

Научная статья

DOI 10.66424/2071-8217-2026-3-5

УДК 004.056

ФОРМАЛЬНАЯ МОДЕЛЬ ИСПОЛНЯЕМОГО КОДА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ НА ОСНОВЕ ГРАФА ИСПОЛНЕНИЯ ДЛЯ РЕШЕНИЯ ЗАДАЧИ БИНАРНОГО МОРФИНГА

Н. В. Гололобов*

Санкт-Петербургский политехнический университет Петра Великого,
Санкт-Петербург, Россия

✉ *gololobov_nv@ibks.spbstu.ru

ДЛЯ ЦИТИРОВАНИЯ

Гололобов Н. В. Формальная модель исполняемого кода программного обеспечения на основе графа исполнения для решения задачи бинарного морфинга // Проблемы информационной безопасности. Компьютерные системы. 2026. № 3. С. 72–83.
DOI: 10.66424/2071-8217-2026-3-5

ПОСТУПИЛА 14.07.2026

ПРИНЯТА 29.07.2026

ОПУБЛИКОВАНА 13.10.2026

© Гололобов Н. В.

Издатель: Санкт-Петербургский политехнический университет Петра Великого

АННОТАЦИЯ

Рассматривается задача построения формальной модели исполняемого кода программного обеспечения, предназначенной для строгого математического описания процессов динамического преобразования (морфинга) программ в процессе их выполнения. Актуальность исследования обусловлена необходимостью разработки математического аппарата, обеспечивающего формальное обоснование методов изменения структуры исполняемого кода без нарушения функциональной корректности программ. Целью работы является разработка архитектурно-независимой формальной модели исполняемого кода, позволяющей определить инвариантные свойства программ при выполнении операций морфинга и установить условия сохранения их наблюдаемого поведения. Предложенная модель объединяет статическую и динамическую составляющие исполняемого кода. Статическая часть описывает программу как совокупность базовых блоков, объединенных графом потока управления с конфигурацией размещения кода в адресном пространстве и множеством управляющих ссылок между блоками. Динамическая составляющая включает формальное описание трассы исполнения программы, последовательности состояний вычислительной системы и наблюдаемого поведения, определяемого исключительно внешне наблюдаемыми результатами выполнения. Практическая значимость исследования заключается в создании единого формального аппарата, пригодного для последующего доказательства корректности алгоритмов морфинга исполняемого кода, разработки методов их верификации и анализа защищенности программных систем. Полученные результаты формируют теоретическую основу дальнейших исследований в области динамической трансформации исполняемого кода и разработки перспективных технологий защиты программного обеспечения.

КЛЮЧЕВЫЕ СЛОВА

Кибербезопасность, модель программного обеспечения, граф исполнения, теория графов, анализ защищенности, преобразование кода

Original article

DOI 10.66424/2071-8217-2026-3-5

A FORMAL MODEL OF EXECUTIVE SOFTWARE CODE BASED ON AN EXECUTION GRAPH FOR SOLVING THE BINARY MORPHING PROBLEM

N. V. Gololobov*

Peter the Great St. Petersburg Polytechnic University, St. Petersburg, Russia

✉ *gololobov_nv@ibks.spbstu.ru

FOR CITATION

Gololobov N. V. A formal model of executive software code based on an execution graph for solving the binary morphing problem. *Problems of information security. Computer systems*. 2026. No. 3, pp. 72–83.
DOI: 10.66424/2071-8217-2026-3-5
(In Russian)

RECEIVED 14.07.2026**ACCEPTED** 29.07.2026**PUBLICATION** 13.10.2026**ABSTRACT**

This paper addresses the problem of developing a formal model of executable software code intended to provide a rigorous mathematical description of runtime program transformation (code morphing) processes. The relevance of the study is determined by the need to develop a mathematical framework that provides a formal justification for methods of modifying the structure of executable code without violating the functional correctness of software. The aim of the study is to develop an architecture-independent formal model of executable code that enables the identification of invariant software properties during code morphing operations and establishes the conditions required to preserve the observable behavior of a program. The proposed model integrates both static and dynamic components of executable code. The static component represents a program as a collection of basic blocks connected by a control flow graph, together with a code placement configuration in the address space and a set of control references between the blocks. The dynamic component provides a formal description of the program execution trace, the sequence of computational system states, and the observable behavior defined exclusively by externally observable execution results. The practical significance of the study lies in the development of a unified formal framework suitable for proving the correctness of executable code morphing algorithms, designing methods for their formal verification, and analyzing the security of software systems. The obtained results establish a theoretical foundation for further research in the field of dynamic executable code transformation and the development of advanced software protection technologies.

KEYWORDS

Cybersecurity, software model, execution graph, graph theory, security analysis, code transformation

1. ВВЕДЕНИЕ

Разработка формальной модели программного обеспечения является необходимым условием для строгого определения операций динамического бинарного морфинга и доказательства их корректности. Создание модели позволит однознач-

но описать структуру исполняемого кода и отношения между его элементами, что заложит основу для оценки функциональной эквивалентности двух образцов кода. На основе построенной модели можно сформулировать доказуемые положения сохранения функциональной эквивалентности при преобразованиях.

Математическая модель программного обеспечения, разрабатываемая в контексте задачи динамического морфинга, должна удовлетворять ряду требований, определяемых как общими принципами построения математических моделей, так и специфическими особенностями решаемой задачи:

1. Полнота модели. Она должна охватывать все структурные элементы исполняемого кода, существенные для корректного описания операций морфинга, а также для анализа механизма ROP-атаки. В рамках исследований к таким элементам отнесены: отдельные машинные инструкции, базовые блоки, функции, управляющие переходы между ними, а также модель состояния вычислительного процесса – значения регистров, содержимое стека и кучи, отображение виртуальных адресов [3].

2. Формальность. Все вводимые понятия должны быть определены строго, без содержательной неоднозначности, допускающей различные интерпретации. Данное требование обусловлено фундаментальностью модели в контексте задачи бинарного морфинга – на ее основе доказываются утверждения о корректности операций преобразования [4].

3. Операциональность. Модель должна обеспечивать возможность формального описания операций преобразования кода и анализа их воздействия на структуру программного обеспечения. При этом описание должно быть применимо в том числе для конкретных, измеримых параметров программного кода.

4. Применимость к бинарному коду. Модель должна быть определена непосредственно на уровне машинных инструкций, без предположения о наличии исходного кода или промежуточного представления. Данное требование позволяет унифицировать применение модели как для анализа собственного кода, так и для анализа методом «черного ящика».

5. Вычислительная обозримость. Сложность операций с объектами модели должна допускать практическую реализацию алгоритмов морфинга в реальном времени. Модели, основанные на полном переборе всех возможных трасс исполнения,

не удовлетворяют данному требованию в общем случае ввиду больших накладных временных расходов [7].

Совокупное выполнение данных требований позволит создать формальную модель, применимую для оценки функциональной эквивалентности исходной программы и модифицированной посредством бинарного морфинга.

2. АНАЛИЗ ИЗВЕСТНЫХ ФОРМАЛЬНЫХ МОДЕЛЕЙ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Формальные математические модели программного обеспечения представляют программу как строго определенный математический объект, допускающий доказуемый анализ свойств, преобразований и поведения. В научных работах они используются для задания семантики программ, формальной спецификации, верификации корректности и анализа вычислительных свойств программных систем.

В научных работах по формальным методам и математическому описанию программ чаще выделяются логические, автоматные, грамматические, семантические, функциональные и информационно-структурные модели [1–17]. Эти модели различаются тем, какой аспект программы принимается за основной: логические свойства, переходы между состояниями, синтаксическая структура, математический смысл вычисления, отображение входов в выходы либо организация потоков данных и зависимостей.

Логическая модель описывает программу через формулы логики предикатов, предусловия, постусловия и инварианты, связывающие входные данные, промежуточные состояния и результаты вычисления [1–4]. В рамках такого подхода программа рассматривается как объект, для которого можно формально доказать выполнение спецификации, если для всех допустимых состояний выполняются заданные логические утверждения. Наиболее характерная область применения логической модели связана с аксиоматической

верификацией и исчислением Хоара, где корректность конструкции программы выражается в виде троек вида *PSQ*. Достоинство данного подхода состоит в высокой строгости и возможности получения доказательства корректности, однако для крупных программных систем он требует значительных трудозатрат на построение спецификаций и инвариантов.

Автоматная модель представляет программу как систему состояний и переходов между ними, инициируемых событиями, условиями или входными воздействиями [5–7]. Такое представление особенно характерно для реактивных систем, встроенного программного обеспечения, коммуникационных протоколов и других объектов, где ключевым является не вычисление значения, а последовательность допустимых состояний поведения. Формальное описание в данном случае задано кортежем состояний, входного алфавита, выходов и функции переходов, что позволяет применять методы анализа достижимости и проверки темпоральных свойств. Основное ограничение автоматной модели связано с проблемой взрыва пространства состояний, которая существенно осложняет анализ сложных распределенных или многопоточных программ.

Грамматическая модель описывает программу или алгоритм через формальные грамматики, регулярные выражения и правила порождения, задающие допустимую структуру конструкций [8–10]. В научных работах этот подход используется не только для синтаксического анализа языков программирования, но и для формального представления логической структуры алгоритмов и схем вычислений. Такая модель удобна в тех случаях, когда требуется исследовать композицию программных конструкций, допустимые шаблоны построения программ и эквивалентность структурных преобразований. Вместе с тем грамматического описания обычно недостаточно для полноценного выражения семантики изменяемых состояний, побочных эффектов и сложного взаимодействия с окружением [9].

Семантическая модель ориентирована на строгое описание смысла программы

и того, как ее конструкции интерпретируются в математических терминах [11, 12]. В научной литературе выделяются операционная, денотационная и аксиоматическая семантики, каждая из которых формализует связь между текстом программы и ее поведением. Операционная семантика задает правила выполнения шагов программы, денотационная сопоставляет программе математический объект или функцию, а аксиоматическая описывает свойства программы через систему логических утверждений [12–14]. Семантические модели образуют теоретическую основу формальной верификации, корректности трансляции и сопоставления разных реализаций одного и того же алгоритма.

Функциональная модель рассматривает программу как математическую функцию, отображающую множество допустимых входов в множество выходов [13–15]. Такой подход особенно естественен для чистых вычислений, в которых результат определяется только аргументами и не зависит от скрытого изменяемого состояния. Научная значимость данной модели состоит в том, что она упрощает доказательство эквивалентности программ, анализ композиции вычислений и применение аппарата теории типов и лямбда-исчисления [13]. Однако при описании императивных, интерактивных и параллельных программ функциональная модель в чистом виде оказывается недостаточной и требует расширения средствами моделирования состояний и эффектов [14].

Информационно-структурная модель описывает программу через структуру зависимостей, потоки данных и отношения между операциями преобразования информации [16, 17]. В работах по математическому моделированию программ данный подход используется для выделения информационной структуры алгоритма как самостоятельного объекта анализа наряду с логической структурой. Практическая значимость этой модели проявляется при исследовании параллельных вычислений, конвейерной обработки и аппаратно-программной реализации алгоритмов. Ее ограничение состоит в том,

что она лучше отражает организацию вычислительного процесса и зависимостей данных, чем поведенческие свойства программы в терминах состояний и событий [17].

С точки зрения полноты математического описания программы перечисленные модели не являются взаимоисключающими, а скорее дополняют друг друга. Логическая и аксиоматическая модели лучше подходят для доказательства корректности, автоматная модель – для анализа поведения во времени, грамматическая – для описания структуры, функциональная – для представления вычисления как отображения, а информационно-структурная – для исследования потоков данных и организации вычислений. Для формального анализа программного обеспечения требуется сочетание нескольких моделей одновременно. Например, синтаксическая структура программы может быть задана грамматически, ее поведение – автоматически, а свойства корректности – логически.

Совокупное использование рассмотренных моделей является избыточным в контексте оценки функциональной эквивалентности при выполнении бинарного морфинга, а использование какой-либо конкретной модели не обеспечивает полноту. Таким образом, наиболее перспективным подходом является создание собственной модели программного обеспечения, ориентированной прежде всего на возможность функционального сравнения двух и более программ без доступа к исходному коду.

3. СТАТИЧЕСКАЯ СТРУКТУРА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Статическая структура программного обеспечения описывает организацию исполняемого кода в виртуальном адресном пространстве процесса безотносительно к конкретной последовательности его исполнения. Данный уровень абстракции модели является основным положением для определения операций морфинга,

ввиду того, что именно расположение базовых блоков и значения управляющих ссылок изменяется в ходе преобразования.

Конфигурацией расположения кода программы P называется отображение $\kappa: V^{JIG} \rightarrow A$, сопоставляющая каждой вершине межпроцедурного графа потока управления $ICFG(P)$ адрес соответствующего базового блока в виртуальном адресном пространстве. Конфигурация κ является биективной: различные базовые блоки располагаются по разным адресам, и ни один адрес не принадлежит одновременно двум блокам [18].

Конфигурация расположения кода κ полностью определяет пространственное размещение всех базовых блоков программы в памяти. При нормальном запуске программы конфигурация κ устанавливается загрузчиком операционной системы и, в отсутствие механизмов динамической модификации, остается неизменной на протяжении всего сеанса исполнения. Таким образом, в случае утечки хотя бы одного адреса, нарушитель получает возможность предсказать конфигурацию κ и осуществить успешную ROP-атаку [19].

Управляющей ссылкой называется пара $(loc, target)$, где $loc \in A$ – адрес местоположения ссылки в виртуальном адресном пространстве (адрес операнда инструкции перехода, ячейки таблицы переходов или ячейки стека, содержащей адрес возврата), $target \in A$ – адрес целевого базового блока. Множество всех управляющих ссылок программы в конфигурации κ обозначается $\Gamma(\kappa)$.

Управляющие ссылки можно разделить на несколько типов в зависимости от механизма их хранения и использования – прямые, косвенные, ссылки через таблицы и стековые ссылки. Прямые ссылки встроены в виде непосредственного операнда инструкции перехода ($JMP\ rel32$, $CALL\ rel32$) и формируют целевой адрес относительно адреса следующей инструкции. Косвенные ссылки хранятся в регистре или ячейке памяти и разыменовываются в момент исполнения инструкции ($JMP\ [rax]$, $CALL\ [rsp+8]$). Ссылки через таблицы переходов используются компиляторами для реализации операторов *switch*

и виртуальных вызовов. Они хранятся в таблицах данных, на которые указывают косвенные инструкции. Стековые ссылки (адреса возврата из функций) помещаются в стек инструкцией *CALL* и извлекаются инструкцией *RET*.

Полное множество управляющих ссылок $\Gamma(\kappa)$ однозначно определяется конфигурацией κ : при изменении адресов базовых блоков все ссылки на них должны быть соответствующим образом обновлены. Это требование является центральным условием корректности операций морфинга.

Таким образом, возможно определить формальную модель программы P как тройку:

$$\mathcal{M}(P) = (ICFG(P), \kappa_0, \Gamma(\kappa_0)),$$

где $ICFG(P)$ – межпроцедурный граф потока управления программы; κ_0 – начальная конфигурация расположения кода, устанавливаемая загрузчиком; $\Gamma(\kappa_0)$ – множество управляющих ссылок в начальной конфигурации.

Созданная формальная модель $\mathcal{M}(P)$ абстрагируется от конкретного бинарного представления инструкций и оперирует исключительно понятиями базовых блоков, переходов и адресов [20]. Это позволяет рассматривать операции морфинга независимо от конкретных реализаций инструкций целевой архитектуры.

4. ДИНАМИКА ИСПОЛНЕНИЯ: ТРАССА ИСПОЛНЕНИЯ И НАБЛЮДАЕМОЕ ПОВЕДЕНИЕ

Динамический уровень формальной модели описывает фактическое исполнение программы как последовательность состояний вычислительного процесса. Данный уровень необходим для формального определения функциональной эквивалентности: две программы эквивалентны в том и только том случае, если их динамические характеристики совпадают.

Пусть дана программа P с формальной моделью $\mathcal{M}(P) = (ICFG(P), \kappa_0, \Gamma(\kappa_0))$. В результате операции морфинга получается

новая конфигурация расположения кода κ_1 , отличная от κ_0 . При этом необходимо определить, при каком условии программа в конфигурации κ_1 функционально эквивалентна программе в конфигурации κ_0 .

Тогда конфигурацию расположения кода κ_0 и κ_1 программы P можно определить как функционально эквивалентные (обозначение $\kappa_0 \approx_P \kappa_1$) тогда и только тогда, когда для любых входных данных $d \in D$, где D – множество допустимых входных данных программы P , выполняется:

$$\Omega(P[\kappa_0], d) = \Omega(P[\kappa_1], d),$$

где $P[\kappa]$ – программа P , исполняемая в конфигурации расположения кода κ ; $\Omega(P[\kappa], d)$ – наблюдаемое поведение программы $P[\kappa]$ при входных данных d . Данное определение формулирует эквивалентность как совпадение наблюдаемого поведения при всех допустимых входных данных. При этом важно, что эквивалентность должна выполняться для всего множества D .

Отношение $\approx_P$ является отношением эквивалентности на множестве всех конфигураций расположения кода программы P : оно рефлексивно ($\kappa \approx_P \kappa$ для любой κ), симметрично (если $\kappa_0 \approx_P \kappa_1$, то $\kappa_1 \approx_P \kappa_0$) и транзитивно (если $\kappa_0 \approx_P \kappa_1$ и $\kappa_1 \approx_P \kappa_2$, то $\kappa_0 \approx_P \kappa_2$). Свойство транзитивности при этом гарантирует, что последовательность морфингов сохраняет эквивалентность при условии, что каждый отдельный шаг ее сохраняет [21].

5. СЛАБАЯ И СИЛЬНАЯ ЭКВИВАЛЕНТНОСТЬ

Теория формальных методов различает несколько уровней эквивалентности программ – слабую и сильную функциональную эквивалентность [21].

Экстраполируя теорию формальных методов, конфигурации κ_0 и κ_1 называются слабо функционально эквивалентными (обозначение: $\kappa_0 \approx_P^w \kappa_1$), если для любых $d \in D$ последовательности наблюдаемых событий $\Omega(P[\kappa_0], d)$ и $\Omega(P[\kappa_1], d)$ совпадают с точностью до значений

адресов: то есть для каждого события $\omega_i \in \Omega(P[\kappa_0], d)$ существует соответствующее событие $\omega'_i \in \Omega(P[\kappa_1], d)$ такое, что они совпадают по типу, порядку и параметрам, не являющимися адресами виртуального адресного пространства.

Аналогично можно определить сильную функциональную эквивалентность в контексте предлагаемой модели. Конфигурации κ_0 и κ_1 называются сильно функционально эквивалентными (обозначение $\kappa_0 \approx_P^s \kappa_1$), если $\Omega(P[\kappa_0], d) = \Omega(P[\kappa_1], d)$ как последовательности, включая все значения параметров событий, в том числе значения адресов.

Различие между слабой и сильной эквивалентностью принципиально для задачи морфинга. Сильная эквивалентность, требующая полного совпадения всех значений параметров наблюдаемых событий, в общем случае не может быть обеспечена операциями морфинга. Если программа передает адрес функции через системный вызов или в качестве данных другому процессу, то изменение адресной конфигурации неизбежно изменит данный адрес, что нарушит сильную эквивалентность. Слабая эквивалентность, наоборот, допускает изменение адресных значений при сохранении всех прочих параметров поведения. Именно слабая эквивалентность является практически достижимым и содержательно корректным критерием для оценки корректности морфинга в общем случае [22].

Для программ, не передающих указатели на код вовне (не использующих адреса функций в качестве данных, передаваемых другим процессам или сохраняемых в файлах), слабая и сильная эквивалентности совпадают в части семантически значимых результатов. Большинство системного и прикладного программного обеспечения удовлетворяет данному условию, что позволяет применять слабую эквивалентность в качестве полноценного критерия корректности морфинга для данного класса программ [9].

Далее, если не уточняется иное, термин «функциональная эквивалентность» используется в смысле слабой функциональной эквивалентности, поскольку именно она является достижимым и операцио-

нально проверяемым критерием для задачи динамического бинарного морфинга.

Трассой исполнения программы P при входных данных d называется последовательность состояний $\tau(P, d) = \langle S_0, S_1, S_2, \dots \rangle$, где S_0 – начальное состояние процесса, определяемое загрузчиком и входными данными d , а каждое последующее состояние S_{i+1} получается из S_i применением операционной семантики инструкции, адресуемой счетчиком команд $PC \in S_i$ [9].

Трасса исполнения $\tau(P, d)$ полностью описывает поведение программы P при входных данных d , включая все промежуточные состояния. Однако для формального определения функциональной эквивалентности полная трасса избыточна: большинство изменений внутреннего состояния процесса – значения промежуточных регистров и организация стека – не наблюдаемы извне и не влияют на функциональный результат программы. Поэтому вводится понятие наблюдаемого поведения, выделяющее из трассы существенную с функциональной точки зрения информацию.

Наблюдаемым поведением программы P при входных данных d называется проекция трассы исполнения $\tau(P, d)$ на множество наблюдаемых событий $\Omega(P, d) = \langle \omega_0, \omega_1, \dots \rangle$, где каждое событие ω_i принадлежит одному из следующих классов: событие ввода-вывода – операция чтения входных данных или записи выходных данных; событие системного вызова – обращение к операционной системе с конкретными параметрами и результатом; событие завершения – нормальное или аварийное завершение процесса с кодом возврата. Все прочие изменения состояния считаются ненаблюдаемыми.

Определение наблюдаемого поведения через упомянутые три класса событий соответствует стандартной концепции наблюдаемости в теории языков программирования и формальной верификации. События ввода-вывода и системных вызовов представляют собой все точки взаимодействия программы с внешней средой, доступные наблюдателю. Любое изменение, не влияющее на эти события, является семантически незначимым с точки зрения

функционального поведения программы. Это утверждение соответствует принципу наблюдаемой эквивалентности в денотационной семантике [23].

6. МОДЕЛЬ ВИРТУАЛЬНОГО АДРЕСНОГО ПРОСТРАНСТВА И УПРАВЛЕНИЕ СТРАНИЦАМИ

Для описания операций морфинга необходимо также определить модель виртуального адресного пространства процесса, ввиду того, что морфинг предполагает динамическое перераспределение страниц памяти между исполняемыми и записываемыми регионами.

Моделью адресного пространства процесса в контексте настоящих исследований принято отображение $\mu: A \rightarrow 2^{\{r,w,x\}}$, сопоставляющая каждому адресу $a \in A$ подмножество разрешенных операций из $\{r,w,x\}$ (чтение, запись, исполнение). При этом страница памяти $pg \subseteq A$ обладает однородными разрешениями: для всех $a \in pg$ выполняется $\mu(a) = \mu(pg)$. Изменение $\mu(pg)$ осуществляется системным вызовом *mprotect()* в ОС семейства Linux или *VirtualProtect()* в ОС семейства Windows [18].

В нормальном режиме исполнения программа соблюдает принцип $W \wedge X$, т.е. страницы кода обладают разрешениями $\{r,x\}$, страницы данных – $\{r,w\}$, страницы стека – $\{r,w\}$. При выполнении операции морфинга система временно нарушает данный принцип для целевой страницы кода, устанавливая $\{r,w\}$ для записи нового кода, после чего восстанавливает $\{r,x\}$. Временной интервал, в течение которого страница одновременно записывается и исполняема, должен быть исключен: данное ограничение формализуется как условие атомарности операции морфинга.

Кодовым регионом программы P в конфигурации κ называется объединение всех страниц памяти, содержащих хотя бы один базовый блок:

$$ICFG(P):CR(\kappa) = \bigcup_{\{B \in V^{III\mathcal{G}}\}} pages(\kappa(B)),$$

где $pages(a)$ обозначает множество страниц, содержащих байты блока B , начиная

с адреса a . Для всех страниц $pg \in CR(\kappa)$ выполняется $x \in \mu(pg)$.

Кодовый регион $CR(\kappa)$ является динамически изменяемым объектом: при морфинге блок B перемещается из одного адреса в другой, что влечет изменение $CR(\kappa)$ – одни страницы выбывают из кодового региона, другие включаются в него. Корректное управление разрешениями страниц при этих изменениях является обязательным условием целостности выполнения [16].

7. ГРАФ ПОТОКА УПРАВЛЕНИЯ КАК ОБЪЕКТ МОРФИНГА И МНОГОПОТОЧНОЕ ИСПОЛНЕНИЕ

Операция морфинга воздействует исключительно на конфигурацию расположения кода κ и соответственно на множество управляющих ссылок $\Gamma(\kappa)$. Топологическая структура $ICFG(P)$ – множество вершин $V^{III\mathcal{G}}$, множество ребер E и разметка ребер типами переходов – при морфинге не изменяется. Семантика каждой инструкции внутри базовых блоков также остается неизменной: морфинг перемещает блоки, но не модифицирует их содержимое (за исключением операндов инструкций переходов, кодирующих управляющие ссылки). Данное разграничение является ключевым для доказательства функциональной эквивалентности [17, 21].

При корректном выполнении операции морфинга следующие элементы формальной модели программы остаются неизменными: топологическая структура $ICFG(P)$, семантика каждой инструкции внутри базовых блоков, наблюдаемое поведение $\Omega(P,d)$ для любых входных данных d . В свою очередь изменениям подвергаются такие элементы, как конфигурация расположения кода κ , множество управляющих ссылок $\Gamma(\kappa)$, кодовый регион $CR(\kappa)$. Любая операция морфинга должна строго удовлетворять данным требованиям для обеспечения функциональной эквивалентности.

Требование неизменности наблюдаемого поведения, в сущности, является более слабым, чем требование совпадения

полных трасс исполнения. В рамках модели допускается, что трассы могут различаться значениями адресов в регистрах и стеке, однако наблюдаемые события – операции ввода-вывода и системные вызовы – должны совпадать.

Одним из важных вопросов является возможность применения модели для программ с количеством потоков $T > 1$. Введенная модель, описывающая однопоточное исполнение, требует расширения для корректного охвата данного класса систем. Хотя полное формальное описание многопоточного исполнения выходит за рамки настоящей статьи, необходимо определить основные принципы расширения.

Многопоточной моделью процесса называется кортеж:

$$M(P) = (T, \mathcal{M}(P), M_shared),$$

где $T = \{t_1, t_2, \dots, t_k\}$ – множество потоков исполнения; $\mathcal{M}(P)$ – разделяемая формальная модель программы (*ICFG* и конфигурация k являются общими для всех потоков); $M_shared \subseteq A$ – разделяемая область памяти, доступная всем потокам. Каждый поток t_i обладает собственным состоянием $S_i(t) = (PC_i, R_i, M_{stack_i})$, где M_{stack_i} – приватный стек потока [21].

В контексте морфинга многопоточность порождает требование синхронизации: операция морфинга, изменяющая конфигурацию k , должна быть согласована со всеми активными потоками. Поток, исполняющий базовый блок в момент его перемещения, должен либо завершить исполнение блока до начала перемещения, либо быть приостановлен на время операции. Данное требование реализуется посредством механизма *stop-the-world*, аналогичного применяемому в системах сборки мусора с перемещением объектов, и формализуется как часть условия атомарности [22].

8. ОГРАНИЧЕНИЯ МОДЕЛИ

Разработанная формальная модель программного обеспечения обладает высокой степенью общности и теоретической строгостью. Тем не менее она опирается на ряд упрощающих предположений,

которые не в полной мере отражают особенности реального программного обеспечения.

Наиболее чувствительным ограничением является предположение о статической и полностью восстанавливаемой структуре *CFG*. На практике точное построение *CFG* бинарного файла является нерешимой задачей в общем случае (согласно теореме Райса), и реальные инструменты бинарного анализа строят аппроксимацию *CFG* с возможными ошибками двух типов: пропуском части блоков или ребер (недоаппроксимация) и включением несуществующих блоков или ребер (переаппроксимация). Строго говоря, переаппроксимация безопасна с точки зрения корректности морфинга. Лишние трассы влияют исключительно на общий объем обновлений, а равно на время выполнения морфинга.

Недоаппроксимация *CFG* является критической для корректности морфинга. В случае, если блок не включен в *ICFG*, его управляющие ссылки не попадают в реестр $\Gamma(k)$. Данная проблема решается посредством консервативного подхода: динамический морфинг применяется только к тем блокам, *CFG*-окружение которых полностью идентифицировано. Блоки с неопределенными управляющими ссылками исключаются из множества кандидатов для морфинга [3].

Другим ограничением модели являются динамически преобразовываемые программы, к которым относится самомодифицирующийся код или механизм динамической загрузки кода.

Самомодифицирующийся код на уровне исполнения нарушает предположение о статической структуре *CFG*, что требует дополнительного механизма мониторинга изменений кода с последующим перестроением затронутых фрагментов *ICFG*. Решение этой задачи выходит за рамки исследований, а сама задача определена как направление дальнейших исследований.

Динамическая загрузка кода порождает новые компоненты *ICFG* в процессе исполнения. Система морфинга должна перехватывать события загрузки библиотек и включать соответствующие блоки в *ICFG* и реестр $\Gamma(k)$ до начала их исполнения. Обеспечить функциональную экви-

валентность при этом не представляется возможным ввиду нарушения принципа замкнутости программного обеспечения.

В рамках модели предполагается независимость и равномерность выбора конфигурации на каждом шаге морфинга. На практике выбор целевых адресов для перемещения ограничен доступными свободными страницами памяти и архитектурными требованиями выравнивания, что нарушает условие строгой равномерности. Для обеспечения корректности применения данной модели предполагается использование криптографически стойкого генератора псевдослучайных чисел для выбора целевых адресов, в которых размещается преобразованный исполняемый код [9].

9. ЗАКЛЮЧЕНИЕ

Построена формальная модель исполнения кода программного обеспечения, включающая: статическую компоненту – формальную модель программы $M(P)$ как размеченный *ICFG* с конфигурацией расположения кода k и множеством управляющих ссылок $\Gamma(k)$; динамическую компоненту – трассу исполнения $\tau(P, d)$ и наблюдаемое поведение $\Omega(P, d)$; модель адресного пространства с управлением разрешениями страниц; разграничение инвариантных и изменяемых элементов

модели при морфинге; принципы расширения модели на многопоточное исполнение.

Формальная модель определена в архитектурно-независимых терминах: понятия базового блока, *CFG*, управляющих ссылок и наблюдаемого поведения применимы к любой архитектуре. Тем не менее конкретные детали реализации – размеры инструкций, кодирование операндов переходов, механизм таблиц переходов – существенно различаются для архитектур *x86-64*, *ARM64* и *RISC-V*.

Таким образом, систематически проанализированы ограничения разработанной модели: неполнота *CFG*-аппроксимации, самомодифицирующийся код и динамическая загрузка кода, семантика передачи адресов через ввод-вывод, временные ограничения реального времени, предположения вероятностной модели и архитектурная специфика реализации. Для каждого ограничения определена его практическая значимость и пути уточнения.

Расширение практической реализации на архитектуру *ARM64* – актуальную для встроенных систем – является перспективным направлением дальнейших исследований. Формальная модель при этом остается неизменной. Адаптации требует только компонент бинарного анализа и операции кодирования/декодирования управляющих ссылок.

КОНФЛИКТ ИНТЕРЕСОВ / CONFLICT OF INTERESTS

Автор заявляет об отсутствии конфликта интересов / The author declare no conflict of interests.

СПИСОК ИСТОЧНИКОВ

1. Chand D. R., Yadav S. B. Logical construction of software // Communications of the ACM. 1980. Vol. 23. № 10. P. 546–555.
2. Kryvyi S., Grinenko O., Opanasenko V. Logical approach to the research of properties of software engineering ecosystem // 2020 IEEE 11th International Conference on Dependable Systems, Services and Technologies (DESSERT). IEEE, 2020. P. 456–464.
3. Wang Y. Software science: on the general mathematical models and formal properties of software // Journal of Advanced Mathematics and Applications. 2014. Vol. 3. № 2. P. 130–147.

4. **Patil L. V., Waghmode M., Joshi Sh. D., Khanna V.** Generic model of software cost estimation: A hybrid approach // 2014 IEEE International Advance Computing Conference (IACC). IEEE, 2014. P. 1379–1384.
5. **Posnett D., Hindle A., Devanbu P.** A simpler model of software readability // Proceedings of the 8th working conference on mining software repositories. 2011. P. 73–82.
6. **Milicev D., Jovanovic Z.** A finite state machine based format model of software pipelined loops with conditions // Progress in Computer Research. 2001. P. 51.
7. **Awedikian R., Yannou B., Lebreton P. et al.** A simulated model of software specifications for automating functional tests design // DS 48: Proceedings DESIGN 2008, the 10th International Design Conference, Dubrovnik, Croatia. 2008. P. 561–570.
8. **Arafteh B. R.** A graph grammar model for concurrent and distributed software specification-in-large // Journal of Systems and Software. 1995. Vol. 31. № 1. P. 7–32.
9. **Davis D.** Modelled on software engineering: Flexible parametric models in the practice of architecture: PhD thesis. RMIT University, 2013.
10. **Wang Y.** On the informatics laws and deductive semantics of software // IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews). 2006. Vol. 36. № 2. P. 161–171.
11. **Heisel M., Santen T., Souquieres J.** Toward a formal model of software components // International Conference on Formal Engineering Methods. Berlin, Heidelberg : Springer Berlin Heidelberg, 2002. P. 57–68.
12. **Brinkkemper S., Pachidi S.** Functional architecture modeling for the software product industry // European Conference on Software Architecture. Berlin, Heidelberg : Springer Berlin Heidelberg, 2010. P. 198–213.
13. **Molan G., Dolinar G., Bojkovski J. et al.** Model for Quantitative Estimation of Functionality Influence on the Final Value of a Software Product // IEEE Access. 2023. Vol. 11. P. 115599–115616.
14. **Awedikian R., Yannou B., Lebreton P. et al.** A simulated model of software specifications for automating functional tests design // DS 48: Proceedings DESIGN 2008, the 10th International Design Conference, Dubrovnik, Croatia. 2008. P. 561–570.
15. **Pham T. L., Dau T. K. T., Nguyen P. B. A.** The structural model of software adoption and organizational performance: Innovation acceptance perspective // Journal of the Knowledge Economy. 2025. P. 1–36.
16. **Wang Baohua, Yixiang Chen, Shun Zhang, Hengyang Wu.** Updating model of software component trustworthiness based on users feedback // IEEE Access. 2019. Vol. 7. P. 60199–60205.
17. **Achermann R.** On memory addressing: A thesis submitted to attain the degree of Doctor of Sciences. ETH Zurich, 2020. 312 p.
18. **Bansal A., Mishra D.** A practical analysis of ROP attacks // arXiv preprint arXiv:2111.03537. 2021.
19. **Гололобов Н. В.** Формальная постановка задачи морфинга исполняемого кода в процессе работы программного обеспечения // Проблемы информационной безопасности. Компьютерные системы. 2024. № S2(60). С. 17–23. DOI: 10.48612/jisp/kg3-84nt-b5z2. EDN: HYURDA.
20. **Simon H. A.** The functional equivalence of problem solving skills // Cognitive psychology. 1975. Vol. 7. № 2. P. 268–288.
21. **Зегжда П. Д., Ростовцев А. Г.** Современные проблемы кибербезопасности // Неделя науки СПбПУ : сборник докладов научного форума с международным участием, 1–6 декабря 2014 г., Санкт-Петербург, Россия. СПб. : Санкт-Петербургский политехнический университет Петра Великого, 2014. С. 262–268. EDN: TWSMIF.
22. **Mosses P. D.** Denotational semantics // Formal Models and Semantics. Elsevier, 1990. P. 575–631.

REFERENCES

1. **Chand D. R., Yadav S. B.** Logical construction of software. *Communications of the ACM*. 1980. Vol. 23. No. 10, pp. 546–555.
2. **Kryvyi S., Grinenko O., Opanasenko V.** Logical approach to the research of properties of software engineering ecosystem. 2020 IEEE 11th International Conference on Dependable Systems, Services and Technologies (DESSERT). IEEE, 2020, pp. 456–464.
3. **Wang Y.** Software science: on the general mathematical models and formal properties of software. *Journal of Advanced Math-*

- ematics and Applications*. 2014. Vol. 3. No. 2, pp. 130–147.
4. **Patil L. V., Waghmode M., Joshi Sh. D., Khanna V.** Generic model of software cost estimation: A hybrid approach. 2014 IEEE International Advance Computing Conference (IACC). IEEE, 2014, pp. 1379–1384.
 5. **Posnett D., Hindle A., Devanbu P.** A simpler model of software readability. Proceedings of the 8th working conference on mining software repositories. 2011, pp. 73–82.
 6. **Milicev D., Jovanovic Z.** A finite state machine based format model of software pipelined loops with conditions. *Progress in Computer Research*. 2001, pp. 51.
 7. **Awedikian R., Yannou B., Lebreton P. et al.** A simulated model of software specifications for automating functional tests design. DS 48: Proceedings DESIGN 2008, the 10th International Design Conference, Dubrovnik, Croatia. 2008, pp. 561–570.
 8. **Arafteh B. R.** A graph grammar model for concurrent and distributed software specification-in-large. *Journal of Systems and Software*. 1995. Vol. 31. No. 1, pp. 7–32.
 9. **Davis D.** Modelled on software engineering: Flexible parametric models in the practice of architecture: PhD thesis. RMIT University, 2013.
 10. **Wang Y.** On the informatics laws and deductive semantics of software. IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews). 2006. Vol. 36. No. 2, pp. 161–171.
 11. **Heisel M., Santen T., Souquieres J.** Toward a formal model of software components. International Conference on Formal Engineering Methods. Berlin, Heidelberg : Springer Berlin Heidelberg, 2002, pp. 57–68.
 12. **Brinkkemper S., Pachidi S.** Functional architecture modeling for the software product industry. European Conference on Software Architecture. Berlin, Heidelberg : Springer Berlin Heidelberg, 2010, pp. 198–213.
 13. **Molan G., Dolinar G., Bojkovski J. et al.** Model for Quantitative Estimation of Functionality Influence on the Final Value of a Software Product. *IEEE Access*. 2023. Vol. 11, pp. 115599–115616.
 14. **Awedikian R., Yannou B., Lebreton P. et al.** A simulated model of software specifications for automating functional tests design. DS 48: Proceedings DESIGN 2008, the 10th International Design Conference, Dubrovnik, Croatia. 2008, pp. 561–570.
 15. **Pham T. L., Dau T. K. T., Nguyen P. B. A.** The structural model of software adoption and organizational performance: Innovation acceptance perspective. *Journal of the Knowledge Economy*. 2025, pp. 1–36.
 16. **Wang Baohua, Yixiang Chen, Shun Zhang, Hengyang Wu.** Updating model of software component trustworthiness based on users feedback. *IEEE Access*. 2019. Vol. 7, pp. 60199–60205.
 17. **Achermann R.** On memory addressing: A thesis submitted to attain the degree of Doctor of Sciences. ETH Zurich, 2020, 312 p.
 18. **Bansal A., Mishra D.** A practical analysis of ROP attacks. *arXiv preprint arXiv:2111.03537*. 2021.
 19. **Gololobov N. V.** Formal statement of the problem of morphizing executive code during the work of software. *Problems of information security. Computer systems*. 2024. No. S2, pp. 17–23. DOI: 10.48612/jisp/kg3-84nt-b5z2. EDN: HYURDA.
 20. **Simon H. A.** The functional equivalence of problem solving skills. *Cognitive psychology*. 1975. Vol. 7. No. 2, pp. 268–288.
 21. **Zegzhda P. D., Rostovtsev A. G.** Modern Problems of Cybersecurity. Nedelya nauki SPbPU : sbornik dokladov nauchnogo foruma s mezhdunarodnym uchastiem, 1–6 dekabrya 2014 g., St. Petersburg, Russia. St. Petersburg : Peter the Great St. Petersburg Polytechnic University, 2014, pp. 262–268. EDN: TWSMIF.
 22. **Mosses P. D.** Denotational semantics. Formal Models and Semantics. Elsevier, 1990, pp. 575–631.

СВЕДЕНИЯ ОБ АВТОРЕ / INFORMATION ABOUT AUTHOR

ГОЛОЛОБОВ Никита Вячеславович – ассистент, Санкт-Петербургский политехнический университет Петра Великого, Россия, 195251, Санкт-Петербург, ул. Политехническая, д. 29
E-mail: gololobov_nv@ibks.spbstu.ru
ORCID: 0000-0003-0623-9891

GOLOLOBOV Nikita V. – Assistant, Peter the Great St. Petersburg Polytechnic University, Russia, 195251, St. Petersburg, Polytechnicheskaya str., 29