hojung B., Sungdeok C., Junbeom Y., Geeyong P., et al. Testing of Timer Function Blocks in FBD. *Proceedings of 13th Asia Pacific Software Engineering Conference, APSEC'06, 06–08 December 2006, Bangalore, India*. IEEE; 2006. p.243–250. DOI:10.1109/APSEC.2006.55

21. McCanne S., Jacobson V. The BSD Packet Filter: A New Architecture for User-Level Packet Capture. *Proceedings of the Winter USENIX Technical Conference, 25–29 January 1993, San Diego, USA*. USENIX Association; 1993.

22. Kim M., Jang H., Shin Y. Avengers, Assemble! Survey of WebAssembly Security Solutions. *Proceedings of 15th International Conference on Cloud Computing, CLOUD, 10–16 July 2022, Barcelona, Spain*. IEEE; 2022. p.543–553. DOI:10.1109/CLOUD55607.2022.00077

23. Chuvilin K.V. Parametric Approach to the Construction of Syntax Trees for Partially Formalized Text Documents. *Machine Learning and Data Analysis*. 2016;2(2):201–217. (in Russ.)

24. 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.)

25. 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.)

26. Izrailov K. The Genetic Decompilation Concept of the Telecommunication Devices Machine Code. *Proc. of Telecom. Universities*. 2021;7(4):10–17. DOI:10.31854/1813-324X-2021-7-4-95-109 (in Russ.)

27. Izrailov K.E. Applying of genetic algorithms to decompile machine code. *Zaŝita informacii. Inside*. 2020;3(93):24–30. (in Russ.)

28. Izrailov K.E., Romanov N.E. Application of genetic algorithm for reverse engineering of machine code. *Proceedings of the XIth International Conference on Infotelecommunications in Science and Education, 15–16 February 2022, St. Petersburg, Russia*. St. Petersburg: The Bonch-Bruevich Saint-Petersburg State University of Telecommunications Publ.; 2022. p. 239–243. (in Russ.)

Статья поступила в редакцию 20.02.2023; одобрена после рецензирования 27.02.2023; принята к публикации 25.03.2023.
The article was submitted 20.02.2023; approved after reviewing 27.02.2023; accepted for publication 25.03.2023.

Информация об авторе:

ИЗРАИЛОВ Константин Евгеньевич

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