МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ,
СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Санкт-Петербургский государственный университет телекоммуникаций
им. проф. М.А. Бонч-Бруевича»

На правах рукописи

Шариков Павел Иванович

РАЗРАБОТКА СТРАТИФИЦИРОВАННЫХ МЕТОДИК СОЗДАНИЯ И ВЛОЖЕНИЯ УСТОЙЧИВОГО К АТАКАМ ДЕКОМПИЛЯЦИЕЙ И ОБФУСКАЦИЕЙ ЦИФРОВОГО ВОДЯНОГО ЗНАКА В БАЙТ-КОД CLASS-ФАЙЛОВ JAVA-ПРИЛОЖЕНИЙ И ИНФОРМАЦИОННЫХ СИСТЕМ

Специальность 2.3.6 – Методы и системы защиты информации,
информационная безопасность

Диссертация на соискание ученой степени
кандидата технических наук

Научный руководитель:
к.т.н., доцент
Красов Андрей Владимирович

Санкт-Петербург – 2023

СОДЕРЖАНИЕ

Введение

Актуальность проблемы. Языки программирования всегда отвечают запросам бизнеса, сообщества, программистов, которые их используют для решения своих задач. В данный момент одним из популярных языков является Java. Данный язык много лет занимал первое место в различных рейтингах, которые отслеживают популярность языка программирования по таким метрикам как размер сообщества, количество кода в хранилищах исходного кода, поисковых запросов в сети Интернет и другим. Java используется в раных областях начиная от кассовых аппаратов и заканчивая сложными финансовыми продуктами. В силу этих и других факторов данный язык программирования крайне популярен не только среди разработчиков, но и как следствие у злоумышленников, которые хотят получить доступ к модулям корпоративных систем и других java-приложений, которые содержат пользовательские данные или ключевую бизнес-логику.

Class-файлы легко анализируются [7, 71]. При должной подготовке, специалисту удастся получить исходный код программы на Java, что позволит использовать его в своих целях. Возможность получить исходный код программы, написанной на Java, является задокументированным и правовым свойством языка и виртуальной машины Java. Отказаться от использования данного языка программирования, значит максимально сократить круг поиска специалистов на проекты, что экономически не выгодно. Также, существует дополнительный ряд факторов, сдерживающих от прекращения использования языка программирования Java. Например, репутация языка у разработчиков, самое большое и профессионально подготовленное сообщество, другими словами, язык Java – экономически целесообразный выбор компаний.

В силу описанных выше причин, программные продукты на языке программирования Java подвергаются пристальному вниманию, как со стороны разработчиков, компаний, так и злоумышленников. Существует потребность в защите java-приложений и информационных, как объектов интеллектуальной

собственности. Также необходимо гарантировать устойчивость к атакам, направленным на целостность программного обеспечения информационных систем, нарушение работоспособности и другим. Различные варианты решения части озвученных проблем уже существуют, однако, в данном исследовании будет рассматриваться вариант применения цифровых водяных знаков. Такие цифровые водяные знаки должны быть незаметны для пользователя, являться частью java-приложения, трудно разрушаемыми и легко идентифицируемыми для юридической защиты прав на java-приложение компанией-разработчиком.

В связи с тем, что в языки программирования постоянно добавляются новые функции, изменяется внутренняя логика уже существующих и производятся другие различные изменения, то в определенный момент времени существующие исследования устаревают. Исходный код скомпилированный на новой версии компилятора может разительно отличаться внутренней логикой взаимодействия с процессором от исходного кода, скомпилированного на старой версии компилятора. Таким образом, исследования связанные с произведением вложения цифрового водяного знака в class-файлы java-приложений на разных конфигурациях, с устойчивостью скрытого вложения, с проверкой целостности исполняемого class-файла и другими атрибутами являются динамическими исследованиями. Язык программирования Java и виртуальную машину Java постоянно развивают и дорабатывают.

Изучением вопросов скрытого вложения цифровых водяных знаков в class-файлы и их защитой занимались многие ученые, в том числе: Красов А.В., J. Palsberg, A. Monden, Seiji Umatani, Krishan Kumar, Hamilton James, Gupta G., Pieprzyk J и др.

В более широком смысле, применимым не только к class-файлам, но и к изображениям, мультимедиа, изучением цифровой стеганографии занимались такие ученые, как: В.И. Коржик, В.А. Яковлев, И.Н. Оков, В.Г. Грибунин, G. Morales-Luna, R.G. van Schyndel, A.Z. Tirkel, C.F. Osborne, Z.H. Wie, P. Qin, Y.Q. Fu.

Степень разработанности темы. Произведен анализ и исследование работ заграничных коллег на данную тематику, либо косвенно относящиеся к ней.

Как, например, исследование J. Palsberg, S. Krishnaswamy, K. Minseok, D. Ma, Q. Shao, Y. Zhang. Experience with Software Watermarking. В данной работе, авторы, исследуют возможность вложения ЦВЗ в структурах данных, которые являются результатом исполнения программы в реальном времени. Для произведения вложения используется простое число, которое может быть представлено в виде графа, постороенного на основе структуры объекта java-приложения. Авторы производят вложение ЦВЗ в данных о изменении размера исходного кода, времени исполнения java-приложения и пространства «кучи». Работа косвенно относится к текущему исследованию. В связи с тем, что в ней предлагается только устаревшая, на текущий момент времени, методика для вложения цифрового водяного знака в программы на Java.

В статье A. Monden, H. Iida, K. Matsumoto, K. Inoue, K. Torii, «A practical method for watermarking Java programs», предлагается методика скрытого вложения ЦВЗ в байт-код class-файла. В данной методике вложение производится за счет замен эквивалентных опкодов байт-кода в скомпилированном class-файле Java. Анализируется байт-код всего class-файла, далее согласно спецификации JVM, эквивалентные опкоды заменяются друг другом, образуя другую последовательность бит. Перебирая данные последовательности бит и изменяя только необходимые опкоды, авторы данной статьи предлагают производить вложение ЦВЗ в байт-код class-файла.

Авторы Seiji Umatani, Tomoharu Ugawa, Masahiro Yasugi в работе «Design and Implementation of a Java Bytecode Manipulation Library for Clojure», исследуют различные программы, которые могут анализировать и изменять байт-код Java. Большинство из них работают с class-файлами так же, как и DOM (document object model) HTML или XML. Они генерируют дерево, которое отражает структуру формата class. Авторы рассказывают про собственную разработку «BcMacro», главным отличием которой является то, что дерево шаблонов является первым классом объекта Clojure.

Работа Minyoung Sung, Soyoung Kim, Sangsoo Park, Naehyuck Chang, Heonshik Shin «Comparative performance evaluation of Java threads for embedded applications:

Linux Thread vs. Green Thread». Авторы производят сравнение скорости работы «green threads» (потоки выполнения, управление которыми вместо операционной системы выполняет виртуальная машина) и потоков операционной системы Linux в Java приложениях. Во время исследования запускаются несколько тестовых задач, которые используют многопоточность, замеряется время синхронизации потоков и время выполнения программ.

Krishan Kumar, Viney Kehar, Prabhpreet Kaur Tulsi провели исследование «An Evaluation of Dynamic Java Bytecode Software Watermarking Algorithms».

Авторы данной работы произвели вложение цифрового водяного знака в 35 class-фалов с помощью динамических алгоритмов анализа кода. Далее проверили реакцию файлов на атаки. Использовали 2 алгоритма: алгоритм Арбойта (используются ранг утверждений (предикатов) в библиотеке, у каждого есть своё значение (число), которое используется как ЦВЗ) и алгоритм Коллберга-Томпсона (используется число, которое представлено в виде графа структуры объекта во время выполнения).

Hamilton James, Sebastian Danicic, «An Evaluation of the Resilience of Static Java Bytecode Watermarks Against Distortive Attacks». Авторы поместили цифровой водяной знак в 60 файлов разными алгоритмами вложения. Провели ряд атак на данные файлы. Использовали 14 различных статичных алгоритма вложения на 3 разных системах. В итоге было применено 37 разных методик атаки изменения программы. Все используемые jar-файлы являются плагинами для текстового редактора «jEdit», так как данные файлы минимального размера, но передают весь спектр java-приложений. Результаты: всего 840 фалов, успешно цифровой водяной знак вложен в 671. Из 671 файла с цифровым водяным знаком, до атаки удалось успешно обнаружить 588 вложения. К каждому файлу применили все методы атак. В сумме вышло 26169 атак.

Gupta G., Pieprzyk J. в своей работе «Source code watermarking based on function dependency oriented sequencing» рассматривают алгоритм вложения цифрового водяного знака в файлы C/C++. Идея алгоритма цифрового водяного знака заключается в том, чтобы модифицировать последовательность так, чтобы

функции этой последовательности были представлены в виде последовательно зашифрованного цифрового водяного знака.

Существует несколько отечественных исследований, которые проверяли быстродействие JVM на разных платформах и операционных системах.

В 2009 году Хашковский В.В., Лутай В.Н. и Юрченко В.В. сравнивали скорость выполнения программ на платформах Windows XP и Linux Open SUSE с помощью JVM. Они использовали возможности JVM для подсчета времени выполнения программы.

В 2015 году исследователи Дидух А.И. и Тищенко В.В. проводили эксперименты на микрокомпьютере Raspberry Pi, где они изменяли частоту процессора и сравнивали скорость выполнения программ на Java.

Сегодня большинство из этих работ устарело в связи со значительными изменениями и оптимизациями внутренних алгоритмов работы Java и JVM. Также, во время исследования некоторых работ были выявлены недостатки методик, значительное снижение эффективности, следствием чего было, также изменение внутренних алгоритмов работы виртуальной машины Java.

Цель диссертационного исследования состоит в том, чтобы повысить объем цифрового водяного знака и его устойчивость к атакам, предложив методики создания и вложения цифрового водяного знака на основе эквивалентных замен операционных кодов байт-кода java.

Научной задачей является разработка методик создания и вложения увеличенного объема и устойчивого к атакам цифрового водяного знака в байт-код class-файлов java-приложений и информационных систем.

Объектом исследования является байт-код исполняемых файлов интерпретируемых языков программирования выполняющихся в виртуальной машине Java.

Предметом исследования является процесс создания и вложения устойчивого к атакам цифрового водяного знака в байт-код class-файлов за счет не декларированных возможностей языка программирования Java.

В данной постановке научная задача формулируется впервые. При этом тематика, научная задача и цель исследования соответствуют паспорту специальности 2.3.6. «Методы и системы защиты информации, информационная безопасность» по пунктам:

п. 7 «Модели и методы формирования комплексов средств противодействия угрозам информационной безопасности для различного вида объектов защиты (систем, цепей поставки) вне зависимости от области их функционирования»;

п. 17 «Методы, модели и средства разработки безопасного программного обеспечения, выявления в нем дефектов безопасности, противодействия скрытым каналам передачи данных и выявления уязвимостей в компьютерных системах и сетях»;

Для достижения поставленной цели необходимо решить следующие задачи:

1. Разработка методики создания и вложения цифрового водяного знака увеличенного объема в байт-код class-файла. Исследование произведенного вложения, средний объем информации возможной к вложению в class-файл от его объема в процентах, сравнение с другими методиками.

2. Разработка методики создания и вложения цифрового водяного знака в class-файлы java-приложения устойчивого к атакам декомпиляцией направленных на его разрушение. Исследование устойчивости произведенного вложения к атакам декомпиляцией.

3. Разработка методики создания и вложения цифрового водяного знака в class-файлы информационной системы устойчивого к атакам обфускацией направленных на его разрушение. Исследование устойчивости произведенного вложения к атакам обфускацией.

Основные результаты, выносимые на защиту

1. Методика создания и скрытого вложения цифрового водяного знака в байт-код class-файла на основе не декларированных возможностей виртуальной машины Java.

2. Методика создания и вложения цифрового водяного знака в class-файлы java-приложения устойчивого к атакам декомпиляцией направленных на его разрушение.

3. Методика создания и вложения цифрового водяного знака в class-файлы информационной системы устойчивого к атакам обфускацией направленных на его разрушение.

Научная новизна результатов. Полученные в работе новые научные и прикладные результаты обладают следующими отличительными признаками новизны.

1) Методика создания и скрытого вложения цифрового водяного знака в байт-код class-файла на основе не декларированных возможностей виртуальной машины Java, отличается тем, что позволяет произвести вложение цифрового водяного знака увеличенного объема в байт-код class-файла за счет использования расширенного набора опкодов байт-кода, пригодных для эквивалентных замен и ранее не используемых в работах по аналогичной тематике, в скомпилированном class-файле не используя перекомпиляцию с добавлением фиктивных методов. За счет этого цифровой водяной знак является неотделимой частью исполняемого файла. Создание цифрового водяного знака происходит на основе анализа структуры байт-кода class-файла, что позволяет значительно повысить невозможность изъятия цифрового водяного знака из class-файла.

Использование разработанной методики позволяет повысить объем цифрового водяного знака в class-файле до 3,8 раза, тем самым обеспечив возможность вложения больше подробной информации о правообладателе исходного кода, возможность вложения нескольких цифровых водяных знаков в один class-файл.

2) Методика создания и вложения цифрового водяного знака в байт-код class-файлов java-приложения, устойчивого к атакам перекомпиляцией, отличается тем, что помимо использования расширенного набора операционных кодов байт-кода Java для эквивалентных замен в class-файле, использует новое внедрение методов и конструкций, затрудняющих декомпиляцию и повторную компиляцию class-

файла, основанных на архитектурных особенностях языка Java и виртуальной машины Java, что ранее не использовалось в работах по аналогичной тематике.

Методика позволяет произвести создание цифрового водяного знака на основе анализа структуры байт-кода class-файлов java-приложения и поиска наименьшего возможного к созданию цифрового водяного знака, на основе анализа всех class-файлов java-приложения, для произведения тиражируемого вложения во все class-файлы java-приложения. Таким образом, возможно произвести тиражируемое вложение цифрового водяного знака в различном объеме во все class-файлы java-приложения.

Цифровой водяной знак устойчив к атакам направленной перекомпиляции, как отдельных class-файлов, так и полноценного java-приложения.

Использование разработанной методики позволяет снизить вероятность успешной декомпиляции class-файла java-приложения для получения исходного кода до 45%, а вероятность разрушения вложенного данной методикой цифрового водяного знака атакой декомпиляцией снизить до 5%.

3) Методика создания и вложения цифрового водяного знака в class-файлы информационной системы устойчивого к атакам обфускацией направленных на его разрушение, отличается тем, что позволяет произвести вложение единого цифрового водяного знака регистратора в байт-код class-файлов информационной системы, за счет анализа взаимосвязей и зависимостей java-приложений и их class-файлов в информационной системе и вложении единого цифрового водяного знака регистратора, разделенного на части по количеству модулей информационной системы, в class-файлы с ключевой логикой java-приложений. Данный подход является новым и впервые используемым для вложения цифрового водяного знака в исполняемые файлы.

Также в данной методике, впервые рассматриваются и используются инструменты обфускации не как способ оптимизации исходного кода class-файла и его защиты от анализа, декомпиляции и кражи логики, а как инструмент для повреждения и уничтожения цифрового водяного знака информационной системы.

Использование разработанной методики позволяет снизить вероятность успешного повреждения или разрушения цифрового водяного знака известными обфускаторами с 90% до 24% и повысить вероятность неработоспособности class-файла или java-приложения части информационной системы после атаки обфускацией с 55% до 80%, тем самым повысив сложность разрушения цифрового водяного знака информационной системы посредством инструментов оптимизации кода.

Также, использование разработанной методики позволяет использовать единый цифровой водяной знак регистратор для двухэтапной проверки целостности информационной системы на уровне java-приложений и на уровне class-файлов java-приложений, что ранее не использовалось в работах по аналогичной тематике.

Новизна научных результатов подтверждается результатами практических экспериментов на реальных объектах, актами о внедрении, публикациями в журналах из перечня ВАК, зарубежных журналах из перечня Scopus, объектами интеллектуальной собственности.

Таким образом, в работе получены новые результаты, которые позволяют разрешить научную задачу, имеющую важное значение для развития теории цифровой стеганографии, применительно к различным информационным системам разработанных на языке программирования Java (соответствует п. 9 «Положения о присуждении ученых степеней»).

Теоретическая значимость результатов работы заключается ее вкладом в развитие теории методик создания и вложения цифрового водяного знака в исполняемые файлы, а именно: в доказательстве возможности использования расширенного набора операционных команд байт-кода для создания и вложения цифрового водяного знака в class-файлы посредством эквивалентных замен; в установлении возможности вложения устойчивого к атакам декомпиляцией цифрового водяного знака в class-файлы java-приложения; в доказательстве устойчивости цифрового водяного знака к атакам декомпиляцией; в установлении возможности вложения устойчивого к атакам обфускацией цифрового водяного

знака в class-файлы информационной системы; в доказательстве устойчивости цифрового водяного знака к атакам.

Практическая значимость работы заключается в следующих результатах:

1. Предложенная методика создания и скрытого вложения цифрового водяного знака в байт-код class-файла на основе не декларированных возможностей виртуальной машины Java заключается в использовании расширенного набора операционных команд виртуальной машины Java для эквивалентных замен в исполняемых class-файлах с целью вложения цифрового водяного знака увеличенного объема равному 0,2-5,7% от общего объема исполняемого class-файла, в отличие от существующих методик позволяющих произвести вложение цифрового водяного знака объема равного 0,2-1,5% от общего объема исполняемого class-файла; методика доведена до реализации алгоритма программы для ЭВМ;

2. Предложенная методика создания и вложения цифрового водяного знака в class-файлы java-приложения устойчивого к атакам декомпиляцией направленных на его разрушение заключается в повышенной устойчивости цифрового водяного знака к атакам декомпиляцией; эффективность методики достигает 95%, что на 80-85% больше по сравнению с существующими методиками;

3. Предложенная методика создания и вложения цифрового водяного знака в class-файлы информационной системы устойчивого к атакам обфускацией направленных на его разрушение заключается в возможности вложения цифрового водяного знака распределенного по частям информационной системы обладающего устойчивостью к атакам обфускацией; вероятность устойчивости цифрового водяного знака к атакам обфускацией достигает 76%, что на 66% больше по сравнению с существующими методиками; методика доведена до реализации алгоритма программы для ЭВМ.

Полученные результаты подтверждаются тестовыми расчетами и проведенными экспериментами. Также, результаты данной работы использовались в НИР «Построение доверенной вычислительной распределенной

среды», заказчик ООО «Ланк» 2018-19 гг., №10200 ИИ28037, кафедры Защищенных Систем Связи под научным руководством А.В. Красова.

Методы исследования. При решении поставленных задач использовались методы цифровой стеганографии, информационной безопасности в программных средах, теории управления, современные методы и средства разработки программного обеспечения, теории графов, теории алгоритмов.

Степень достоверности результатов, выносимых на защиту диссертационного исследования, выводов научного характера подтверждаются обоснованием выбора контейнера для цифрового водяного знака и операционных кодов языка программирования Java для эквивалентных замен, функций языка программирования для затруднения декомпиляции, оценке эффективности декомпиляторов и обфускаторов результатами экспериментальной проверки разработанных методик, оценке эффективности методик посредством метрик исходного кода и поведения java-приложений, анализом работ существующих зарубежных и отечественных практик решения аналогичных задач, апробацией результатов работы на международных и российских конференциях, а также подтверждением о внедрении предложенных методик в организациях и предприятиях.

Публикация результатов исследования. По тематике диссертационного исследования всего опубликовано 24 работы, из них: 2 статьи [153-154] в изданиях (журнал «International Symposium on Intelligent and Distributed Computing. – Springer, Cham», журнал «IOP Conference Series: Materials Science and Engineering. – IOP Publishing»), индексируемых международной базой Scopus; 2 статьи [155-156] в изданиях, индексируемых международной базой Web of Science (сборник докладов конференции «2020 12th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT). – IEEE», сборник докладов конференции «2021 13th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT). – IEEE»); 8 статей [31, 34, 38, 41, 43 45, 46, 47] в рецензируемых изданиях, рекомендованных ВАК для публикаций диссертационных исследований по защищаемой научной

специальности и отрасли наук (журнал «Вестник Санкт-Петербургского университета Государственной противопожарной службы МЧС России» (категория К3), журнал «Вестник Санкт-Петербургского государственного университета технологии и дизайна. Серия 1: Естественные и технические науки» (категория К2), журнал «Т-Comm-Телекоммуникации и Транспорт» (категория К1), журнал «Электронный сетевой политематический журнал «Научные труды КубГТУ»» (категория К2), журнал «I-methods»); журнал «Современная наука: актуальные проблемы теории и практики. Серия: Естественные и Технические Науки» (категория К3); 7 статей опубликованы в сборниках статей и материалах конференций [30, 32, 33, 36, 39, 44]; 1 монография [37], изданная в издательстве «ИП Петрив Роман Богданович». Кроме того, имеются 1 учебно-методическое пособие, 3 свидетельства о государственной регистрации программ на ЭВМ (Приложения А-В), которые, в соответствии с п. 11 «Положения о присуждении ученых степеней», приравниваются к публикациям в рецензируемых изданиях, рекомендованных ВАК.

Работы автора, посвященные тематике диссертационных исследований, широко цитируются научной общественностью, что подтверждает теоретическую и практическую значимость проведенного исследования.

Из вышеуказанных 23 работ 6 работ [30, 32, 33, 43, 45, 46] выполнены единолично, что подтверждает самостоятельность выполнения исследования и личное получение основных результатов работы.

В работах [31, 34, 38, 41, 153-156], выполненных в соавторстве, соискателю принадлежит ведущая роль в постановке задач на исследования, участие в получении научных результатов (методик), а также в их верификации и исследовании. В остальных работах, выполненных в соавторстве, авторский вклад распределен равномерно между всеми соавторами.

В работе предложены методика создания и скрытого вложения увеличенного объема цифрового водяного знака в байт-код class-файла, методика создания и вложения цифрового водяного знака в class-файлы java-приложения устойчивого к атакам декомпиляцией направленных на его разрушение, методика

создания и вложения цифрового водяного знака в class-файлы информационной системы устойчивого к атакам обфускацией, разработаны программы для ЭВМ, реализующие предложенные методики. Перечисленные результаты получены автором лично.

Апробация результатов исследования. Основные положения работы докладывались и обсуждались на международных и республиканских конференциях, в том числе:

1. II всероссийская научно-техническая конференция «Теоретические и прикладные проблемы развития и совершенствования автоматизированных систем управления военного назначения» [35];
2. Международный конгресс (CTN-2016) "Коммуникационные технологии сети" [172];
3. Международная научно-техническая и научно-методическая конференция «Актуальные проблемы инфотелекоммуникаций в науке и образовании» 2018 [36];
4. VI Всероссийская научно-техническая конференция «Проблема комплексного обеспечения информационной безопасности и совершенствование образовательных технологий подготовки специалистов силовых структур» – КОНФИБ [44];
5. Международная научно-техническая и научно-методическая конференция «Актуальные проблемы инфотелекоммуникаций в науке и образовании" 2019 [33];
6. 13th International Symposium on Intelligent Distributed Computing IDC 2019 [153];
7. 2020 12th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT) 5-7 Oct. 2020 [155];
8. The 13th International Congress' workshop on Ultra Modern Telecommunications and Control Systems/ The 7th young researchers international conference on the internet of things and its enablers "Artificial Intelligence in Communication Networks" 25-27 October, 2021 [156];

Подготовлено учебно-методическое пособие «Разработка защищенных приложений» (Россия, Санкт-Петербург, СПбГУТ).

Результаты данной работы использовались в НИР «Построение доверенной вычислительной распределенной среды», заказчик ООО «Ланк» 2018-19 гг., №10200 ИИ28037, кафедры Защищенных Систем Связи под научным руководством А.В. Красова.

Внедрение и реализация результатов исследования. Результаты проведенных исследований нашли практическое применение в разработках, в которых автор принимал личное участие в качестве ответственного исполнителя или исполнителя. Об использовании и внедрении результатов исследования имеется 2 акта о реализации результатов научной работы из 2 организаций (приложение Г, Д):

1) из Санкт-Петербургского государственного университета телекоммуникаций им. проф. М. А. Бонч-Бруевича (СПбГУТ) – результаты работы используются кафедрой защищенные системы связи федерального государственного бюджетного образовательного учреждения высшего образования «Санкт-Петербургский государственный университет телекоммуникаций им. проф. М.А. Бонч-Бруевича» в учебном процессе на старших курсах обучения бакалавров по направлению подготовки 10.03.01 «Информационная безопасность» по дисциплине «Программно-аппаратные средства защиты информации» (рабочая программа дисциплины, регистрационный № 22.05/381-Д), по дисциплине «Основы проектирования защищенных инфокоммуникационных систем», (рабочая программа дисциплины, регистрационный № 22.05/397-Д) при чтении курсов лекций, проведении практических занятий и лабораторных работ;

2) из ООО «ДИЛЕР» – результаты работы использовались при разработке подходов для вложения цифрового водяного знака в исполняемые файлы программного обеспечения, предназначенного для тестирования телекоммуникационного оборудования, с целью защиты исходного кода программного обеспечения, как интеллектуальной собственности в случае компрометации его различных модулей;

Глава 1. Анализ возможности скрытого вложения информации в исполняемые файлы языка программирования Java, требований к цифровым водяным знакам в исполняемых файлах, существующих методик

1.1 Компьютерная стеганография

Стеганография – наука о скрытой передаче информации, как совокупность методов, которые позволяют производить вложение дополнительной информации, секретных данных, в различные сообщения, известна уже на протяжении длительного промежутка времени, но компьютерная стеганография – молодое и успешно развивающееся направление [28]. Основным требованием, предъявляемым к стеганографии, является невозможность определения факта наличия в покрывающем объекте скрытого вложения.

Преимущество стеганографии перед чистой криптографией состоит в том, что сообщения не привлекают к себе внимания. В криптографии мы знаем, что сообщение, которое мы рассматриваем, зашифровано, то есть факт шифрования не скрыт. Стеганография скрывает сам факт наличия скрытого сообщения.

Также, стеганография делится на два класса: «стеганография» (СГ) и «цифровые водяные знаки» (ЦВЗ). Оба класса имеют похожие методы вложения и извлечения стеговложения. Классы отличаются атаками, осуществляемыми на них. Для стеганографии, ключевой атакой будет обнаружение стеговложения в покрывающем контейнере. Для цифровых водяных знаков ключевой атакой является разрушение, повреждение, искажение или удаление ЦВЗ без значительного повреждения основного покрывающего контейнера.

Компьютерная стеганография — направление классической стеганографии, основанное на особенностях компьютерной платформы и использовании специальных свойств компьютерных форматов данных. Представляет собой сокрытие сообщения или какого-либо файла в покрывающем объекте или иначе, контейнере.

Примеры — стеганографическая файловая система «StegFS» для Linux, скрытие данных в неиспользуемых областях форматов файлов, замена символов в названиях файлов, текстовая стеганография.

Также широко используются незадокументированные возможности различного программного обеспечения и форматов файлов. Классическим примером, может являться использование зарезервированных полей компьютерных форматов файлов, которые заполняются нулевыми битами и, соответственно, могут быть использованы для вложения дополнительной информации. Также, существуют компиляторы и инстументы по обфускации, способные производить вложение цифрового водного знака на основании переданного на вход исполняемого файла. Однако такие инструменты чаще всего не могут использоваться в проектах сложнее лабораторных работ, так как возможность поддержки и исправления таких программ под вопросом. Также повышается риск потери работоспособности и существенного увеличения временных затрат на выполнение функций программы.

Цифровой водяной знак (ЦВЗ) – это эффективное средство, обеспечивающее защиту авторских прав и интеллектуальной собственности владельцев цифровой информации. Целью систем ЦВЗ является вложение дополнительной информации в покрывающий объект, которая устойчива к различным атакам нелегитимных пользователей, направленных на удаление этой иформации. Также, факт вложения информации в покрывающий объект, в системах ЦВЗ, не обязательно должен быть скрытым.

Характерной чертой систем цифровых водяных знаков является устойчивость к удалению вложенной информаци [13, 29].

В качестве данных может использоваться абсолютно любая информация: текст, изображения, аудиоданные, сообщение, исполняемые файлы.

На текущий момент сильное развитие получает направление вложения цифровых водяных знаков в изображения различных форматов. Например, решение проблем аутентификации, хранения и контроля целостности всевозможных изображений [9].

Основные положения стеганографии:

- Методы и методики сокрытия стеговложения должны обеспечивать неизменость работоспособности файла, его целостность и объема.
- Предполагается, что криптографу полностью известны возможные стеганографические методы.
- Стеганографическое преобразованием не должно влиять на работу основных свойств передаваемого файла при вложении в него секретного сообщения и неизвестной противнику информации — ключа.
- Если факт сокрытия стеговложения стал известен противнику, то извлечение секретного сообщения должно являться сложной вычислительной задачей.

Также, системы ЦВЗ можно классифицировать по различным параметрам. Таким как, объем вкладываемой информации, критерии секретности, отношение к способам их извлечения [29].

В связи с тем, что в данный момент тренд на глобальную информатизацию, и роль глобальных компьютерных сетей постоянно растет, вместе с этим становится все более важным значении стеганографии. Был проиведен анализ по открытым информационным источникам на предмет использования стеганографии. Анализ показал, что в наибольшей степени стеганография используются для решения ряда проблем в различных сферах.

Стеганографические системы активно используются для решения следующих задач [5]:

- Защита конфиденциальной информации от несанкционированного доступа, кражи, копирования
- Электронная коммерция, защита от несанкционного копирования
- Аутентификация. Обход систем мониторинга и управления сетевыми ресурсами [16]

- Проверка и контроль целостности графических файлов, различных медиа форматов на основе особенностей данных форматов файлов [136]
- Маскирование отдельных частей кода ключевой бизнес логики программного обеспечения
- Защита авторского права на различные виды интеллектуальной собственности
- Скрытое аннотирование электронных документов [12]
- Скрытая связь. Военные приложения.

1.1.1 Цифровая стеганография

Благодаря компьютерным технологиям стеганография получила новый импульс развития, были разработаны новые методы и методики защиты информации, обеспечения безопасной передачи данных по каналам телекоммуникаций. Методы, основанные на надежном сокрытии и незметном внедрении дополнительной битовой информации в цифровые объекты, вызывая при этом некоторые искажения этих объектов – это Цифровая стеганография [10].

Цифровая стеганография включает в себя:

- Произведение вложения информации для ее скрытой передачи по каналам связи
- Скрытое вложение цифровых водяных знаков
- Вложение уникальных, идентификационных цифровых водяных знаков в цифровые покрывающие объекты. Аналогия маркировки товаров уникальными идентификационными номерами, где у каждого товара свой идентификационный номер.
- Скрытое вложение дополнительной информации посредством заголовков. Целью является хранение разнородной информации в едином контейнере.

Стегосистема – это совокупность методов и средств, которые используются при формировании скрытого канала передачи сообщения. Примером широко известных, на текущий момент, стегосистем, могут быть такие как, Outguess, F5 и Model-based при вложении информации в неподвижные изображения [10]. На рисунке 1.1 представлена обобщенная модель стегосистемы.

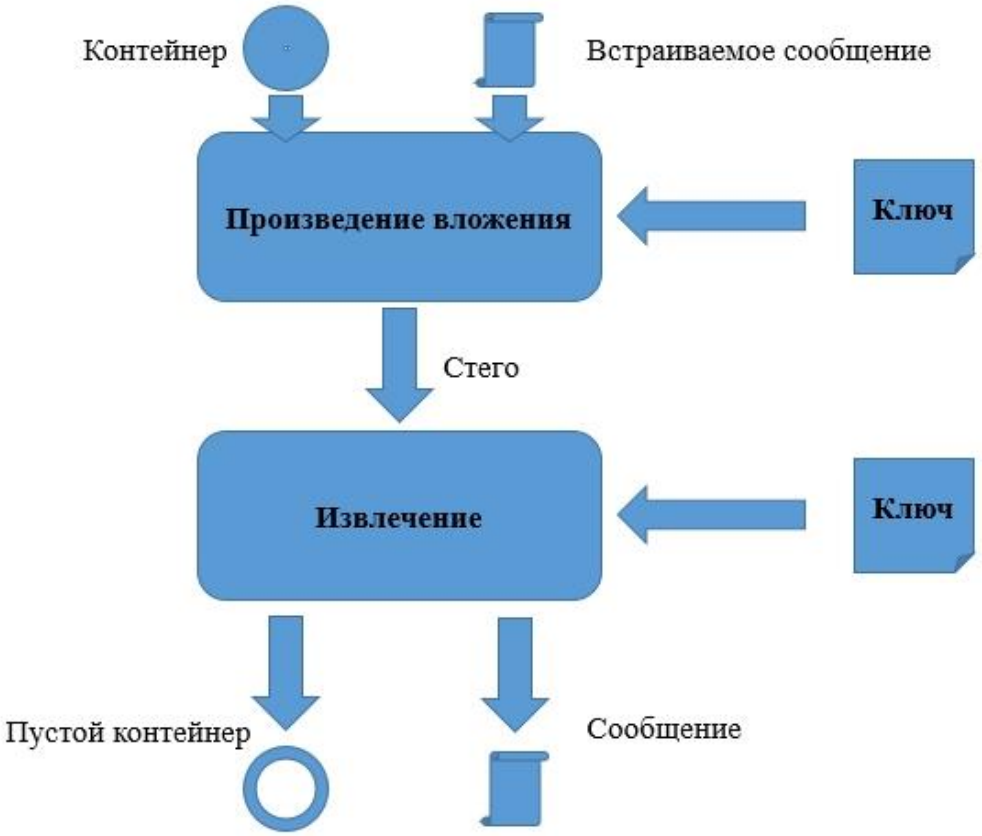


Рисунок 1.1 – Обобщенная модель стегосистемы

В настоящее время компьютерная стеганография развивается по двум основным направлениям:

- Методы и методики, реализованные на использовании специальных, редкоиспользуемых или недокументированных свойств компьютерных форматов
- Методы и методики, основанные на избыточности свойств медиа-форматов

В компьютерной стеганографии используется два основных понятия:

1. Стеговложение – информация, которая предназначена для сокрытия

2. Контейнер – файлы какого-либо формата, предназначенный для хранения стеговложения или скрытого сообщения.

Чаще всего контейнеры подраздляются на два вида: «Пустой» - контейнер, не содержащий в себе никакой другой скрытой информации и «заполненный» контейнер, содержащий скрытое стеговложение [6]. Для стеговложений существует понятие ключа, который подразумевает наличие секретного элемента, определяет порядок занесения секретной информации в контейнер для возможного последующего извлечения или проверки.

В данной работе в качестве покрывающего контейнера для сокрытия секретного сообщения выбран исполняемый class-файл языка программирования Java. Также будет проведено исследование возможности использования специальных свойств данного компьютерного формата.

Есть множество исполняемых файлов, созданных на других языках программирования, но большинство из них, как минимум либо уже рассматривались ранее и не представляют никакого исследовательского интереса, либо не удовлетворяют актуальным критериям эффективности, следовательно, не подходят для данного исследования.

Вдобавок, исполняемые class-файлы кроссплатформенные, что позволяет сделать вывод, о том, что, если использовать class-файл в качестве контейнера, то сообщение, вложенное в него, не деформируется, а работа файла не будет нарушена во всех известных операционных системах, поддерживающих работу виртуальной машины Java. Также не будут заметны изменения в функциональности программы, в ее работе и целостности. Class-файлы имеют небольшой объем, но тот объем информации, который существует возможность вложить в них, достаточен для цифрового водяного знака, который призван однозначно определить юридического владельца программы.

До сих пор нет стойкого и качественного алгоритма для подписи кода. Для аудио, изображений, видео, текстовых файлов уже есть широко используемые и проверенные алгоритмы их подписи, защиты от атак, подтверждения авторства [14,

116, 117, 118]. В случае с программным кодом, до сих пор существуют пробелы в исследовании данного направления.

1.1.2 Необходимость наличия цифрового водяного знака в современных программных разработках

Одна из основных проблем – это «пиратство» или нелегальное копирование/использование, различных программных продуктов, их взлом, а также, кража чужого кода, чтобы впоследствии выдать его как свой собственный. Компании несут огромные убытки, когда их же программный продукт кто-то нелегально копирует, видоизменяет и выпускает на рынок под другим брендом [19].

Как уже говорилось ранее, с юридической защитой других представлений информации, помимо программного кода, на текущий момент, особенных проблем не возникает, существуют отработанные инструменты и алгоритмы [2, 48, 142].

Существует несколько вариантов, когда использование цифрового водяного знака в java-приложениях оправдано:

- Доказательство прав собственности на ПО
- Обнаружение нелегальной программы.

Доказательство прав собственности на программный продукт. При обнаружении ПО у нелегитимного пользователя не всегда быстро возможно доказать свои права на данное ПО. В таком случае, необходимо иметь доказательства прав собственности на ПО. Только при наличии доказательств собственности на ПО заявителям, злоумышленник понесет наказание, предусмотренное статьей 146 УК РФ и обязан будет выплатить штраф за незаконное использование ПО. Для ситуации, которая описана выше подойдет цифровой водяной знак, ранее встроенный в java-приложение. Данный цифровой водяной знак позволит аутентифицировать разработчика, как собственника на интеллектуальную собственность, а именно данное ПО. Более того, если цифровой

водяной знак компании будет обнаружен в файлах стороннего ПО, то это может быть поводом для судебного иска.

Обнаружение нелегальной программы. Java-приложения состоят из исполняемых class-файлов. Если из десктопного приложения скомпрометировали набор class-файлов составляющих ключевую логику и использовали в другом приложении, с другим интерфейсом, реквизитами и дополнительными функциями, то обнаружить нелегальное использование исходного кода будет проблематично. В таком случае разработчики могут не указать в спецификации своего ПО откуда они использовали различные модули логики, особенно, если большая часть логики является краденой. Таким образом, сложность определения скомпрометированных class-файлов достаточно острая проблема для компаний и разработчиков программного обеспечения. Обнаружить нелегальное использование class-файлов может помочь цифровой водяной знак. В таком случае необходимо разработать агента, который будет анализировать java-приложения и по-умолчанию из всех class-файлов java-приложения производить попытку извлечь цифровой водяной знак и расшифровать его. Предполагается, что данный агент будет работать в сети Интернет. При обнаружении цифрового водяного знака, который полностью или частично совпадает с искомым, агент уведомляет разработчика о том, что его интеллектуальная собственность используется незаконно. И хотя данная проблема актуальна, открытых исследований в данной области опубликовано мало.

1.2 Обзор способов отбора наиболее эффективного контейнера для вложения цифрового водяного знака.

Перед тем как производить вложение цифрового водяного знака, требуется выбрать контейнер, который может быть элементом интерфейса программы, конфигурационным файлом, исполняемым файлом или графической частью интерфейса, содержащей другие элементы. Если графическая часть выбрана в качестве контейнера, она обеспечивает возможности управления элементами и отображения элементов пользователю. Различные контейнеры могут быть

использованы для стеганографических методов, позволяющих вставлять секретные сообщения в контейнер, чтобы сообщение стало частью контейнера и было невозможно обнаружить [17]. Однако, внедрение сообщения в контейнер может нарушить его целостность и другие свойства, что может привести к неправильному функционированию контейнера. Например, использование исполняемого файла в качестве контейнера может привести к его неисправности.

На основе предыдущих рассуждений возможно сделать вывод, что надежность и обнаружение скрытого сообщения в стегосистеме зависят от выбранного контейнера [18]. Следующий этап - рассмотрение контейнера, который может быть использован в программном обеспечении. В этом случае контейнер должен содержать различные типы данных, которые связаны между собой внутренней логикой.

Контейнер, с которым не взаимодействовал стегокодер является пустым, контейнер после стегокодера является заполненным. Контейнер может быть фиксированным или потоковым. У контейнеров потокового типа имеется недостаток постоянного изменения данных внутри них, соответственно, может возрастать нестабильность использования их в качестве контейнера ЦВЗ. Контейнеры фиксированного типа имеют произвольный доступ и их размер, содержимое и атрибуты известны заранее [6]. Исследование, проводимое в данном случае, использует фиксированные контейнеры, а именно исполняемые class-файлы Java, которые имеют фиксированную длину.

Произвести вложение в исполняемый файл возможно двумя вариантами:

- Вложение информации в структуру
- Вложение информации в исходный код

При вложении информации в структуру исполняемого файла базисом будут являться спецификации и документация используемого формата файла. Вложение информации может производиться в разделы файла используемые редко или при определенных ситуациях. Могут быть использованы заголовки или отладочные атрибуты, не влияющие на структуру и работоспособность файла [75]. В данном

варианте не производится редактирование исходного кода. Документация к форматам исполняемых файлов чаще всего является общедоступной.

При вложении информации в исходный код структура формата файла остается неизменной, редактируется только логика работы исполняемого файла. Таким образом, контейнером становится исходный код, но не сам файл, содержащий исполняемый код.

Также, при использовании для скрытого вложения информации контейнеров с фиксированной длинной, заранее известны все атрибуты контейнера, а, следовательно, размер исполняемого файла, который позволит произвести расчет доступного объема для вложения скрытой информации. Также файлы с фиксированной длиной позволяют заранее ознакомиться со структурой и определить наиболее эффективные методики вложения информации. Данные файлы являются массовыми и часто используемыми, что делает их идеальным вариантом для использования в качестве контейнеров для скрытой информации.

Для форматов исполняемых файлов, которые используются в качестве контейнеров существует ряд критериев. Если исполняемый файл является часть программного обеспечения с интерфейсом, то изменения, которым подвергся контейнер в результате скрытого вложения информации, не должны быть заметны пользователю данного ПО. Также не должен существенно изменяться занимаемый размер данным файлом. Одним из критичных требований является сохранение целостности и функционала исполняемого файла или программы [88, 163].

1.2.1 Способы отбора контейнера с применением методов стеганографии

Способ выборки контейнера может осуществляться методами конструирующей, селективной или суррогатной стеганографии [5].

Суррогатная стеганография предполагает замену битов контейнера на биты сообщение, вложение которого производится. При таком подходе объем доступных к ложению данных крайне мал. Селективная стеганография включает генерацию множества контейнеров для последующего выбора наиболее

подходящего. Конструирующая стеганография подразумевает более детальный отбор скрываемого сообщения для минимального изменения модели первоначального шума контейнера. Контейнеры могут быть систематическими или несистематическими и иметь различное форматирование данных [5]. Объем встраиваемых данных является важным фактором при выборе метода скрытия сообщений в файлах цифрового формата.

Практически для всех методов, которые применяются для скрытого вложения информации, применима зависимость объема ЦВЗ к объему контейра. Данная зависимость представлена на рисунке 1.2.

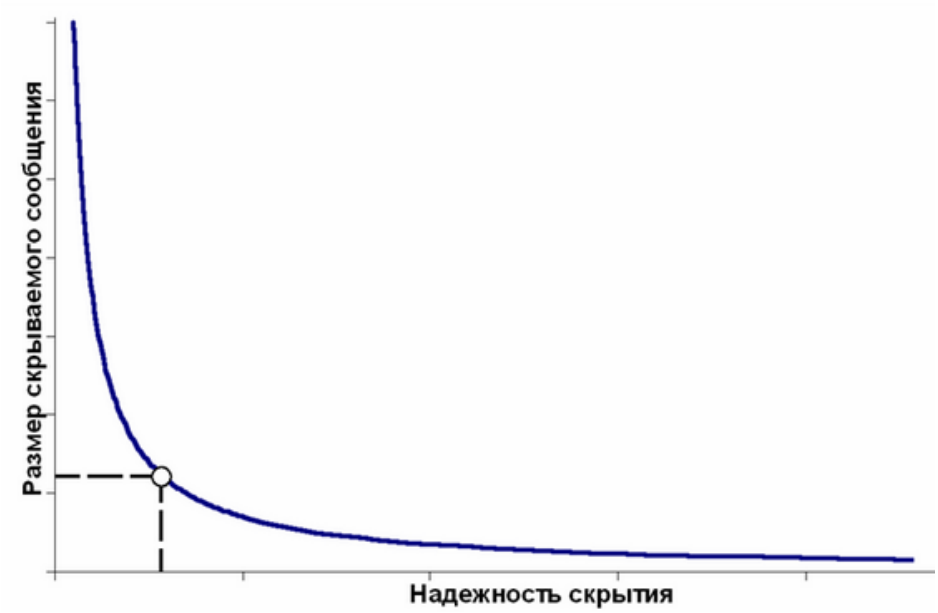


Рисунок 1.2 – Зависимость надежности скрытия от размера сообщения

Увеличивая объем информации к вложению, снижается надежность сокрытия данной информации и, возможно, качество контейнера. В форматах исполняемых файлов увеличение объема информации к вложению неминуемо ведет к ухудшению производительности исполняемого файла.

1.3 Основные свойства языка программирования Java и его окружения

1.3.1 Анализ принципов работы Java и Java Virtual Machine

Программа разрабатывается на языке Java. Java – высокоуровневый язык. Таким образом, программа сначала пишется на высокоуровневом языке, а потом компилируется Java Virtual Machine в java байт-код [127].

Байт-кодом являются операционные коды виртуальной машины Java, в которые был интерпретирован компилятором Java исходный код программы. Данные операционные коды являются инструкциями, которые обрабатываются виртуальной машиной Java и передаются на исоплнение. Один опкод равен 1 байту. В одном байте может быть 1 значение, всего 256 возможных значений. Среди 256 возможных типов операций, используются только 205, так как 51 операция зарезервирована для будущих доработое JVM. Байт-код – это фактически машинный код, но понимает его только Java Virtual Machine. Именно JVM исполняет написанный программистом код на любой платформе, для которой разработана виртуальная машина Java [127].

В текущий момент развития JVM такой «промежуточный» подход к исполнению программ практически никаким образом не сказывается на скорости работы программы. Последние версии виртуальной машины Java высоко оптимизированы, таким образом, что части байт-кода «на лету» т.е. в режиме реального времени, компилируются в родные инструкции процессора. На рисунке 1.3 представлен принцип работы JVM.

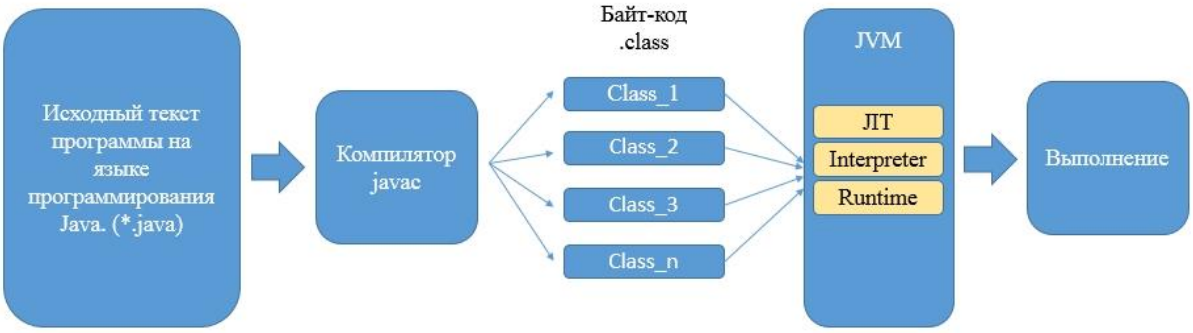


Рисунок 1.3 – Принцип работы JVM

Компиляцию исходного кода в байт-код осуществляет компилятор Java `javac`. Данный компилятор является частью JDK. Компилятор осуществляет компиляцию файлов с расширением `.java`, которые содержат в себе исходный код, а после компиляции создает файл с расширением `.class`, который содержит в себе байт-код, необходимый для исполнения программы виртуальной машиной [127].

Байт-код стеко-ориентированный язык, похожий по своей структуре на ассемблер. Поскольку виртуальная машина не имеет регистров для хранения abitrary значений, все должно быть в стеке, прежде чем может быть использовано в расчетах.

Основные операции в байт-коде работают со стеком. Для выполнения операций необходимо сначала положить данные на стек. Номера переменных используются вместо их имен. В стеке лежат параметры вызываемых методов, объекты для вызова методов и остальные переменные. Перед выполнением условных операторов и арифметических действий с двумя переменными, сравниваемые значения также помещаются в стек. Для каждого типа есть свои операторы сравнения. Результат выполнения операций заносится в стек в порядке LIFO (последний пришел, первый ушел) [27].

После запуска программы в виде файла .java происходит следующий процесс:

1. Компиляция исходного кода программы компилятором javac. Происходит интепретация исходного кода в байт-код JVM.

2. Производится циклический анализ байт-кода и каждого его опкода в JVM. Производятся внутренние проверки верификатора и загрузчика классов, после чего class-файл исполянется построчно.

3. Также, запускается механизм под названием JIT-компиляция (Just-In-Time). Который анализирует байт-код и при наличии в нем фрагментов, которые используются в программе несколько раз, один раз скомпилирует их в машинный

код и в последствии просто будет подставлять в места их вызова. Естественно эта функция в разы ускоряет выполнение программы.

1.3.2 Анализ структуры class-файла

Разбор структуры байт-кода невозможен без понимания структуры class-файла. Class-файл, представляет собой, последовательность байт, состоящих из 8 бит. Типы u1, u2, u4 – беззнаковые типы, размеры которых имеют значения 1,2 и 4 байта соответственно. Сам class-файл представляет скомпилированный исходный код Java. Как было показано ранее, код хранится и пишется в файлах формата *.java, далее, компилятор Java компилирует эти файлы в файлы формата *.class. Виртуальная машина Java умеет работать с форматом файлов *.class, соответственно, получая на вход скомпилированные файлы она начинает исполнять их преобразуя команды, хранящиеся в class-файлах в машинные команды родного процессора, на котором запускаются class-файлы [27].

Class-файл состоит из:

- Заголовка, который идентифицируется всегда по первым байтам CAFEBABE и содержит версию формата файла.
- Константного пула (constant pool), который содержит в себе числа, строки, имена классов, полей, методов.
- Объявление класса (модификаторы, имя класса, имя суперкласса, имена реализуемых интерфейсов)
- Поля класса
- Методы класса
- Атрибуты класса (аннотации, debug info)

Структура class-файла представлена в листинге 1.1.

Листинг 1.1 – Структура class-файла

```
ClassFile {
        u4 magic;
        u2 minor_version;
        u2 major_version;
        u2 constant_pool_count;
        cp_info constant_pool[constant_pool_count-1];
        u2 access_flags;
        u2 this_class;
        u2 super_class;
        u2 interfaces_count;
        u2 interfaces[interfaces_count];
        u2 fields_count;
        field_info fields[fields_count];
        u2 methods_count;
        method_info methods[methods_count];
        u2 attributes_count;
        attribute_info attributes[attributes_count];
}
```

1. **Magic.** В данном элементе хранится информация содержащая номер, который определяет формат class-файла. Данный номер имеет значение 0xCAFEBABE. Всегда.

2. **minor_version и major_version.** Младшая часть номера версии (minor_version) и старшая часть номера версии (major_version). Данные значения совместно определяют версию class-файла. Например, если class-файл имеет

старшую часть номера версии равную 5 и младшую часть номера версии равную 2, то общая версия class-файла будет обозначаться 5.2.

3. **constant_pool_count.** Значение количества константных пулов на единицу больше количества элементов константного пула в таблице.

4. **constant_pool[constant_pool_count-1].** Таблица константных пулов содержит структуры, которые представляют строковые константы, имена классов, интерфейсов, полей и другие константы, на которые ссылаются в структуре Class-файла и ее подструктурах. Таблица индексируется от 1 до значения константных пулов – 1.

5. **access_flags.** Флаги доступа. Значение данного элемента – это маска флагов, которая определяет уровень доступа к данному классу или интерфейсу, а также его свойства. Значения установленных флагов приведены в таблице 1.1.

6. **this_class.** Содержит ссылку на текущий класс. Для данного элемента требуется использовать действительный индекс из таблицы константных пулов.

7. **super_class.** Родительский класс также называется "предком". Элемент, соответствующий этому свойству, может принимать значение либо ноль, либо действительный индекс из таблицы константных пулов. Если значение элемента отлично от нуля, то соответствующий элемент таблицы константных пулов должен являться структурой CONSTANT_Class_info, содержащей информацию о непосредственном предке данного класса, который определен в текущем Class-файле. Если значение равно нулю, то данная Class-файл описывает класс Object, который является общим предком для всех классов.

8. **interfaces_count.** Количество интерфейсов или счетчик интерфейсов. Данное значение дает представление о количестве интерфейсов предков данного класса или интерфейса.

9. **interfaces[interfaces_count].** Массив интерфейсов. Каждый элемент в массиве interfaces[] представляет собой действительный индекс в таблице constant_pool[].

10. **fields_count.** Количество полей является элементом, который определяет число элементов field_info в таблице field[]. Структура field_info

описывает все поля, которые объявлены в этом классе или интерфейсе, включая поля, которые принадлежат экземпляру, а также те, которые принадлежат классу. Этот элемент также может быть именован как "счетчик полей".

11. **fields[fields_count].** Массив полей. Каждая структура в таблице представляет значение типа field_info, предоставляющее полную информацию о поле в классе или интерфейсе. Таблица field[] содержит поля, которые были объявлены в данном классе или интерфейсе.

12. **methods_count.** Количество методов или счетчик методов. Содержит число элементов method_info в таблице methods[].

13. **methods[methods_count].** Каждый элемент в таблице methods[] представляет собой структуру method_info, дающую полное описание метода в классе или интерфейсе. Структура method_info содержит в себе все методы, объявленные в классе или интерфейсе, а также методы экземпляра, методы класса.

14. **attributes_count.** Соответствует числу элементов в массиве atributes[].

15. **attributes[attributes_count].** Массив атрибутов. Каждый элемент таблицы представляет собой структуру attribute_info.

Таблица 1.1 – Значения установленных флагов

Имя флага	Значение	Пояснение
ACC_PUBLIC	0x0001	Объявлен, как public; доступен из других пакетов.
ACC_FINAL	0x0010	Объявлен, как final; наследование от класса запрещено.
ACC_SUPER	0x0020	При исполнении инструкции invokespecial обрабатывать методы класса предка особым способом
ACC_INTERFACE	0x0200	Class-файл определяет интерфейс, а не класс.
ACC_ABSTRACT	0x0400	Объявлен как abstract; создание экземпляров запрещено.

ACC_SYNTHETIC	0x1000	Служебный класс. Отсутствует в исходном коде.
ACC_ANNOTATION	0x2000	Объявлен, как аннотация
ACC_ENUM	0x4000	Объявлен как перечисление (тип enum)

Дескрипторы поля.

Дескриптор поля описывает типы классов, экземпляров или локальных переменных. Представляет собой последовательность символов, удовлетворяющих правилам грамматики. В таблице 1.2 представлено описание символов типов полей.

Таблица 1.2 – Описание символов типов полей

Символ базового типа	Полное название типа	Описание
B	byte	Знаковый байт
C	char	Символ Unicode в кодировке UTF-16
D	double	Значение с плавающей точкой двойной точности
F	float	Значение с плавающей точкой одинарной точности
I	int	Целое значение
J	long	Длинное целое
LClassname;	reference	Экземпляр класса *Classname*
S	short	Короткое целое
Z	Boolean	true или false
[	reference	Одно измерение массива

Дескрипторы методов.

Дескриптор метода класса или интерфейса действителен только, если общее число входных параметров метода (с учётом неявного параметра this) меньше либо равно 255. Если тип параметра не является long или double, каждая встречающаяся

переменная увеличивает счетчик параметров на один. В случае, если тип параметра является long или double, количество параметров увеличивается на два.

Таким образом дескриптором метода:

```
Object mymethod(int i, double d, Thread t)
```

Будет:

```
(IDLjava/lang/Thread;)Ljava/lang/Object;
```

1.3.3 Анализ структуры байт-кода Java

Когда производится запуск скомпилированных class-файлов, виртуальная машина Java, считывает, находящиеся в class-файле, байт-коды построчно и исполняет их, интерпретируя в операционные коды родной системы, внутри которой производится запуск class-файлов [121].

Каждая инструкция имеет вид:

<индекс> <код> [[<операнд 1>][<операнд 2>]][<комментарий>]

<индекс> - индекс операции в массиве байт-кодов соответствующего метода.

<код> - мнемоническое обозначение кода инструкции

<операнд N> - операнды инструкции, которые могут отсутствовать.

[<комментарий>] – необязательный комментарий в конце строки.

Локальная переменная иметь тип: boolean, byte, char, long, short, int, float, double, reference, returnAddress [128].

Когда создается новая переменная, то стек операндов используется для хранения значения новой переменной. Затем значение новой переменной сохраняется в массиве локальных переменных в соответствующий слот. Если переменная не является элементарным значением, то слот хранит только ссылку. В ссылке указывается адрес на объект, хранящийся в «куче».

Пример инструкции:

```
12 bipush 69     //записать в стек константу 69 с типом int.
```

Что происходит в памяти, когда выполняется данная команда, показана на рисунке 1.4 ниже.

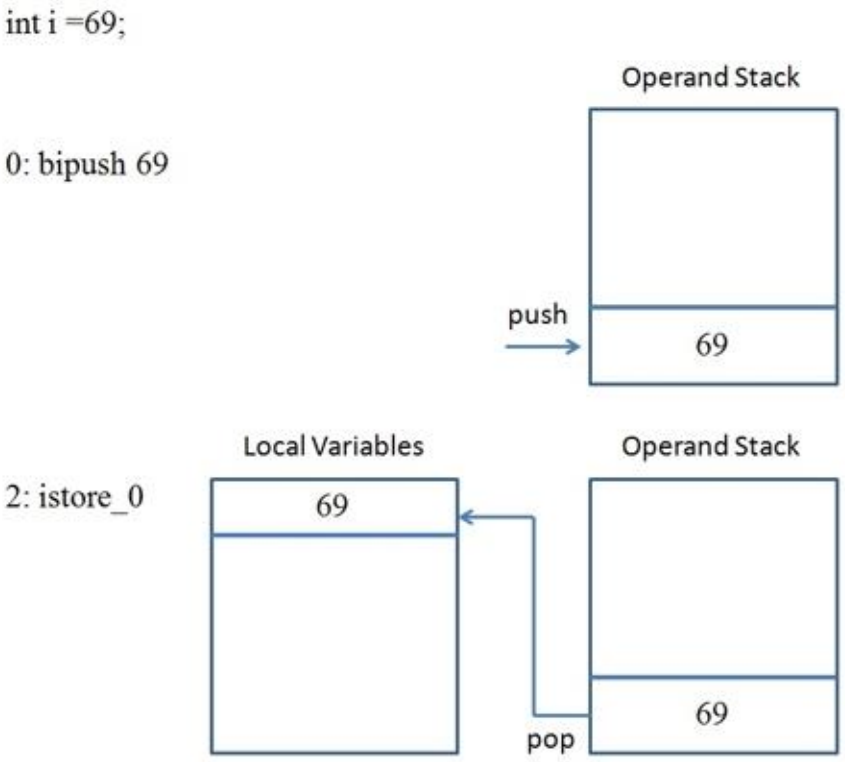


Рисунок 1.4 – Пример работы стека при выполнении операции «bipush»

Операнды, представляющие индексы значений в пуле констант, снабжаются символом решетки и комментарием, описывающим значение константы, на которую ссылается индекс.

1.3.4 Анализ исполнения байт-кода class-файла в виртуальной машине Java

Допустим, что у нас есть условный оператор, который принимает один параметр, являющийся меткой перехода. Прежде чем вызывать этот оператор, необходимо поместить сравниваемые значения на стек. Для каждого типа данных существует свой оператор сравнения. Например, для типа int используется IF_ICMPGE. Если мы хотим сравнить значение с нулем, то нужно использовать отдельную команду. После выполнения сравнения, сравниваемые значения удаляются со стека. Если мы выполняем арифметические операции над двумя

переменными, то также нужно поместить эти переменные на стек. После выполнения операции результат заменит эти переменные в стеке. Результат может быть помещен на место первой переменной в стеке [140].

Операции получают данные из стека, обрабатывают их и заносят в стек результат по правилу LIFO (последний пришел, первый ушел). В таблице 1.3 показан пример, как вычисляется выражение **a*4+b** на виртуальной машине.

Таблица 1.3 – Пример исполнения выражения

Байт код	Команда	Операция со стеком
0x1b	iload_1	Поместить в стек переменную **a**
0x05	iconst_2	Поместить в стек число **4**
0x68	imul	Извлечь из стека число **4** и переменную **a**, произвести умножение и поместить его результат в стек
0x1c	iload_2	Поместить в стек переменную **b**
0x60	iadd	Извлечь переменную **b** и результат умножения, произвести сложение и поместить его результат в стек

В массиве локальных переменных хранятся:

- различные параметры метода
- значения переменных.

Размер массива зависит от количества переменных. Операционные коды байт-кода работают со стеком, создают, забирают, отправляют туда значения. Стек используется для получения значений, возвращаемых методом return. У каждого метода свой массив байт-кодов.

Вирутальная машина Java хранит фреймы в стеке. Фрейм создается с каждым новым вызовом метода и состоит из стека операндов, массива локальных переменных, а также ссылок на время выполнения постоянного пула класса текущего метода. В каждый момент работы программы, скорее всего, будет создано много кадров и соответственно стеков операндов для текущего

управляющего потока. Количество кадров соответствует глубине вложенности вызовов методов. Активным является стек операндов только текущего метода.

Набор инструкций виртуальной машины Java спроектирован таким образом, что позволяет отличать типы операндов той или иной инструкции по разному байт-коду инструкции. Если метод работает только с типами значений int, то все инструкции скомпилированного байт-кода (`iconst_0, istore_1, iinc, iload_1, if_icmplt`) также оперируют только с типами данных int [27].

1.3.5 Обзор наиболее используемых операционных кодов виртуальной машины Java

Далее рассмотрим наиболее важные и частые инструкции из списка инструкций виртуальной машины Java.

aaload - Данная инструкция загружает ссылку из массива.

aload - Загружает ссылку из локальной переменной в стек.

astore - Считывает ссылку из стека и сохраняет в локальную переменную.

bipush - Записывает значение типа byte в стек.

goto - Безусловный переход. Смещение выполняется относительно индекса инструкции goto, управление передается команде находящейся по рассчитанному смещению.

iadd - Складывает два значения типа int

iaload - Записывает в стек значение int из массива.

iconst_ - Записывает в стек константу типа int.

if_acmp<cond> - Переход, если значение типа reference равны.

if_icmp<cond> - Переход, если значения типа int удовлетворяют условию.

iinc - Увеличивает локальную переменную на заданную константу.

iload - Загружает значение типа int из локальной переменной в стек.

imul - Умножает два значения типа int.

istore - Считывает из стека значения типа int и записывает в локальную переменную.

Инструкции делятся на несколько групп:

- Загрузка и сохранение (aload_0, istore)
- Арифметические и логические операции (iadd, fcmpl)
- Преобразование типов (I2B, D2I)
- Создание и преобразование объекта (new, putfield)
- Управление стеком (dup, pop)
- Операции перехода (goto, ifeq)
- Вызовы методов и возврат (invokestatic, ireturn)

Большинство инструкций передачи управления языка Java (if-then-else, do, while, break и continue) компилируются в байт-код тривиальным образом.

Пример:

```
void whileIntMyTest() {
        int iz2 = 0;
        while (iz2<34){
        iz2++;
        }
}
```

Будет скомпилировано так:

```
Метод void whileInt()
0   iconst_0
1   istore_1
2   goto 8
5   iinc 11
8   iload_1
9   bipush 34
11 if_icmplt 5
14 return
```

Проверка цикла while (реализуемая с помощью инструкции if_icmplt) расположена в конце цикла в компилированном байт-коде. Почему таким образом работает виртуальная машина Java, поможет разобраться следующий пример.

Допустим, нам необходимо получить ответ для данного выражения:

2+3*4

JVM будет работать несколько иначе, чем мы привыкли решать подобные выражения. Виртуальная машина Java будет работать следующим образом:

1. 2
2. 3
3. 4
4. *
5. +

Таким образом, скомпилированный код будет выглядеть следующим образом:

0 iconst_2

1 iconst_3

2 iconst_4

3 imul // Инструкция берет два верхних значения из стека и производит их перемножение. Результат также кладется на стек.

4 iadd // Инструкция складывает два верхних (2 и 12) значения со стека и результат кладет обратно на стек.

Числа, которые предшествуют каждой инструкции байт-кода указывают позицию байта. Например, данная инструкция занимает только один байт в длину, так как нет операндов: `iconst_1`. Раз у предыдущей инструкции нет операндов, значит, следующая будет в номере 2. Инструкции такие как: `bipush 5`. Будут занимать два байта. Один байт для опкода bipush и один байт для операнда 5. В данном случае следующая инструкция будет на номере 3.

1.4 Безопасность языка программирования и виртуальной машины Java

1.4.1 Анализ загрузчиков class-файлов в виртуальную машину Java

Перед тем как выполнить написанный программистом код, виртуальная машина Java производит некоторые проверки этого кода, с помощью внутренних служебных классов. Скомпилированный class-файл необходимо загрузить в виртуальную машину Java, таким образом, чтобы не была утеряны зависимости текущего class-файла с другими, описанными внутри текущего. Следовательно, зависимые class-файлы также необходимо загружать в виртуальную машину Java, но только в тот момент, когда они действительно понадобятся. Таким образом, виртуальная машина распределяет ресурсы, загружая, проверяя и запуская только тот код и классы, которые нужны в текущий момент [140].

Загружают скомпилированные class-файлы в виртуальную машину Java – загрузчики классов. В виртуальной машине они представлены иерархией и имеют зависимость «родитель – потомок».

После того, как компилятор произвел компиляцию исходных операторов языка Java в файл с расширением .class, понятный виртуальной машине Java, происходит интерпретация байт-кода в машинные команды целевой платформы. Действия, выполняемые виртуальной машиной Java описаны ниже:

1. С интернета или с жесткого диска, загружается class-файл.
2. Если в class-файле присутствуют ссылки на классы других типов, помимо базовых, то дополнительно происходит загрузка файлы этих классов.
3. Далее, виртуальная машина Java, выполняет метод main() в главном классе.
4. Если для самого метода main() или для методов, которые находятся в нем, нужны дополнительные классы, то они также загружаются.

Каждой запускаемой программе на Java сопутствуют несколько загрузчиков классов. Представление иерархии загрузчиков классов показано на рисунке 1.5.

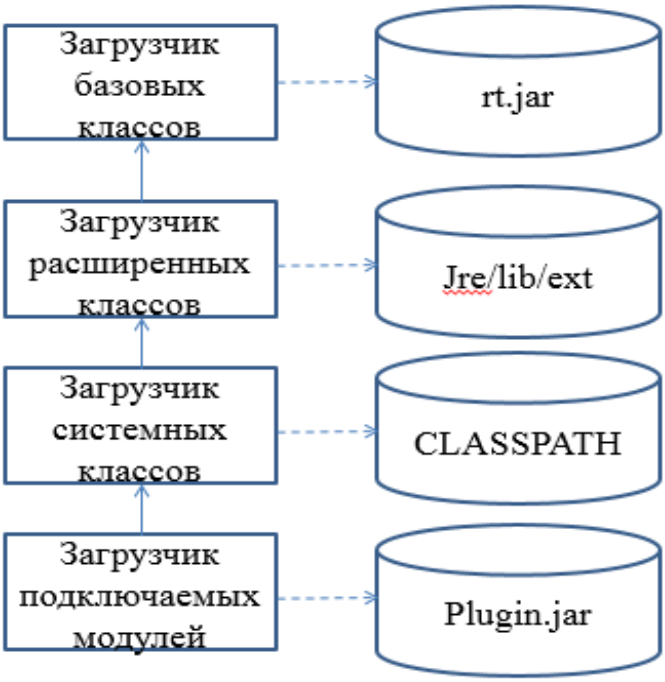


Рисунок 1.5 – Иерархия загрузчиков классов.

Крайне важно производить контроль в исходном коде, чтобы все зависимые классы загружались правильными загрузчиками, для устранения явления «инверсии загрузчиков классов». Чтобы такого не происходило, необходимо указывать нужный загрузчик классов в виде параметра, или устанавливать загрузчик классов загрузчиком по умолчанию в текущем потоке [82].

Таким образом, необходимо осуществлять проверку, что необходимый метод, находящийся в другом class-файле, будет вызван из основного class-файла, посредством правильного загрузчика классов. В рамках данного исследования, загрузчикам классов уделяется не много внимания, так как это ключевой механизм безопасности Java и его не правильное или не задокументированное использование может привести к частичной или полной неработоспособности программы, что противоречит критериям успешности скрытого вложения цифрового водяного знака в исполняемый файл.

1.4.2 Верификация байт-кода

После того, как загрузчик классов успешно загрузил необходимый class-файл, он передает виртуальной машине Java его байт-код. Виртуальная машина, перед тем, как начать исполнять полученный байт-код от загрузчика классов будет осуществлять его проверку. Таким образом, происходит запуск верификатора.

Суть работы верификатора в том, чтобы проверить, какие команды в полученном байт-коде могут нанести вред виртуальной машине Java. Верификатор содержит в себе большое количество проверок, позволяющих ему однозначно определить считать класс поврежденным или нет. Ниже представлены некоторые из видов проверок верификатора:

1. Отсутствие переполнение стека
2. Доступ к локальным переменным в стеке во время выполнения
3. Соблюдение правил доступа к закрытым данным и методам
4. Согласование типов ссылок при вызове метода
5. Инициализация переменных перед их использованием

Данный этап защиты виртуальной машины Java разработан в связи с тем, что даже при условии того, что любой class-файл сформированный компилятором Java проходит верификацию, формат байт-кода, представленный в этих файлах хорошо задокументирован. Таким образом, программист владеющий знанием ассемблера, теоретически может создать class-файл с достоверными, но опасными командами для виртуальной машины Java. В следствии этого, функция верификатора заключается больше в том, чтобы блокировать злонамеренно измененные файлы классов, кроме проверки class-файлов, созданных компилятором [25, 27, 140].

Из этого, можно предположить, что class-файл, созданный компилятором Java, имеет либо некоторую метку, либо определенную структуру байт-кода в class-файле. Таким образом, заменить определенные команды в байт-коде class-файла, скомпилированного при помощи компилятора Java и добиться запуска данного файла без блокировки его со стороны верификатора виртуальной машины Java, проще, чем в class-файле, созданном вручную в шестнадцатеричном редакторе.

Также, стоит отметить, что в документации операционных кодов, используемых в class-файлах, есть эквивалентные операции, которые используются в зависимости от размера программы и сложности логики в коде. В дальнейшем, будет продемонстрировано, что заменить значение или тип переменной в байт-коде скомпилированного class-файла, несколько сложнее, нежели произвести замену эквивалентных операционных кодов.

Однако, есть и другой путь. Разработчики виртуальной машины Java, предполагают, что иногда может потребоваться запуск недопустимого class-файла и, возможно, для отладки, был введен параметр:

`java –noverify  verifier.MyClass` или `–Xverify:none`

При использовании данного параметра, существует шанс получить программу, которая работает не так, как от нее ожидается по заявленной спецификации [82].

1.5 Обзор существующих методик вложения информации в байт-код Java

Данная работа посвящена вложению цифровых водяных знаков в байт-код class-файлов java-приложения или информационной системы. Исследования, находящиеся в открытом доступе по данной тематике необходимо проанализировать на актуальность в текущий момент времени.

Необходимо иметь ввиду, что данное исследование относится к разделу стеганографии о цифровых водяных знаках («Digital Watermarking»). Так как цели у стегосистемы и цифровых водяных знаков разные. Целью ЦВЗ является устойчивость к удалению, робастность. Также, так как речь идет о защите авторских прав на разрабатываемое программное обеспечение, необходимо, чтобы цифровой водяной знак внутри покрывающего сообщения, было возможно обнаружить специализированным ПО для используемого алгоритма вложения. Главное, чтобы ПО с ЦВЗ внутри оставалось приемлемым для использования.

Направление исследований по разработке и внедрению цифровых водяных знаков, в настоящий момент, активно развивается. Однако, русскоязычных

научных работ, которые исследуют проблему скрытого вложения цифровых водяных знаков в байт-код java, в открытом доступе, представлено крайне мало. Большинство из них принадлежит кафедре Защищенных Систем Связи Санкт-Петербургского государственного университета телекоммуникаций им. проф. М.А. Бонч-Бруевича. Тем не менее, исследования за пределами кафедры Защищенных Систем Связи, которые относятся к тематике данной работы прямо или косвенно, есть, в связи с этим, ниже рассмотрены существующие работы коллег.

В исследовании, проведенном В.В. Хашковским, В.Н. Лутаем и В.В. Юрченко в 2009 году, была рассмотрена языковая платформа Java как более удобный вариант для создания крупных и переносимых программных сред [5]. Изучалась скорость выполнения программ на разных платформах - Windows XP и Linux Open SUSE на одинаковых аппаратных платформах с использованием возможностей JVM по подсчету времени исполнения программы. Однако с тех пор производительность новых версий виртуальной машины Java значительно выросла, и появились новые технологии, недоступные в 2009 году.

Комяков Д.С., Елсаков С.М. рассматривают алгоритмы внедрения статических водяных знаков в исходный код ПО [8]. Рассматриваются такие методики, как внедрение методов-болванок в исходный код, расставление разработчиком маркеров, для успешного внедрения цифрового водяного знака. Поднимается проблема различных атак на цифровые водяные знаки в ПО, таких как аддитивные, субстрактивные и искажающие атаки.

Выше были рассмотрены две работы российских авторов, которые наиболее относятся к тематике виртуальной машины Java и вложению цифровых водяных знаков в байт-код Java. Далее обратимся к зарубежным исследованиям в данной области, где она представляет несколько более развитой интерес. Зарубежные коллеги, на текущий момент, наиболее глубже исследуют данный вопрос, используя различные сочетания методик и алгоритмов по вложению информации в байт-код совместно с атаками на результаты работы этих методик.

1.5.1 Статические методики вложения

Наиболее близкой работой по идее, но не по содержанию и реализации, является работа зарубежных коллег по модифицированию операндов байт-кода [157]. В данной работе исследуется и реализуется алгоритм маркировки программного обеспечения цифровым водяным знаком. Путем модификации операндов и/или кодирования опкодов некоторых выбранных инструкций виртуальной машины Java, информация о водяном знаке встроена в байткоды файлов класса Java в виде двоичных кодов. Однако, как было описано ранее, все кодирования и модификации байт-кода работоспособны и эффективны только до того момента, пока не понадобится загрузить class-файл в виртуальную машину Java на исполнение. Потому что, в связи с отклонением наименований или определений операционных команд от оригинальных, вызовет ошибку при проверке файла верификатором виртуальной машины Java из-за некорректных, с точки зрения правил верификации, байт-кодов в class-файле.

В работе «Software Watermarking for Java Program Based on Method Name Encoding» [66], как понятно из названия, авторы предлагают методику маркировки цифровым водяным знаком class-файлов на основе имен методов, находящихся в class-файле. В данной работе предлагается методика маркировки цифровым водяным знаком программного обеспечения на языке Java, основанная на кодировании наименований методов. Авторы делят биты цифрового водяного знака на количество методов в программе. Проверка осуществляется получением наименований всех методов программы и вычленением из них нужных битов цифрового водяного знака, в правильной последовательности. К сожалению, в данной работе, нет экспериментального подтверждения устойчивости данной методики к обфускации/деобфускации всего программного кода.

Curran D., Hurley N. J. и Cinnéide M. Ó. в своем исследовании [76], рассматривают различные способы вложения цифрового водяного знака в байт-код Java, с основным упором на разработанный метод, основывающийся на модуляции слабого сигнала на исходный сигнал хоста. Процедура осуществляется с помощью измерения глубины графа вызовов в разных точках во время выполнения

программы на каком-то конкретном входе. Однако среди примеров «слабого сигнала», в данной работе, встречается только «dummy-method», другими словами - фиктивный метод, который обнаруживается в байт-коде java-приложения без декомпиляци и анализа невооруженным взглядом и крайне легко разрушается, приводя в неработоспособное состояние class-файл и к разуршиению ЦВЗ.

Авторы исследования «A comparative analysis of static java bytecode software watermarking algorithms» [123], проводили анализ существующих алгоритмов для создания и вложения цифровых водяных знаков в jar-файлы java, а затем применяя искажающие атаки к каждой программе с ЦВЗ, обфускацию и оптимизацию. Изучив полученные результаты, авторы обнаружили, что часть внедренного водяного знака была удалена как результат применяемой трансформации. Попутно, авторы, как часто встречается, большую часть своей работы рассуждают о уже существующих атаках, критериях, выдвигаемых к цифровым водяным знакам и прочим уже известным и устоявшимся фактам вложения информации в исполняемые файлы Java. Одним из основных выводов был результат удаления практически всех цифровых водяных знаков, после применения атаки обфускации. К похожим выводам пришли Hamilton J. и Danicic S. в своем исследовании [91], что необходимо добавлять другие формы защиты, такие как запутывание.

Существует еще один алгоритм добавления цифрового водяного знака в программу, его авторы Collberg, C., Huntwork, A., Carter, E. и Townsend, G. рассматривают добавление информации в исполняемый файл на основе топологии графа [73]. Авторы помечают исполняемую программу добавлением программного кода, для которого используется топология графа потока управления для кодирования водяного знака. Граф представляет собой блоки кода (иначе функции), имеющие один вход и один выход, соединенные между собой. Граф водяных знаков объединяется с графиком целевой программы путем добавления дополнительных ребер управляющего потока. Таким образом получается в базовом методе программы добавляется дополнительный вызов метода, являющегося простой проверкой булева значения. Естественно, сам алгоритм имеет более сложную структуру, которая в свою очередь оказывает влияние на байт-код

программы, создавая там артефакты, такие как метки goto, которые значительно затрудняют исполнение программы и возможности ее отладки. Также, данные метки имеют грубые якорные «привязки» к оригинальным методам программы, которые достаточно легко разрушаются простой обфускацией. Фундаментальная зависимость от статической маркировки блоков оставляет программу с водяными знаками уязвимой для атак обфускацией.

Авторы работы «A software fingerprinting scheme for java using classfiles obfuscation» [85], поднимают проблему идентичности внедряемых водяных знаков в java-приложении, когда необходимо, чтобы цифровой водяной знак отражал уникальный идентификатор сотрудника, а не конкретный общий авторский знак компании. Предлагаемый алгоритм основан на обфускации файлов Java, таким образом, чтобы была возможность извлечь цифровой водяной знак не из кода, а из структуры и типа примененной обфускации. Однако результаты работы данного алгоритма легко уничтожается вторичной обфускацией, так как обфускаторы не могут определить какой методикой был обфуцирован переданный на вход class-файл, а, следовательно, будут работать на прямую и грубо, перезаписывая «поверх» структуры работы предыдущего обфускатора. Также, авторы используют подготовку файлов, модифицируя способ доступа к методам и полям на публичный и указанием полей статичными, что просто невозможно сделать без последствий в работе программы, например, при многопоточном запуске программы или продуманной архитектуре, где существует большое количество логики основанной на модификаторах доступа.

Суть процедуры кодирования водяного знака состоит в создании фиктивного метода (пустой фиктивный метод, как указано в работе, должен быть достаточно большим, чтобы вместить водяной знак), компиляция, внедрение образов отпечатков пальцев сотрудников через байт-код class-файла в фиктивные методы, применение схемы обфускации собственной разработки.

В исследовании «A practical method for watermarking java programs» [135] авторы предлагают производить вложение цифрового водяного знака посредством замены различных операционных команд в байт-коде java. Однако для подобных

замен, авторы предлагают использовать фиктивные методы в исполняемом коде, которые тесно встроены в структуру программы.

В большой обзорной научной работе «An Evaluation of the Resilience of Static Java Bytecode Watermarks Against Distortive Attacks» [92, 93] авторы начинают исследование с разбора условий, которым должны соответствовать программы, содержащие в себе цифровой водяной знак. Затрагиваются такие проблемы как способы и критерии эффективности программных водяных знаков, а также стоимость внедрения программного обеспечения. Помимо прочего, авторы произвели разбиение цифровых водяных знаков в программах по категориям, после чего поместив в эти же категории типы атак на цифровые водяные знаки. Авторы исследовали несколько коммерческих систем вложения цифрового водяного знака в программы различными способами, дали комментарии по полученным результатам, дали подробный обзор по уже ранее известным методикам, которые имеют построение на основе графов, фиктивных методов, замене блоков программы, обфускации и других.

После теоретической части работы, авторы приступили к экспериментальной проверке всех методик. В результате было получено, что много утилит и программ уже не совместимы с jar-файлами текущих образцов, некоторые методики не защищены от перезаписи цифрового водяного знака, еще часть программ не прошла проверку верификатором виртуальной машины Java. В выводах авторы сделали ряд заявлений, таких как «Мы подтвердили, что ни один из 14 алгоритмов статического водяного знака не устойчив к сохранению семантики преобразований. Наши результаты сравниваются с предыдущими оценками некоторых алгоритмов статического водяного знака» или «…Программное обеспечение водяных знаков должно быть дополнено другими формы защиты, такие как запутывание или защита от взлома, чтобы лучше защитить программу от нарушения авторских прав и декомпиляции. Хотя мы не оценили все аспекты алгоритмов водяных знаков, мы показали, что статических водяных знаков недостаточно для подтверждения права собственности на программное обеспечение из-за их неспособности к сохранению семантики преобразований».

1.5.2 Динамические методики вложения

Помимо методик и алгоритмов, работающих со статичными данными и кодом, набирает популярность исследование возможности вложения цифровых водяных знаков в байт-код Java «на лету», иначе, во время выполнения, без необходимости остановки работы программы и ее декомпиляции. Считается, что динамические водяные знаки обладают большей стойкостью к различного рода атакам, нежели статические.

Одной из значимых работ, в данной подобласти, является работа «Evaluating the WaterRpg software watermarking model on Java application programs» [70], в которой авторы представляют динамическую модель водяного знака WaterRpg. Авторы рассматривают открытые и скрытые подходы к внедрению цифрового водяного знака в java-приложение во время исполнения, посредством изменения соответствующих вызовов конкретных функций входной прикладной программы в трассируемом потоке исполнения. К сожалению, в данной работе не рассмотрено поведение программы и алгоритма в целом при наличии «зашумляющего» трафика, реализованного, например, посредством дополнительных потоков исполнения программы или реализация другой программы в виде java-агента запущенной в том же потоке исполнения, что и основная программа.

Расширяя свое исследование Kumar K. и Kaur P. дополнительно провели ряд экспериментов по вложению динамического цифрового водяного знака в исполняемые файлы [122]. В своей работе, авторы приводят используемые ими методики создания и вложения цифровых водяных знаков, производят классификацию. Основными алгоритмами по вложению динамического водяного знака являются: использование трассировки программы для выбора ветви кодирования, динамическое внедрение цифрового водяного знака в топологию структуры графа построенной во время выполнения. В качестве основных атак были использованы атаки преобразования, такие как различного рода обфускация кода. Основными выводами исследования являются следующие: алгоритмы водяных знаков с постоянными строками дают лучший результат и наиболее эффективны против искажающих атак, алгоритмы динамических цифровых

водяных знаков несколько сильнее, чем статические алгоритмы, когда применяются искажающие атаки.

Выводы

1) Цифровой водяной знак в исполняемых файлах, как и в медиа форматах необходим для скрытой передачи информации и однозначного определения владельца файла. С каждым годом количество кибер-преступлений растет, как и кража исполняемых файлов, исходного кода программного обеспечения злоумышленниками. Компаниям и разработчикам все сложнее защищать результаты своей интеллектуальной деятельности. Результат опубликован в работах [2, 19, 48, 142].

2) Объем возможного вложения в форматы исполняемых файлов существенно ниже, чем у медиа форматов. В форматы исполняемых файлов существует возможность вложения информации в структуру формата файла и непосредственно в исходный код. При вложении информации в структуру исполняемого файла высокий риск обнаружения вложения и минимально возможный объем в размере 0,2-0,3% от общего объема исполняемого файла. Также нет гарантий устойчивости к атакам. При вложении информации в исходный код, хранящийся в файле, необходимо его редактирование, что приводит к изменению логики работы исполняемого файла. Существует два варианта: динамическое и статическое вложение информации в исполняемые файлы. Динамическое вложение информации в исполняемые файлы может быть менее ресурсоемким, но с худшим результатом и с высокими рисками ложных срабатываний при помехах или различного рода атаках. Также, динамическое вложение информации в исполняемые файлы не может основываться на глубоком анализе структуры исходного кода, взаимосвязях с другими исполняемыми файлами программного обеспечения, что сужает возможности по выбору контейнера для цифрового водяного знака и увеличивает риск его разрушения при атаках. Результаты опубликованы в работах [70, 75, 122].

3) Статическое вложение цифрового водяного знака в исполняемые файлы имеет гораздо большее количество вариаций реализации. На текущий момент, все исследования в области статического вложения цифрового водяного знака в исполняемые файлы основаны на добавлении новых сущностей в исходный код, создании фиктивных методов, вложения цифрового водяного знака в «мертвый код», изменение наименований методов и функций или изменении наименований операторов байт-кода. Данные варианты применимы только в случаях, когда злоумышленник не знает о существовании цифрового водяного знака в исполняемом файле, во всех остальных случаях цифровой водяной не устойчив к различного рода атакам. Изменение наименований операционных кодов в байт-коде работает до тех пор, пока исполняемый файл не запустили на новой версии виртуальной машины Java, верификатор которой не произведет запуск такого исполняемого файла, пометив его не пригодным.

Вложение цифрового водяного знака в созданные фиктивные методы или «мертвый код» не эффективно по ряду причин: цифровой водяной знак легко удалить без вреда для исполняемого файла, он не является частью исходного кода, не устойчив к атакам перекомпиляцией и вложение не может быть произведено в любой исполняемый файл в виду того, что вложение в «мертвый код» накладывает существенные ограничения по выбору исполняемых файлов для вложения цифрового водяного знака.

Также, помимо ограниченности выборки исполняемых файлов пригодных для вложения цифрового водяного знака, существенным недостатком является малый объем возможного вложения равный не более 1,5% от общего объема исполняемого файла. Результаты опубликованы в работах [66, 73, 76, 85, 91, 123, 157].

4) Вышеуказанные факторы позволили сформулировать **противоречие в практике – между** необходимостью увеличения объема цифрового водяного знака в исполняемых файлах, повышением его устойчивости к различного рода искажающим атакам и невозможностью получения желаемых результатов без

разработки новых методик создания и вложения цифрового водяного знака в исполняемые файлы.

5) Для разрешения данного противоречия в работе сформулирована актуальная **цель исследования** – повышение объема цифрового водяного знака и его устойчивости к атакам, предложив методики создания и вложения цифрового водяного знака на основе эквивалентных замен операционных кодов байт-кода Java за счет использования расширенного набора опкодов и функций затрудняющих эффективность атак на цифровой водяной знак, что соответствует паспорту специальности 2.3.6. «Методы и системы защиты информации, информационная безопасность».

6) Системный анализ известных публикаций и исследований в рассматриваемой предметной области показал следующее. В известных работах были рассмотрены различные методики создания и вложения цифровых водяных знаков в исполняемые class-файлы. Вместе с тем, в известных работах не учитываются факторы постоянного обновления виртуальной машины Java, цифрового водяного знака как неотделимой части исходного кода, не устойчивости цифрового водяного знака к различного рода атакам, необходимость комплексного анализа всех исполняемых файлов java-приложения или информационной системы. Результат опубликован в работе [8, 38].

7) Проведенный анализ ранее опубликованных работ в исследуемой предметной области позволил сформулировать **научную актуальность (противоречие в науке)** исследования, которая обусловлена невозможностью достижения поставленной цели на базе существующих методик в силу следующих их недостатков:

а) Утрата актуальности в связи с изменениями в ключевых функциях языка программирования Java и глобальных изменениях в работе виртуальной машины Java

б) Малый объем цифрового водяного знака возможного к вложению

в) Отсутствие устойчивости цифрового водяного знака к атакам декомпиляцией

г) Отсутствие устойчивости цифрового водяного знака к атакам обфускацией

д) Использование крайне ограниченного набора опкодов для создания и вложения цифрового водяного знака, что сужает возможности по выбору исполняемых class-файлов в качестве контейнера для цифрового водяного знака

е) Использование не пригодных к эквивалентным заменам операционных кодов, приводящих к значительному снижению эффективности работы Java-приложения или риску неработоспособности

ж) Подавляющее число исследований в области стеганографии в исполняемых файлах Java принадлежат заграничным коллегам, таким как J. Chen, C. Wang, Kumar K., Kehar V., Kim S., Hillert E., Hamilton J., Danicic S., Collberg C., Monden A. Вышеупомянутые исследования ориентированы на возможность вложения цифровых водяных знаков в байт-код class-файлов, но не акцентируют внимание на таких параметрах, как его объем для возможности практического применения вложения или устойчивость к разрушающим атакам.

з) Ни одна существующая методика создания и вложения цифрового водяного знака в class-файлы Java не способна произвести полноценное покрытие информационной системы с возможностью проверки целостности.

8) Для разрешения указанного противоречия в науке в работе была поставлена **научная задача** – разработка методик, позволяющих наиболее эффективно произвести создание и вложение цифрового водяного знака в байт-код class-файлов java-приложений и информационных систем на Java, с повышением устойчивости к различного рода атакам, направленных на разрушение или повреждение цифрового водяного знака.

Объектом исследования является байт-код исполняемых файлов интерпретируемых языков программирования выполняющихся в виртуальной машине Java, а **предметом исследования** – возможность создания и вложения устойчивого к атакам цифрового водяного знака в байт-код class-файлов за счет не декларированных возможностей языка программирования Java и использования расширенного набора операционных команд байт-кода пригодных для эквивалентных замен.

9) Была проведена формализация научной задачи. В общей вербальной форме, постановка научной задачи может быть сформулирована так – увеличение объема цифрового водяного знака и его устойчивости к атакам декомпиляцией и обфускацией, за счет использования расширенного набора операторов байт-кода пригодных для эквивалентных замен, анализа структуры байт-кода class-файлов и их взаимосвязей в java-приложениях и информационных системах, использования функций и шаблонов, затрудняющих декомпиляцию class-файлов, использования единого цифрового водяного знака для информационных систем.

10) Для решения общей научной задачи в интересах достижения поставленной цели, она была декомпозирована на частные научные задачи:

а) Разработка методики создания и вложения цифрового водяного знака увеличенного объема в байт-код class-файла. Исследование произведенного вложения, средний объем информации возможной к вложению в class-файл от его объема в процентах, сравнение с другими методиками;

б) Разработка методики создания и вложения цифрового водяного знака в class-файлы java-приложения устойчивого к атакам декомпиляцией направленных на его разрушение. Исследование устойчивости произведенного вложения к атакам декомпиляцией;

в) Разработка методики создания и вложения цифрового водяного знака в class-файлы информационной системы устойчивого к атакам обфускацией направленных на его разрушение. Исследование устойчивости произведенного вложения к атакам обфускацией.

Глава 2. Методика создания и скрытого вложения цифрового водяного знака увеличенного объема в class-файлы

2.1 Формулирование необходимости цифровых водяных знаков в исполняемых файлах Java

В данный момент, Java один из самых популярных языков программирования по рейтингу TIOBE [60, 167]. Java - язык, на котором разработано огромное количество приложений, используемых в разных областях жизни - от приложений на мобильных устройствах до банковского и строительного секторов. Это привело к тому, что программы и компоненты проектов, написанные на Java, стали целью интереса злоумышленников.

Java является кроссплатформенным языком программирования, что позволяет запускать программы на разных операционных системах. Он широко используется в различных областях, включая бесконтактные карты, роутеры и оборудование Cisco. Однако, распространенность языка также подразумевает определенные риски. Исполняемые class-файлы, которые составляют java-приложения и информационные системы, могут быть легко проанализированы опытным специалистом, что позволяет получить исходный код приложения [18, 52]. Кроме того, существует множество декомпиляторов class-файлов, которые могут привести к повреждению функциональности программы во время декомпиляции.

Также, необходимо добавить примечание, касательно загрузки байт-кода в виртуальную машину Java. Во время запуска Java программы в виртуальной машине Java, при загрузке байт-кода из class-файла загрузчиком классов в виртуальную машину Java необходимо, чтобы байт-код был не только безопасным для виртуальной машины Java, но и, в первую очередь, понятным верификатору [162]. У верификатора виртуальной машины Java существует ряд проверок, проводимых перед загрузкой class-файла в виртуальную машину Java. Некоторое виды проверок, выполняемые верификатором:

- Инициализация переменных перед их использованием
- Согласование типов ссылок при вызове метода
- Соблюдение правил доступа к закрытым данным и методам
- Доступ к локальным переменным в стеке во время выполнения
- Отсутствие переполнения стека
- Проверка правильности указателя. Проверка того, что указатель не используется при получении, как результат другой операции не свойственной ему. Например, арифметической
- Проверка вызова внутри class-файла объекта, который содержит объявленное меньшее количество параметров, чем количество параметров, с которым его вызывают или с неверными типами параметров
- Невозможные операции приобразоваия типов
- Неправильный набор параметров при вызове операции на Java-машине
- Неправильное взаимодействие с регистрами в JVM.

При невыполнении какой-либо из этих проверок класс считается поврежденным и загружаться не будет. Если хотя бы одна проверка дает сбой – файлу будет отказано в загрузке и исполнении.

Таким образом, можно зашифровать class-файл, сделать обфускацию кода, но в момент загрузки class-файла загрузчиком классов и чтением его байт-кода для последующей верификации, будет необходимо, чтобы верификатор определил в полученных командах именно байт-код, с которым он может работать.

Исходя из описанного выше, можно с уверенностью сказать, что методики защиты class-файла посредством его шифрования, по меньшей мере бессмысленны, так как проделанную работу придется «откатывать» назад, для того, чтобы верификатор виртуальной машины Java имел возможность проверить байт-код class-файла и перевести его на следующий уровень проверки и последующего запуска. И именно в момент загрузки байт-кода в виртуальную машину Java, для последующих проверок, он максимально уязвим и не защищен ни от каких видов атак [39, 150, 152].

Таким образом, существует необходимость разработать решение по защите авторских прав на class-файлы. Вложение цифрового водяного знака достаточного объема для того, чтобы подтвердить авторство, в class-файлы java-приложения является необходимым решением [42].

2.2 Теоретико-множественная модель языка программирования Java и математическая модель стеговложения

Для текущего исследования необходимо произвести анализ языка программирования и как устроена экосистема его компонентов. В связи с этим необходимо составить формализованное множественное описание компонентов, принципов их работы и взаимосвязей друг с другом [20, 21].

Байт-код Java состоит из операционных кодов, которые можно представить, как множество.

$$B =< b1, b2, \dots, bi, \dots, bn >$$

Где B – множество операционных кодов программы (Byte-codes), a bi

$$bi \in B$$

Как описывалось выше, компилятор java, производит компиляцию исходного кода, написанный программистом, не в «родные» команды системы, где происходит запуск, а в байт-код, который в последствии виртуальная машина Java интерпретирует в родные команды системы, на которой осуществляется запуск программы [98]. Таким образом, все файлы *.java содержащие в себе исходный код компилируются в файлы с расширением *.class, которые содержат в себе не исходный код, а байт-код понятный виртуальной машине Java. Обычно, программа состоит более, чем из одного class-файла, в зависимости от ее функциональных возможностей. Исходя из этого, программа на Java – это набор class-файлов, которые также могут быть представлены множеством.

$$F =< f1, f2, \dots, fi, \dots, fn >$$

Где F – множество class-файлов программы (Files), а fi

$$fi \in F$$

Таким образом, программу на языке программирования Java, можно представить двумя взаимосвязанными множествами. Если говорить точнее, то множество операционных кодов, байт-кода, можно рассматривать, как подмножество множества class-файлов, так как одни состоят из других.

$$B \subset F$$

Также, помимо основных элементов программы, в ней всегда присутствуют дополнительные, такие как – ресурсы. Это различные файлы конфигураций, тестовые данные, данные поиска, интерфейсные элементы программы, которые также можно представить множеством ресурсов.

$$R =< r1, r2, \dots, ri, \dots, rn >$$

Где R – множество ресурсов программы (Resources), а ri.

$$fi \in R$$

Так как множество ресурсов программы представляет собой различные ресурсы, целесообразно выделить их в подмножества. Такие как, файлы конфигураций, тестовые данные, интерфейсы, журнал операций программы. Соответственно, формула множества ресурсов программы приобретает вид:

$$R =<\mathrm{c1}, \mathrm{c2}, . . , \mathrm{ci}, \dots, \mathrm{t1}, \mathrm{t2}, \dots, \mathrm{ti}, \dots, \mathrm{i1}, \mathrm{i2}, \mathrm{ii}, \dots, \mathrm{j1}, \mathrm{j2}, \dots \mathrm{ji}>$$

Где c1,c2,..ci – элементы множества C (configurations) – файлов конфигураций программы; t1,t2,…ti – элементы множества T (Test data) – тестовых данных программы; i1,i2,…,ii – элементы множества I (Interfaces) – интерфейсов; j1,j2,…ji – элементы множества J – журнал операций программы. Все вышеописанные

множества, имеет смысл рассматривать, как подмножества множества ресурсов (R). Следовательно,

$$C, T, I, J \subset R$$

Таким образом, конкретная программа, при первоначальном описании, может быть представлена в виде системы, приведенной на рисунке 2.1 и отображена теоретико-множественной формулой.

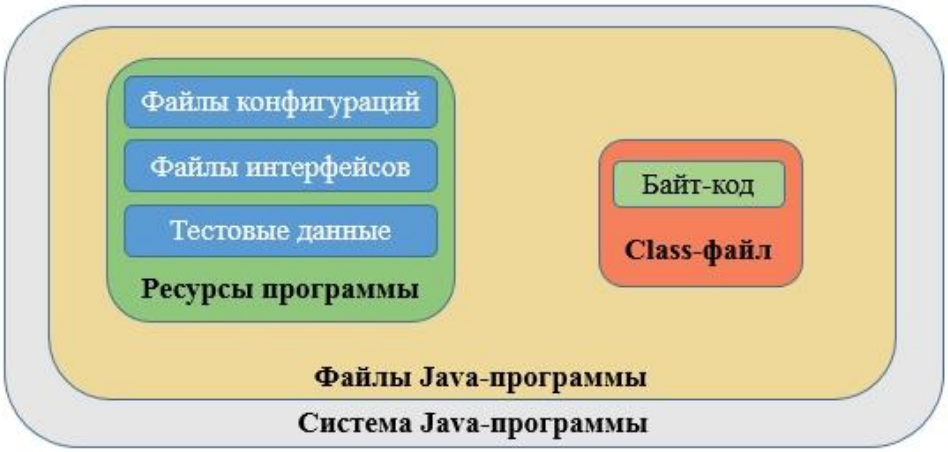


Рисунок 2.1 – Теоретико-множественная модель java-приложения

$$S = < \big((B \subset F), (C, T, I, J \subset R)\big) \subset PF >$$

Где PF (Program Files) – все файлы какого-либо формата и назначения, задействованные в функционировании и запуске java-приложения.

В процессе разработки, внедрения в эксплуатацию, работы и поддержки программы, может происходить изменение и накопление сведений о системе, следовательно, теоретико-множественное представление программы, как системы может меняться с течением времени, и отображать различные отношения между группами множеств и подмножеств [147, 161]. В данном разделе, представлено базовое теоретико-множественное представление экосистемы языка Java и его компонентов.

Модель стеговложения

Допустим, мы рассмотрим java-приложения в качестве машин Тьюринга. Если существует два java-приложения, одно из которых производит анализ и вложение ЦВЗ в class-файлы другого приложения, то время, необходимое для завершения такого процесса, должно быть ограничено полиномиально от размера данных в class-файлах. Для моделирования такого java-приложения может использоваться вероятностная машина Тьюринга, которая ограничена полиномиальным временем и обозначается как PPT.

Для обозначения машины Тьюринга (TM) мы будем использовать символ q, а для двоичной строки, которая является входными данными для TM, - символ x. Размер машины Тьюринга мы будем обозначать как |q|, а результат вычисления TM q на входных данных x - как q(x). Количество шагов (времени), необходимых машине Тьюринга q для вычисления результата на входных данных x, мы будем обозначать как time(q(x)).

Если для любых входных данных x результаты вычисления $q_1(x)$ и $q_2(x)$ равны друг другу, то машины Тьюринга q1 и q2 считаются эквивалентными. Если функция $\nu : N \to [0,1]$ убывает быстрее, чем любая обратная функция многочлена, то она признаётся пренебрежимо малой и обозначается как neg(·). Для любого $k \in N$ существует такое $n_0 \in N$, что для всех $n > n_0$ выполняется неравенство $\nu(n) < 1/n^k$. Производный полином обозначим poly(·).

Таким образом, при реализации стеговложения для семейства TM M необходима такая PPT ST, которая удовлетворяет следующим условиям:

1. **Функциональная эквивалентность.** Для любой TM q, $q \in M$, в результате стеговложения ST(q) строится TM, эквивалентная исходной машине q.

2. **Полиномиальные издержки.** Для любой TM q, $q \in M$, размер и быстродействие любой TM ST(q) отличается от размера и быстродействия TM q не более чем полиномиально, т.е. $|ST(q)| = poly(|q|)$ и $time(ST(q(x))) = poly(time(q(x)))$.

3. **Эффективная неотличимость программ.** Для любой пары эквивалентных ТМ q_1 и q_2 из класса M, имеющих одинаковый размер, распределения вероятностей случайных величин $ST(q_1)$ и $ST(q_2)$ вычислительно неотличимы, для любой PPT D справедливо соотношение $|Pr[D(ST(q_1)) =1] - Pr[D(ST(q_2)) =1]| = \leq neg(|q|)$.

2.3 Отбор контейнера для вложения информации в исполняемые class-файлы текущего исследования

Ранее, был сделан вывод, о том, что наиболее эффективным способом отбора контейнера для вложения информации, видится способ суррогатной стеганографии. При необходимости передачи скрытого сообщения по незащищенным каналам связи, использование только этого метода, действительно, оправдано. Брать любой файл, и производить вложение информации на весь доступный для вложения объем.

Однако, для данного исследования необходимо скорректировать подходы выбора контейнера. Так как в чистом виде, не один из способов отбора контейнера не подходит.

В зависимости от задачи выбираются различные методы стеганографии для использования в контейнере. Применение конструирующей стеганографии позволяет достичь максимальной эффективности, при которой сообщение шифруется таким образом, чтобы сохранить первоначальную модель шума. Это усложняет обнаружение скрытого сообщения. Метод селективной стеганографии требует больших временных и ресурсных затрат и не подходит для малых сообщений. В то время как, метод суррогатной стеганографии, не всегда позволяет вложить сообщение полностью, обладает преимуществом в скорости и подходит для небольших сообщений, не искажая свойства контейнера.

Необходимость производить объемное вложение в class-файлы Java отсутствует, так как задача данного исследования – вложение цифрового водяного

знака в class-файлы, с целью однозначного определения владельца class-файла и прав на разработанный или разрабатываемый программный продукт.

В потоковых контейнерах крайне сложно отследить место начала скрытого сообщения. В контейнерах произвольного доступа эту инфорацию скрыть не представляется возможным. Наибольшую эффективность будут представлять контейнеры произвольного доступа, так как объем вложения и его надежности выше, чем у потоковых контейнеров. Таким образом набор контейнеров произвольного доступа наиболее интресны для целей вложения информации. Использование контейнеров с фиксированной длиной имеет ещё одно преимущество перед потоковыми контейнерами: заранее известен размер скрываемого файла, что позволяет выбрать определенную последовательность скрывающих бит в соответствии с размером файла.

Использование нескольких небольших контейнеров с небольшим объемом информации гораздо более надежно и эффективно, чем использование одного-двух больших контейнеров с большим объемом информации в них [24, 35].

Для эффективного вложения сообщения в исполняемый файл необходимо выполнить несколько шагов. Во-первых, разделить сообщение на несколько более компактных. Для этого можно использовать сообщения меньшего объема или разделить большое сообщение на несколько небольших. После этого необходимо выбрать оптимальное количество контейнеров фиксированной длины. Данные контейнеры могут быть систематическими, чтобы легко определить шумовые биты, которые можно использовать для скрытой информации. После этого, в каждом из выбранных контейнеров необходимо произвести необходимые семантические замены.

Таким образом, скрытая информация станет частью контейнера и не будет выделяться, что не вызовет подозрений у пользователей. Важно понимать, что эти шаги необходимы для успешного вложения сообщения в исполняемый файл и должны быть выполнены с максимальной точностью. Только в этом случае можно быть уверенным в безопасности передачи информации. Следовательно, устойчивость контейнера и объем скрываемых данных увеличивается. Данный

подход эффективно реализуется в приложениях, разработанных в парадигме ООП, где каждый отдельный класс – является отдельным исполняемым файлом, другими словами, контейнером, с возможностью вложения в него информации.

Таким образом, на входе мы уже будем иметь некоторый набор class-файлов, представляющих собой java-проект – программу для ЭВМ (java-приложение). Очевидно, что, исходя только из этого условия, способ селективной стеганографии однозначно не подходит. Однако способ конструирующей стеганографии, также, не полностью удовлетворяет условиям. Так как, корректировки самого информационного сообщения, вложение которого будет производится в class-файлы, рассматриваются как «минимально возможные». Это связанно с тем, что цифровым водяным знаком в исполняемых файлах может выступать каким-либо образом зашифрованное название компании, производящей программный продукт или имя автора. В связи с этим, сильная корректировка скрываемой информации видится невозможной, так как она в любом случае будет незначительного объема.

Также, как уже было описано выше, исполняемые файлы любого языка программирования, не способны вместить большой объем дополнительной скрытой информации. При необходимости сокрытия больших объемов информации, проще будет воспользоваться другими форматами.

Таким образом, от способа конструирующей стеганографии уместно использовать только критерий того, что первоначальный шум в контейнере должен быть неизменен или подвергнут минимальным изменениям.

Однако, способ отбора контейнера методом Суррогатной стеганографии, также не удовлетворяет в полной мере. Рассмотрим подробнее данный способ.

1. Исходя из информации, описанной выше, следует, что набор class-файлов, которые представляют собой некоторую программу для ЭВМ ограничен, но это не означает, что возможность выбора контейнера полностью отсутствует. Возможность ограничена.

2. Метод суррогатной стеганографии предполагает, что для вложения информации будет использоваться любой контейнер. Даже при условии, что контейнер не пригоден для вложения данного сообщения. Этот вариант также

неприемлем, так как, возможна ситуация, когда в проекте существует корневой файл-ядро, содержащий в себе ключевую логику программы, следовательно, защита данного файла цифровым водяным знаком будет первым приоритетом.

3. Производится замена байтов контейнера байтами сообщения, что позволяет сообщению стать частью контейнера. Следовательно, изменение контейнера не будет замечено. Как уже было описано ранее, это не является проблемой для цифровых водяных знаков, используемых в исполняемых файлах.

Также, необходимо подробнее остановится на отборе контейнеров по типу организации. Самым эффективным способ вложения ЦВЗ в class-файлы является использование несистематических контейнеров. Такие контейнеры позволяют скрыть информацию в стеганограме, требуя полной обработки ее содержимого для выделения или раскрытия скрытых данных. Благодаря этому критерию, внедрение цифровых водяных знаков в class-файлы становится еще более надежным и безопасным.

Таким образом, подводя итог, можно сказать, что для данного исследования имеет смысл объединения критериев отбора контейнеров из различных способов в один новый, удовлетворяющий критериям исследования.

В качестве основных шагов модифицированной методики отбора контейнера, предлагается использовать:

1. Полный анализ всех доступных к вложению class-файлов, java-приложения. Под анализом подразумевается анализ структуры файлов и используемых в них определенных синтаксических конструкций языка программирования.

2. Отбор class-файлов производится по их структуре и нелинейности содержащегося в них кода.

3. Выбор наиболее эффективной методики создания ЦВЗ на основе полученной информации о структуре class-файлов.

4. Исходя из полученной информации, производится выбор наиболее оптимальной методики вложения ЦВЗ в class-файлы.

Следовательно, приоритет отдается исполняемым class-файлам с нелинейной структурой кода, несистематическим типом организации, где в зависимости от структуры кода подбирается способ генерации цифрового водяного знака, а также методика вложения цифрового водяного знака в исполняемые class-файлы.

2.4 Критерии применяемые к цифровому водяному знаку

Поскольку вложение цифрового водяного знака будет производиться непосредственно в байт-код class-файла, который будет проходить проверку верификатора виртуальной машины Java, то необходимо выработать ряд ключевых критериев, на основании которых вложение цифрового водяного знака будет считаться допустимым с точки зрения работоспособности программы и эффективным с точки зрения погружения информации в контейнер [35].

Эффективность цифрового одяного знака может быть описана в трех пунктах:

1. Устойчивость цифрового водяного знака к атакам, направленным на его преобразование с целью повреждения или удаления. Преобразование, оптимизация, перекомпиляция, декомпиляция, обфускация кода. Не должно быть возможности отделить цифровой водяной знак от class-файла.

2. Цифровой водяной знак должен находится в пределах допустимого объема вложения для выбранного class-файла программного обеспечения.

3. Пользователь программного обеспечения не должен знать, что в программном обеспечении присутствует цифровой водяной знак. Следовательно, наличие цифрового водяного знака не должно влиять на работоспособность программы, ее отображение, скорость исполнения.

Исходя из описанных выше требований, которые являются маркерами эффективности цифрового водяного знака, можно вывести основные критерии при его создании:

1. ЦВЗ не должен влиять на работоспособность программы.
2. Должен существовать способ извлечения или чтения ЦВЗ с целью его проверки. При больших объемах необходима разработка программного обеспечения, позволяющая проивести чтение ЦВЗ автоматизированно.
3. Если ЦВЗ имеет меньший объем по сравнению с доступным объемом для вложения информации, следует производить вложение до тех пор, пока существует возможность. Данный подход позволяет снизить вероятность разрушения всего ЦВЗ в исполняемом файле. При увелечении количества ЦВЗ на один исполняемый файл не должна пострадать логика работы файла или его отображение пользователю.
4. Методика или алгоритм, позволяющий произвести вложение ЦВЗ в исполняемый файл, не должны быть ключом к методике или алгоритму, позволяющему произвести чтение ЦВЗ из файла.
5. Существует ряд ПО, для которого критическим атрибутом является его скорость работы. ЦВЗ не должен изменять данный атрибут исполняемых файлов.
6. Различного рода манипуляции с исполняемым файлом или программой не должны првести к разрушению ЦВЗ. Различные атаки на ЦВЗ, такие как обфускация исходного кода и перекомпиляция исполняемого файла, также не должны разрушить ЦВЗ, он должен быть устойчив к ним.

2.5 Эквивалентность операционных команд (опкодов) виртуальной машины Java

Особенность, позволяющая языку Java решать проблемы обеспечения безопасности и переносимости программ, состоит в том, что вывод компилятора Java не является исполняемым кодом [78, 168]. Байт-код Java - это промежуточное

представление программ java или набор инструкций исполняемых виртуальной машиной Java [128].

Компилятор Java получая на вход файл с исходным кодом программы, производит его компиляцию в class-файл. В процессе компиляции, семантические конструкции и другие части исходного кода, написанного на языке программирования Java, интерпретируются в операционные команды (байт-код) виртуальной машины Java. Таким образом, при исполнении class-файла JVM оперирует опкодами.

Для любой стандартной атомарной конструкции языка программирования существует ее опкод в виртуальной машине Java. Для каждого типа есть свой оператор сравнения [165]. Например, для int - это if_icmpge.

Набор инструкций виртуальной машины Java спроектирован таким образом, что позволяет отличать типы операндов той или иной инструкции по разному байт-коду инструкции. Если метод работает только с типами значений int, то все инструкции скомпилированного байт-кода (iconst_0, istore_1, iinc, iload_1, if_icmplt) также оперируют только с типами данных int.

Например, сравнение двух значений типа int в цикле for может быть реализовано с помощью одной инструкции if_icmplt. В то же время не существует одной инструкции в наборе виртуальной машины Java, которая реализовывала условный переход по результату сравнения значений типа double. Поэтому для сравнения значений типов double необходимо использовать инструкцию dcmpg и условный переход по результату сравнения - инструкция iflt.

В Java возможно заменить любое ветвление потока выполнения, то есть логические операции с условиями, на эквивалентные, но с противоположным логическим выражением. В этом случае может измениться логика ветвления и, следовательно, результат, который выводится class-файлом. Чтобы избежать этого, при замене логического выражения нужно поменять местами опкоды ветвей условия. Таким образом, логика работы не меняется, но меняются местами наборы команд. Пример в листинге 2.1 демонстрирует замену в байт-коде двух опкодов на эквивалентные с перестановкой ветвей условия.

Листинг 2.1 – Изначальный вариант исходного кода

```
int a = 1;
int b = 2;
if (a < b)
System.out.println(“Это команда №1”);
else
System.out.println(“Это команда №2”);
```

Листинг 2.2 – Байт-код исходного варианта ветвления программы

```
iconst_1
istore_1
iconst_2
istore_2
iload_1
iload_2
if_icmpge 31
ldc "Это команда №1"
goto 34
ldc "Это команда №2"
```

Операционная команда if_icmpge означает для виртуальной машины Java старт сравнения двух переменных, которые до этого были помещены в стек командами istore_1 и istore_2. Суть команды можно определить, обратив внимание на листинг 2.1, в котором продемонстрировано, что выполняется сравнение двух переменных, причем переменная «a» должна быть меньше переменной «b», чтобы на экран вывелась строка «Это команда №1».

Следовательно, команда if_icmpge производит сравнение: значение_1 меньше, чем значение_2». Однако существует эквивалентная ей команда -

if_icmple, которая, в свою очередь, осуществляет сравнение в обратную сторону «Значение_1 больше, чем значение 2».

Соответственно, возможно отредактировать байт-код скомпилированного class-файла произведя замену команд этой операции сравнения [44]. Исходный код в таком случае будет выглядеть так, как представлено в листинге 2.3.

Листинг 2.3 – Представление кода, после редактирования байт-кода

```
int a = 1;
int b = 2;
if (a > b)
System.out.println(“Это команда №2”);
else
System.out.println(“Это команда №1”);
```

Совершив данные манипуляции с байт-кодом Java, мы изменили битовое представление class-файла, так как команда if_icmpge будет представлена как 10100010, а команда if_ icmple как 10100100, что является большим значением. Тем самым мы увеличили объем class-файла, однако, существуют случаи, когда можно сделать замену в точности наоборот. Выводы, полученные в данном разделе, используются и дополняются в последующих разделах данной работы.

2.6 Методика создания и скрытого вложения цифрового водяного знака увеличенного объема class-файлы

Создание цифровых водяных знаков для программного обеспечения отличается от создания цифровых водяных знаков в мультимедиа тем, что водяной знак должен быть вложен в исполняемый файл, а не в медиа-данные, такие как изображение или аудиофайл. Существенная трудность защиты авторских прав в цифровых данных – это невозможность предотвращения копирования цифровых данных. Исходный код программного обеспечения нельзя защитить от

копирования [17], но цифровой водяной знак в исполняемом файле, при проверке, будет доказательством авторских прав на данный исполняемый файл [26, 30, 38].

Методы, применяемые для создания цифровых водяных знаков для программного обеспечения можно разделить на два типа, статические и динамические.

Статические цифровые водяные знаки. Данные знаки помещены в код или данные программного обеспечения. Они скрывают цифровой водяной знак в избыточной области программы или не задокументированных возможностях платформ, систем, языков программирования. Этот тип похож на вложение водяных знаков в избыточных областях мультимедиа. Производится вложение цифрового водяного знака в инструкции кода исполняемого файла.

Динамические цифровые водяные знаки. Данные цифровые водяные знаки используются в структурах создаваемых рабочей программой. Они скрывают водяной знак в избыточных вычислениях программы. Динамические водяные знаки сгенерированы программой во время ее выполнения, обычно на определенной входной последовательности. Основной недостаток данного типа цифровых водяных знаков заключается в том, что способ запуска программы, ее поведения во время выполнения, действия виртуальной машины Java на содержащийся в class-файле код максимально нестабильны [31]. Новые версии виртуальной машины Java выходят гораздо чаще, чем обновления для языка программирования Java. Следовательно, способы и вариации исполнения java-приложения внутри виртуальной машины Java, операционной системы или платформы могут меняться от версии к версии, что ставит под угрозу работоспособность методик, основанных на динамических цифровых водяных знаках.

В этой главе описывается методика создания и вложения увеличенного объема цифрового водяного знака статического типа, который устойчив к различным видам атак. Кража java-приложений может происходить за счет копирования отдельных class-файлов и последующего их повторного использования в других java-приложениях. Разработчики java-приложений могут

легко защитить свои важные class-файлы путем добавления в них цифрового водяного знака [36]. Цифровой водяной знак может быть легко обнаружен там, где был использован украденный class-файл [41].

2.6.1 Анализ и выборка class-файлов java для вложения цифрового водяного знака

Прежде чем приступить непосредственно к созданию цифрового водяного знака с последующим его вложением в байт-код java-приложения, необходимо провести анализ всего Java-проекта. Методики для вложения цифровых водяных знаков постоянно разрабатываются и улучшаются для того, чтобы обеспечить максимальным покрытием программные объекты. Необходимо однозначно понять, что проект и составляющие его class-файлы пригодны в качестве контейнера цифрового водяного знака, вложение которого будет происходить по методике, требующей наличия некоторых условий [32].

Однако необходимо знать какой максимальный объем информации можно вложить в файлы Java-проекта, какими методиками воспользоваться и надо ли производить вложение цифрового водяного знака в каждый class-файл, ведь это трата времени и средств. Если компания нанимает специалиста, то это дополнительные расходы, а также риск кражи бизнес-логики программы нанятым человеком или некачественные рекомендации.

В данный момент, многие компании создают собственные стандарты по разработке и оформлению кода. Например, большая часть участников крупнейших сообществ по Java считают, что сейчас оптимальная длина class-файла не должна превышать 300 строк, некоторые расширяют данное значение до 500, но не более [159, 160]. Таким образом, мы можем предположить, что на текущий момент развития программных средств, техник с подходами к программированию, возможно ориентироваться на средний размер файла в 300-500 строк. На основе данной информации возможно предположить, сколько в среднем на один class-файле java-проекта должно быть операторов условных переходов, циклов, переменных и других опкодов виртуальной машины Java для того, чтобы

существовала возможность вложения цифрового водяного знака минимального объема.

Если такой анализ производить вручную, то это займет большое количество времени, в связи с объемами текущих java-проектов. Поэтому необходимо программное обеспечение, которое произведет расчет необходимых параметров. Таким образом, программе-анализатору необходимо поочередно получить байт-код каждого class-файла из java-проекта, проверить его на наличие операционных кодов, которые могут использоваться в методике для вложения цифрового водяного знака, запомнить количество данных операторов для каждого class-файла java-приложения. В итоге, после завершения работы программы-анализатора должен быть составлен отчет, содержащий в себе сводную таблицу с наименованиями class-файлов, и примерный объем информации, который возможно эффективно вложить в данный class-файл.

Для корректной оценки java-проекта, программа-анализатор должна содержать в себе предварительные данные о среднестатистических характеристиках проектов на языке программирования Java. Таким образом в минимальный набор информации будет входить:

1. Среднее количество строк кода в каждом class-файле подобного рода программного обеспечения. Данный параметр можно именовать как LOC (Line of Code)
2. Информация о требованиях к файлам с кодом для использования каждой из методики вложения цифрового водяного знака в байт-код class-файла
3. Среднее количество повторений паттернов проектирования кода и конструкций языка программирования на один class-файл для каждой известной анализатору методики вложения информации в байт-код class-файла.
4. Среднестатистический процент информации, который можно вложить на каждые 100Kb объема файла.

С данным набором данных изначально добавленных в программу-анализатор возможно сделать приблизительный и грубый подсчет объема цифрового водяного

знака, который можно будет вложить в каждый class-файл java-проекта [155]. Также, принимая на вход информацию о количестве class-файлов в проекте (подсчет количества файлов в папке java-проекта с расширением *.class), программа-анализатор рассчитает примерный объем информации, который будет возможно вложить в качестве цифрового водяного знака в текущий java-проект.

Грубая оценка помогает сэкономить время на специалисте, который потратит на это гораздо больше времени. Также, грубая оценка помогает понять, возможно ли вложение цифрового водяного знака в текущий java-проект. С помощью более глубокого анализа возможно рассчитывать паттерны и используемые конструкции в текущем java-проекте и исходя из полученных данных наиболее эффективно работать с байт-кодом анализируемого проекта с учетом используемых в нем конструкций, ветвлений кода, циклов и архитектуры проекта.

Действия, совершаемые программой-анализатором:

1. Подсчет количества class-файлов проекта
2. Анализ байт-кода каждого class-файла на предмет наличия операторов условного перехода, циклов с условием, переменных (с возможной заменой в сторону увеличения int на long и другие)
3. Выборка class-файлов, содержащих минимально необходимый набор операционных команд виртуальной машины Java, позволяющий произвести вложение цифрового водяного знака
4. Расчет предварительно возможного объема вложения по каждому class-файлу
5. Расчет предварительно возможного объема вложения по всему java-проекту на основе class-файлов, прошедших по условию из пункта 3.
6. Выбор опкодов виртуальной машины Java для редактирования байт-кода class-файла с минимальным изменением структуры кода в class-файле

2.6.2 Создание цифрового водяного знака на основе результатов анализа class-файлов java-проекта

Процесс создания и вложения цифрового водяного знака разделен на фазы, которые разделены на действия. Фаза создания цифрового водяного знака и фаза вложения цифрового водяного знака. В данном блоке речь идет о первой фазе и действиях, направленных на ее успешное завершение. Фаза создания цифрового водяного знака содержит 3 действия. Из которых 2 действия направлены на анализ java-приложения, подготовку к созданию цифрового водяного знака и 1 действие направленное, непосредственно, на создание цифрового водяного знака.

Действие 1. Анализ java-проекта и отбор class-файлов.

В первую очередь, при проведении анализа java-проекта, необходимо определить допустимый объем информации, возможный для вложения в class-файлы. Для этого необходимо:

1. Провести анализ каждого class-файла проекта
2. Определить те, которые пригодны для вложения цифрового водяного знака (содержат в себе необходимые конструкции и операции)

Необходимо дать пояснения по каждому шагу Действия 1. В первую очередь необходимо понимание, в какое количество class-файлов проекта возможно вложение цифрового водяного знака посредством представляемой методики. В следствии этого, требуются подготовительные действия. Методика основана на не задокументированных возможностях виртуальной машины Java, которые связаны с эквивалентными опкодами и принципами их использования. В связи с этим, необходимо провести анализ каждого class-файла java-приложения на предмет наличия необходимых логических конструкций, шаблонов проектирования кода, которые в последствии при интерпретации в байт-код виртуальной машины Java дадут опкоды, которые имеется возможность редактировать или заменять посредством предлагаемой методики [153]. Данное действие демонстрирует рисунок 2.2.

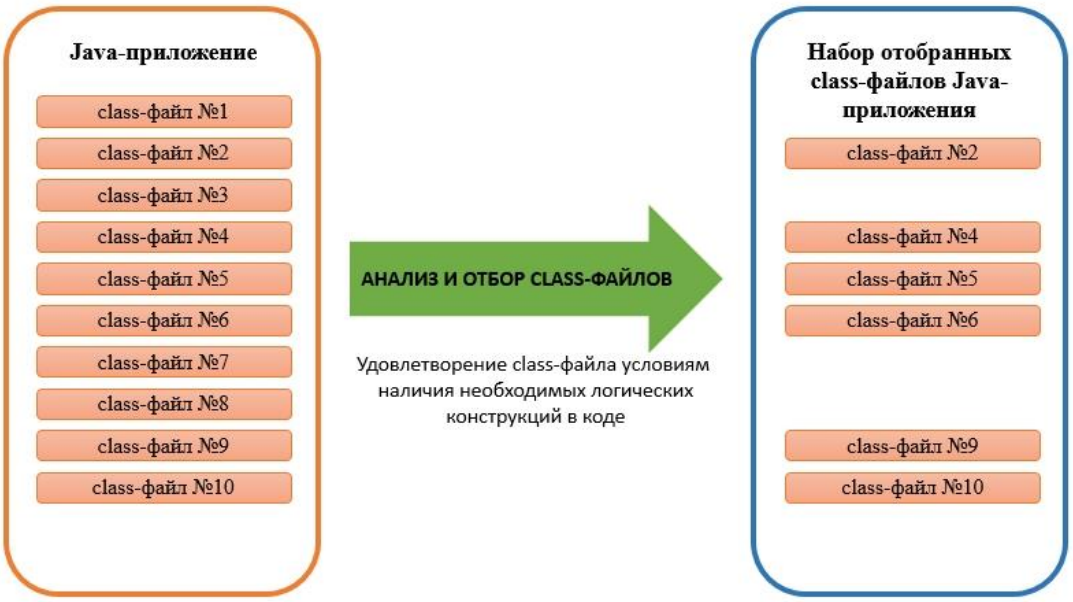


Рисунок 2.2. – Анализ и отбор class-файлов java-приложения по условию наличия необходимых логических конструкций в коде

Анализ возможно производить в ручном режиме, что потребует больших затрат ресурсов, а также воспользовавшись предоставленным ранее алгоритмом работы программы автоматического анализатора class-файлов java-приложения.

Таким образом, необходимо однозначно понимать, пригодно java-приложение или java-проект для погружения информации в class-файлы или нет, потому что, как говорилось ранее, все операции направленные на вложение информации в байт-код class-файлов не имеют возможности работы с большими объемами информации, в следствии, специфики и архитектуры исполняемых файлов.

Действие 2. Определение минимального объема вложения.

Также, для того, чтобы обеспечить максимальное покрытие class-файлов java-приложения, необходимо определиться с минимально допустимым объемом вложения. Данная информация будет являться отправной точкой при выборе количества информации при создании цифрового водяного знака. Так как исходя из минимально допустимого, в рамках текущего java-приложения, объема

информации к вложению, возможно произвести вложение цифрового водяного знака в большее количество class-файлов проекта.

После Действия 1, в наличии будет набор class-файлов, в которые возможно произвести вложение цифрового водяного знака. Каждый class-файл выполняет разные задачи и, следовательно, имеет различную внутреннюю логику, методы, количество переменных, количество операторов условных переходов, циклов с логическим условием. Среди неупорядоченного набора class-файлов, необходимо выявить class-файл, который имеет возможность вложения цифрового водяного знака наименьшего объема.

Например, class-файл занимающий дискового пространства меньше остальных class-файлов и содержащий внутри себя один метод. Скорее всего, среди class-файлов с данными характеристиками, будет найден файл, в который возможно вложить абсолютный минимум информации по сравнению с другими class-файлами java-приложения. Данный файл назначается маркерным и базовое информационное сообщение, на основе которого будет создаваться цифровой водяной знак, необходимо выбирать для данного файла. Процедура упорядочивания class-файлов по объему возможного вложения информации и удовлетворения условия наличия минимума необходимых логических конструкций в коде class-файла продемонстрирована на рисунке 2.3.

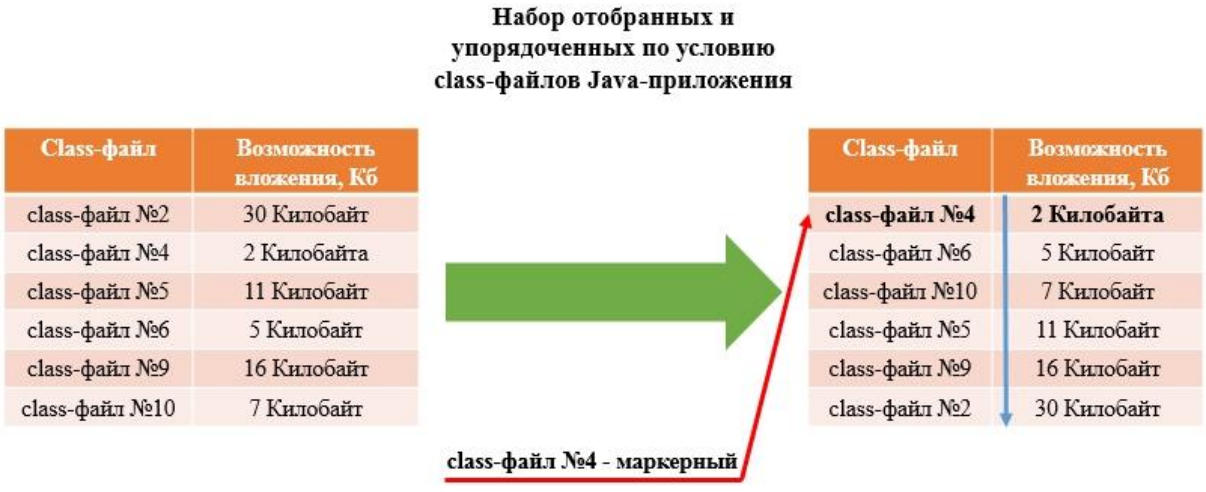


Рисунок 2.3. – Выбор маркерного class-файла по условию

Таким образом, при работе с пригодными для вложения class-файлами, будет существовать возможность вложить минимально возможный объем информации и охватить как можно большее количество class-файлов java-приложения. При наличии в class-файле более распределенной логики, а, следовательно, возможности вложения большего объема информации, существует возможность расширения базового сообщения [155].

Например, добавление к наименованию компании, которое может быть минимально возможным сообщением для скрытого вложения в class-файлы, ее реквизитов, имен учредителей или имен разработчиков ответственных за текущий class-файл, область логики java-приложения, наименование продукта, бренда, хеш-сумму class-файла и другую информацию.

Действие 3. Создание ЦВЗ.

В заключительном действии первой фазы производится создание непосредственно цифрового водяного знака с учетом полученных данных при анализе class-файлов java-проекта в действиях 1 и 2.

Учитывая class-файлы, отобранные при анализе java-проекта и вычисленный минимально возможный объем информации для вложения в каждый отобранный class-файл, производится выбор ключевого слова, которое однозначно, при проверке, позволит достоверно определить правообладателя class-файла и удовлетворяет ограничениям по объему.

Виртуальная машина Java оперирует операционными командами на уровне шестнадцатеричной системы счисления, однако, в некоторых случаях представление class-файла в HEX менее удобно для редактирования, в целях применения методики, чем в BINARY. Для удобства поиска возможных мест пересечения блоков ключевого слова и операционных команд, содержащихся в class-файле, предлагается перевод из HEX в BINARY основных опкодов, которые могут быть задействованы для вложения цифрового водяного знака.

Исходя из вышеописанного, ключевая фраза или слово выбранное в качестве цифрового водяного знака и предназначенное для вложения в class-файлы, переводится в формат представления BINARY. Так как расположение, количество и способы использования опкодов в каждом class-файле уникальны, в зависимости от его задач, то и схема вложения цифрового водяного знака в каждый class-файл java-приложения будет уникальной.

2.6.3 Вложение созданного цифрового водяного знака

После того, как действия по анализу и выборке class-файлов java-приложения, а также действия по созданию цифрового водяного знака, были успешно завершены, становится возможным приступить к фазе вложения цифрового водяного знака. Фаза вложения цифрового водяного знака разделена на два действия, выполняемые итеративно для каждого class-файла выборки.

Первое действие заключается в извлечении всех операционных команд байт-кода class-файла, которые могут использоваться при вложении цифрового водяного знака. Все операционные команды извлекаются в порядке их появления на стеке виртуальной машины Java. После чего, извлеченные операционные команды переводятся в бинарное представление. Для последующей работы с ними.

Второе действие заключается в непосредственном вложении цифрового водяного знака в байт-код class-файла. Производится перебор возможных вариантов изменения операционных команд на эквивалентные или похожие по функционалу для того, чтобы в определенной последовательности битов зашифровать биты кодового слова.

Таким образом, в момент вложения ЦВЗ в class-файл, одновременно создается схема его извлечения для **расшифровки** ключевого слова, но **не для удаления** цифрового водяного знака из class-файла. Вид схемы извлечения цифрового водяного знака из class-файла может варьироваться в зависимости от задач. В частном случае, схема извлечения представляет собой наименования рабочих опкодов, список номеров операционных команд на стеке виртуальной машины Java или наименования операционных команд для извлечения или номера

строк байт-кода для каждого конкретного class-файла. Однако схемы извлечения можно и не составлять.

Ниже, в таблице 2.1, показаны информационные значения наиболее пригодных для использования операционных команд виртуальной машины Java, которые можно редактировать и заменять на эквивалентные с целью вложения цифрового водяного знака.

Таблица 2.1 – Возможные опкоды для вложения ЦВЗ

Мнемоническое представление опкода	**Байт-код (HEX)**	**Байт-код (BINARY)**	**Возможные эквивалентые опкоды (мнемоника)**
ifeq	0x99	10011001	ifne
ifne	0x9a	10011010	ifeq
iflt	0x9b	10011011	ifle, ifgt, ifge
ifle	0x9c	10011100	iflt, ifgt, ifge
ifgt	0x9d	10011101	iflt, ifle, ifge
ifge	0x9e	10011110	iflt, ifle, ifgt
ifnull	0xc6	11000110	ifnonnull
ifnonnull	0xc7	11000111	ifnull
if_icmpeq	0x9f	10011111	if_icmpne
if_icmpne	0xa0	10100000	icmpeq
if_icmplt	0xa1	10100001	if_icmple, if_icmpgt, if_icmpge
if_icmple	0xa4	10100100	if_icmplt, if_icmpgt, if_icmpge
if_icmpgt	0xa3	10100011	if_icmplt, if_icmple, if_icmpge
if_icmpge	0xa2	10100010	if_icmplt, if_icmple, if_icmpgt
if_acmpeq	0xa5	10100101	if_acmpne
if_acmpne	0xa6	10100110	if_acmpeq
short	-	-	int, long
int	-	-	short, long
long	-	-	short, int
float	-	-	double
double	-	-	float

iadd	0x60	1100000	ladd
ladd	0x61	1100001	iadd
fadd	0x62	1100010	dadd
dadd	0x63	1100011	fadd
iload	0x1a	11010	lload
lload	0x16	10110	iload
fload	0x17	10111	dload
dload	0x18	11000	fload
istore	0x36	110110	lstore
lstore	0x37	110111	istore
fstore	0x38	111000	dstore
dstore	0x39	111001	fstore

В таблице 2.1 приведены наиболее часто встречающиеся в class-файлах опкоды, которые возможно использовать для вложения цифрового водяного знака в байт-код, однако, существует множество других, менее часто используемых или порождающих большие изменения в байт-коде class-файла. Даже с представленными в таблице 2.1, опкодами отвечающими за тип переменных в коде, необходимо производить работу крайне аккуратно, в связи с тем, что замена данных опкодов на эквивалентные существенно увеличивает размер занимаемого объема class-файлом.

Также нежелательны для использования в предлагаемой методике опкоды отвечающие за конвертацию одного типа в другой, в связи с тем, что данные опкоды порождают большое количество дополнительного кода, который практически невозможно отслеживать в автоматизированном порядке, а вручную данная процедура занимает продолжительное время.

2.6.4 Пример алгоритма на основе методики

Разработанную и описанную в блоках 2.6.1-2.6.3 методику создания и вложения цифрового водяного знака в class-файлы java, можно представить в виде алгоритма, который в дальнейшем возможно описать на языке программирования,

как полноценное приложение для автоматизированного создания и вложения цифрового водяного знака.

В первую очередь, необходимо описать, каким образом будет происходить работа с эквивалентными опкодами в байт-коде class-файла при вложении цифрового водяного знака. Рассмотрим пример на основе операторов условных переходов – ветвлений. Ветвления исходного кода могут происходить совершенно по-разному, в зависимости от условий. На рисунке 2.4 демонстрируется блок-схема представляющая собой наглядное описание алгоритма ветвления.

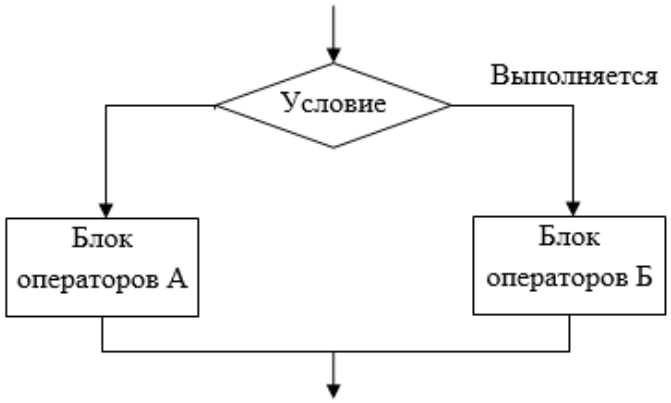


Рисунок 2.4 – Блок-схема ветвления

При истинности «Условие» будет выполнен блок «Блок операторов Б», иначе будет выполнен блок «Блок операторов А». Если при замене опкода логика блока «Условие» изменяется на противоположную, то необходимо поменять ветки условия местами.

Таким образом, задачей алгоритма будет являться поиск таких опкодов class-файла, которые пригодны для эквивалентных замен и замена которых не повлечет изменение первоначальной логики приложения, с целью получения такой последовательности опкодов, которые сформируют цифровой водяной знак. Для удобства рассмотрения, блок-схема одного из возможных алгоритмов по разработанной методике представлена в виде рисунков 2.5-2.6. Рисунок 2.5 демонстрирует алгоритм сбора данных, подготовки к вложению цифрового водяного знака и его создание. Рисунок 2.6 демонстрирует непосредственно логику

вложения цифрового водяного знака в class-файл. Однако стоит дополнительно уточнить, что блок-схемы, продемонстрированные на рисунках 2.5-2.6 разделены только для удобства, в действительности это один алгоритм.

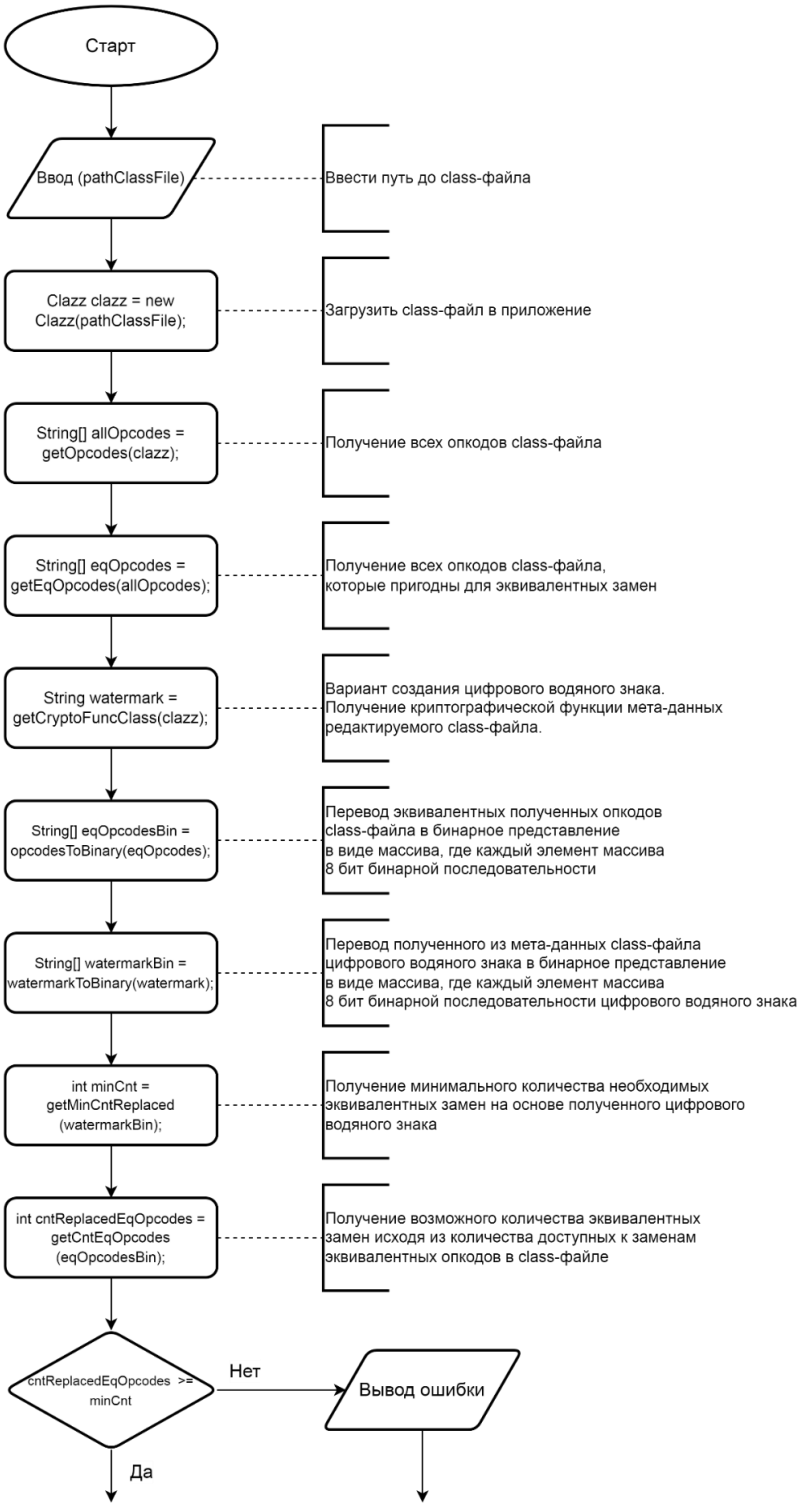


Рисунок 2.5 – Пример блок-схемы алгоритма создания и вложения цифрового водяного знака в class-файл. Часть 1.

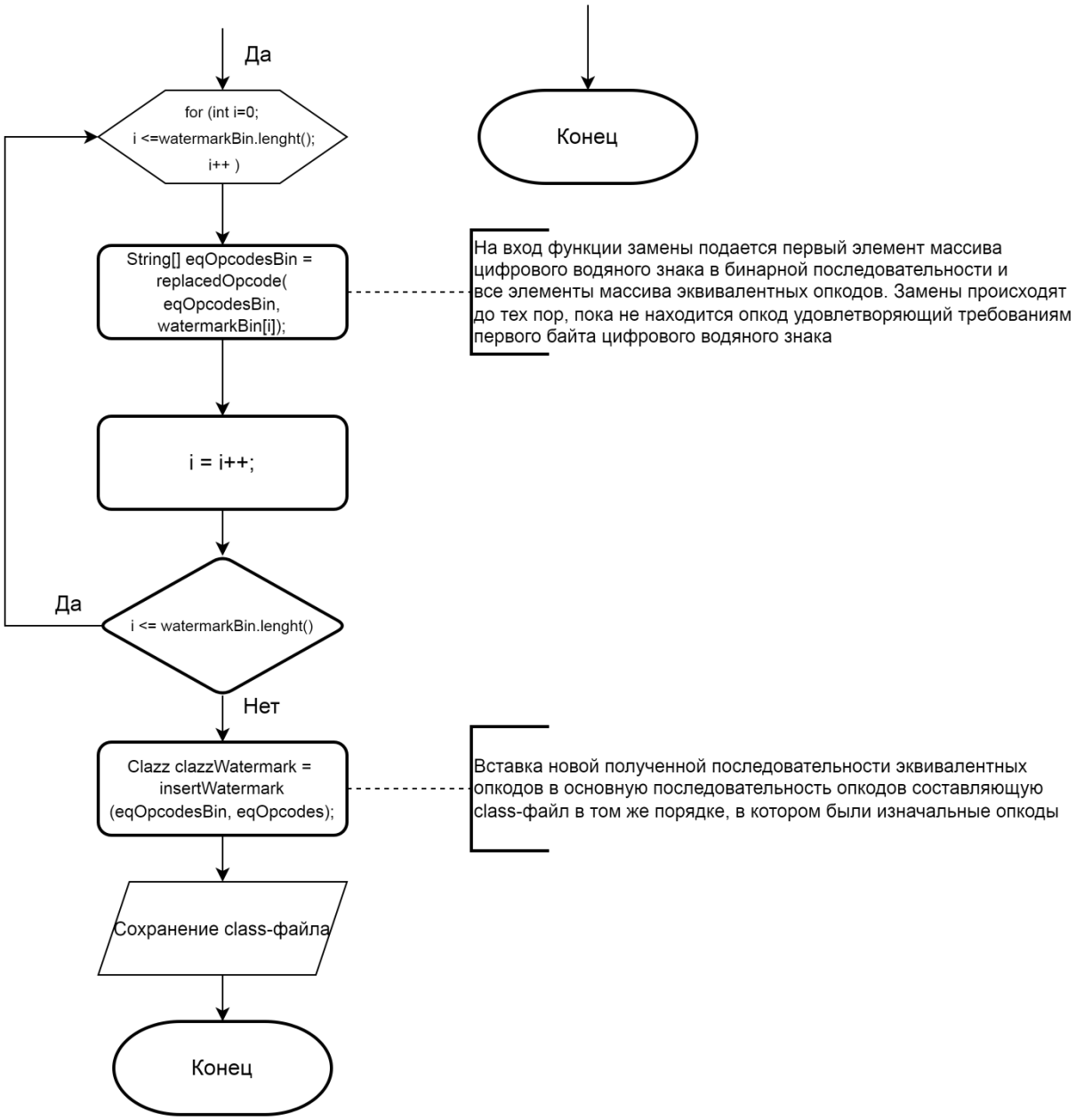


Рисунок 2.6 – Пример блок-схемы алгоритма создания и вложения цифрового водяного знака в class-файл. Часть 2.

2.7 Пример создания и скрытого вложения цифрового водяного знака в байт-код class-файла

Предположим, необходимо защитить от кражи class-файлы лабораторных работ, разработанных в рамках университетской программы для обучения

студентов. Разберем весь цикл работы с class-файлами по вложению цифрового водяного знака на примере одного. Цикл вложения цифрового водяного знака в байт-код class-файла с дальнейшей проверкой вложения состоит из нескольких последовательных этапов:

1. Анализ java-проекта
2. Анализ class-файлов
3. Создание цифрового водяного знака
4. Вложение цифрового водяного знака в class-файл
5. Чтение и расшифровка цифрового водяного знака из class-файла

Так как в данном примере мы рассматриваем вложение и проверку цифрового водяного знака только в один class-файл, который не представляет собой совокупность class-файлов, представляющих собой java-проект, то первым пунктом цикла можно пренебречь.

Исходный код class-файла, в рамках которого будут производится действия по вложению цифрового водяного знака, продемонстрирован в листинге 2.4.

Листинг 2.4 – Пример исходного кода class-файла для вложения ЦВЗ

```
public class EmbeddingWatermark {
    public static void main(String[] args) {
        long a = 1;
        long b = 2;
        long c = 3;
        float d = 4;
        float e = 5;
        long sumAb = a + b;
        long sumCb = c + b;
        float sumDe = d + e;
        if (sumAb < sumCb) {
```

```
    System.out.println("Condition 1: [sumAb < sumCb]");
} else {
    System.out.println("Condition 2: [sumCb < sumAb]");
}
        if (sumAb != sumCb) {
    System.out.println("Condition 3: [sumAb != sumCb]");
} else {
    System.out.println("Condition 4: [sumAb = sumCb]");
}
        if (sumAb >= sumCb) {
    System.out.println("Condition 5: [sumAb >= sumCb]");
} else {
    System.out.println("Condition 6: [sumAb <= sumCb]");
}
        if (sumDe != e) {
    System.out.println("Condition 7: [sumDe != e]");
} else {
    System.out.println("Condition 8: [sumDe = e]");
}
        if (sumDe == d) {
    System.out.println("Condition 9: [sumDe = d]");
} else {
    System.out.println("Condition 10: [sumDe != d]");}}}
```

Как видно из листинга 2.4, в коде происходит работа с переменными двух типов, а именно long и float, однако, присвоенные значения для данных переменных слишком малы. Также в данном коде происходит сложение переменных и работа с условиями для определения равны переменные после сложения или нет, а также какая из переданных в условие переменных имеет большее значение.

После компиляции и выполнения данного кода, закономерно в консоль будут выведены записи, содержащие в себе фразы «Condition 1», «Condition 3», «Condition 6», «Condition 7», «Condition 10», что указывает на первые, каждого из представленных операторов условных переходов, кроме последнего. Байт-код class-файла, скомпилированного из исходного кода, представленного в листинге 2.4, продемонстрирован в листинге 2.5.

Листинг 2.5 – Байт-код скомпилированного кода из листинга 2.4

```
lconst_1
lstore_1
ldc2_w 2
lstore_3
ldc2_w 3
lstore 5
ldc 4.0
fstore 7
ldc 5.0
fstore 8
lload_1
lload_3
ladd
lstore 9
lload 5
lload_3
ladd
lstore 11
fload 7
fload 8
fadd
fstore 13
```

```
lload 9
lload 11
lcmp
ifge 31
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "Condition 1: [sumAb < sumCb]"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
goto 34
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "Condition 2: [sumCb < sumAb]"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
lload 9
lload 11
lcmp
ifeq 42
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "Condition 3: [sumAb != sumCb]"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
goto 45
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "Condition 4: [sumAb = sumCb]"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
lload 9
lload 11
lcmp
iflt 53
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "Condition 5: [sumAb >= sumCb]"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
goto 56
```

```
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "Condition 6: [sumAb <= sumCb]"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
fload 13
fload 8
fcmpl
ifeq 64
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "Condition 7: [sumDe != e]"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
goto 67
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "Condition 8: [sumDe = e]"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
fload 13
fload 7
fcmpl
ifne 75
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "Condition 9: [sumDe = d]"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
goto 78
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "Condition 10: [sumDe != d]"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
return
```

После того, как появляется возможность анализировать и редактировать байт-код class-файла, необходимо начинать первый этап процесса, а именно анализ java-проекта/class-файла. Исходя из ранее изложенных данных о редактировании

определенных операционных команд байт-кода, анализируя текущий class-файл можно выделить те операционные коды, которые возможно подвергать замене на эквивалентные в данном class-файле. Операционные коды анализируемого class-файла, которые возможно подвергать замене на эквивалентные, а также их BINARY-представление продемонстрировано таблице 2.2, в порядке нахождения и вызова в class-файле [129].

Таблица 2.2 – Доступные к эквивалентной замене опкоды class-файла

Мнемоническое представление опкода	**Байт-код (BINARY)**
lconst_1 = 10 (0xa)	1010
ladd = 97 (0x61)	01100001
ladd = 97 (0x61)	01100001
fadd = 98 (0x62)	01100010
ifge = 156 (0x9c)	10011100
ifeq = 153 (0x99)	10011001
iflt = 155 (0x9b)	10011011
fcmpl = 149 (0x95)	10010101
ifeq = 153 (0x99)	10011001
fcmpl = 149 (0x95)	10010101
ifne = 154 (0x9a)	10011010

Для эффективности, процесс анализа необходимо осуществлять в автоматизированном режиме с помощью специально разработанного для данных нужд программного обеспечения [108, 149]. Также, данное программное обеспечение должно иметь возможность получать на вход, в виде команды, тип анализа для того, чтобы была возможность гибкого обхода операционных команд байт-кода class-файла. Однако так как в данном примере один class-файл, то автоматизированное программное обеспечение без надобности, анализ производится специалистом вручную. Выше, в таблице 2.2 представлен результат анализа байт-кода class-файла, продемонстрированного в листинге 2.5.

Как видно из таблицы 2.2, не все опкоды байт-кода анализируемого class-файла пригодны для эквивалентной замены. Причины такой узконаправленной выборки обсуждались выше, однако, стоит дополнительно упомянуть, что многие служебные байт-коды не подлежат редактированию и работа с ними в большинстве случаев приводит class-файл к неработоспособному состоянию. Например, в связи с этим в данную выборку не попали опкоды ответственные за работу со строковыми константами, так как сложность их редактирования без вреда для class-файла превышает полезность их редактирования для вложения цифрового водяного знака [115].

После того, как байт-код class-файла проанализирован, выбраны наиболее эффективные опкоды для эквивалентной замены и последующего вложения цифрового водяного знака, дальнейшая работа ведется с массивом бинарных сущностей, а именно бинарными представлениями всех отобранных, в рамках анализа байт-кода операционных команд в порядке их вызова во время выполнения class-файла в виртуальной машине Java. Таким образом, для автоматизированного программного обеспечения, данная часть этапа происходит в реальном времени и не занимает большого количества ресурсов, а для ручного режима это является дополнительным этапом, требующим значительных временных затрат. Результат преобразования выбранных в рамках анализа байт-кода операционных команд в бинарный вид, продемонстрирован в листинге 2.6.

Листинг 2.6 – Бинарное последовательное представление выбранных опкодов

```
1010 01100001 01100001 01100010 10011100 10011001 10011011 10010101
10011001 10010101 10011010
```

Для удобства чтения, в приведенном листинге 2.6 и в последующих, бинарное представление опкодов байт-кода разделено пробелами, по 8 бит каждое, однако, в рамках редактирования данная информация воспринимается, как цельная, без разделения пустыми символами.

После того, как:

- произведен анализ байт-кода class-файла
- отобраны возможные опкоды для проведения эквивалентных преобразований
- мнемоническое представление преобразовано в бинарное
- сформировано бинарное последовательное представление всех или только отобранных опкодов

начинается этап создания цифрового водяного знака. Цифровой водяной знак возможно сформировать случайно или на основе структуры байт-кода class-файла. Первый вариант быстрее, но гораздо менее эффективен и не устойчив к изменениям в исходном коде. В данном примере демонстрируется полный цикл создания и вложения цифрового водяного знака, поэтому цифровой водяной знак создается на основе структуры.

Допустим, необходимо произвести вложение цифрового водяного знака компании «iNK» (ink – с англ. чернила), занимающейся продажей чернил и типографских красок, в программное обеспечение для продажи товара, разработанное силами программистов данной компании. Также, необходимо указать, что товарный знак зарегистрирован – «TM».

Таким образом, после расшифровки, цифровой водяной знак должен выглядеть следующим образом: «iNK^{TM}». Данный цифровой водяной знак будет приводится к единому виду с выбранными (или всеми присутствующими) опкодами байт-кода class-файла. Каждый символ цифрового водяного знака должен быть преобразован в BINARY формат. Следовательно, необходимо произвести выбор кодировки, которая будет использоваться для сопоставления текстовых символов цифрового водяного знака и их бинарного представления. Данный цифровой водяной знак будет использовать стандартную кодировку ASCII [138]. В таблице 2.3 продемонстрировано соответствие.

Таблица 2.3 – Соответствие символов ASCII и их бинарного представления

ASCII символ	BINARY представление ASCII символа
i	01101001
N	01001110
K	01001011
™	10011001

Соответственно, при получении операционных кодов ответственных за выбранные операции в коде или всех операционных кодов class-файла, представленных в BINARY формате, представленные в таблице 2.3 биты должны присутствовать в выборке в той последовательности, в которой изначально создавался цифровой водяной знак.

В данном примере будет создаваться цифровой водяной знак на основе структуры байт-кода, так как он более показателен. Анализ байт-кода уже проведен, опкоды отобраны, бинарная последовательность получена. В силу того, что пример является упрощенным, существует малое количество опкодов, которые могут быть отобраны для эквивалентной замены. В связи с этим, цифровой водяной знак должен быть небольшого объема. Результат преобразования выбранных символов таблицы ASCII в бинарный вид, продемонстрирован в листинге 2.7.

Листинг 2.8 – Цифровой водяной знак в бинарном представлении

```
01101001 01001110 01001011 10011001
```

Исходя из имеющейся бинарной последовательности, полученной на основе отобранных для возможной эквивалентной замены опкодов байт-кода class-файла, можно увидеть, что в текущем виде в ней отсутствуют необходимые символы таблицы ASCII преобразованные в бинарный формат. В этом можно убедиться совершив поиск каждой строки таблицы 2.3 во всех строках таблицы 2.2. Принцип анализа последовательности опкодов (приведенных к бинарному виду) оригинального class-файла представлен на рисунке 2.7.

Рисунок 2.7 – Возможные символы в битовой последовательности опкодов оригинального class-файла

Из рисунка 2.7 видно, что наибольшее скопление возможных символов таблицы ASCII, которые возможно использовать, находятся в конце class-файла и в его середине, что полностью соответствует местам размещения операторов условных переходов в исходном коде, представленном в листинге 2.4. Также, достаточно большое количество возможных к использованию символов может использоваться в качестве информационного зашумления. Однако среди присутствующих необходимый символ только один – «N». Таким образом, следующим этапом вложения цифрового водяного знака будет производиться эквивалентная замена опкодов байт-кода class-файла.

Ниже в таблице 2.4, представлена последовательность замен, произведенных в байт-коде оригинального class-файла с указанием строки нахождения, заменяемого эквивалентным опкода.

Таблица 2.4 – Произведенные эквивалентные замены опкодов в байт-коде class-файла

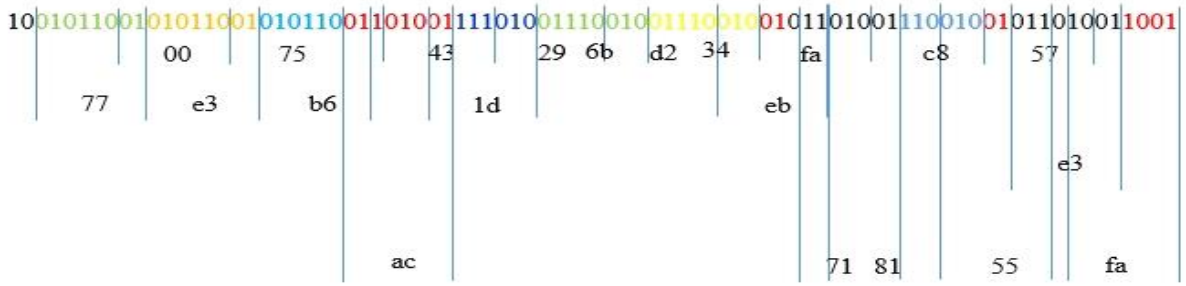
Номер строки в байт-коде class-файла	**Исходный опкод (мнемоника/hex/bin)**	**Новый опкод (мнемоника/hex/bin)**
1	lconst_1 / 0xa / 1010	lconst_0 / 0x9 /1001
13	ladd / 0x61 / 01100001	lsub / 0x65 / 01100101
17	ladd / 0x61 / 01100001	lsub / 0x65 / 01100101
21	fadd / 0x62 / 01100010	fsub / 0x66 / 01100110
26	ifge / 0x9c / 10011100	ifle / 0x9e / 10011110
37	ifeq / 0x99 / 10011001	ifge / 0x9c / 10011100
48	iflt / 0x9b / 10011011	ifge / 0x9c / 10011100
58	fcmpl / 0x95 / 10010101	fcmpg /0x96 / 10010110
59	ifeq / 0x99 / 10011001	ifge / 0x9c / 10011100
69	fcmpl / 0x95 / 10010101	fcmpg /0x96 / 10010110
70	ifne / 0x9a / 10011010	ifeq / 0x99 / 10011001

После произведенных изменений набор символов таблицы ASCII, который присутствует в бинарной последовательности выбранных опкодов оригинального class-файла изменился, что продемонстрировано на рисунке 2.8.

Рисунок 2.8 – Возможные символы в битовой последовательности опкодов оригинального class-файла после эквивалентных замен

Как видно из рисунка 2.8, распределение символов таблицы ASCII в полученной, после редактирования байт-кода, битовой последовательности отобранных опкодов, стало более нормальным и равномерно распределенным по всей последовательности бит. Равномерное распределение символов помогает запутать злоумышленника, создать сложность с выявлением главных символов в последовательности бит. Также, на рисунке 2.8 видно, что существует «запасной» байт, который также является необходимым символом «i», позволяющий получить в одной последовательности два одинаковых цифровых водяных знака за счет разного расположения символов «i» в последовательности. Оба цифровых водяных знака являются набором заданных бит, в правильной последовательности, которые при преобразовании бит в символы таблицы ASCII дают ранее описываемый цифровой водяной знак «iNK™».

Также, необходимо отметить, что в данном исследовании цифровой водяной знак, созданный и вложенный в class-файл с помощью разработанной методики, специально упрощен, с целью демонстрации эффективности методики. Однако методика позволяет производить вложение цифрового водяного знака и большей сложности в class-файлы. Например, на рисунке 2.9 продемонстрирован пример вложения криптографической хеш-функции [22, 158].



77 00 e3 75 b6 ac 43 1d 29 6b d2 34 eb fa 71 81 c8 dd 55 57 e3 fa

Рисунок 2.9 – Вложение ЦВЗ в виде криптографической хеш-функции

Расшифровка цифрового водяного знака происходит посредством тех же этапов в обратном порядке:

- Специалист или программа-анализатор с заранее загруженными требованиями (какие опкоды искать, выбранная кодировка для символов цифрового водяного знака) производит анализ байт-кода class-файла
- Производится отбор необходимых опкодов по заданному условию, например, опкоды отвечающие за операторы условного перехода и только они
- Составляется битовая последовательность из данных опкодов, которая соответствует порядку вызову опкода в class-файле
- Переводится в бинарный вид цифровой водяной знак
- Осуществляется поиск каждого символа цифрового водяного знака в полученной бинарной последовательности
- Проверка соответствия цифрового водяного знака

Метрика Джилба для оценки сложности class-файла. Так как создание и вложение цифровых водяных знаков осуществляется на основе эквивалентных замен опкодов байт-кода, то метрика Джилба для операторов условных переходов, безусловных переходов и циклов может применяться для сравнения class-файла со стеговложением и без.

Таким образом, логическая сложность java-приложения будет определяться как насыщенность его байт-кода опкодами отвечающими за циклы и условные переходы. Конструкции вида: *if_icmpeq, if_icmpne, tableswitch, lookupswitch, goto_w, jsr, jsr_w* и другие.

Введем характеристику C_L, как абсолютную сложность java-приложения, характеризующуюся количеством циклов, операторов условных и безусловных переходов в нем. Также, введем характеристику c_l, как относительную сложность java-приложения, характеризующаяся количеством опкодов циклов, условных и безусловных переходов, т.е. c_l будет определяться, как отношение C_L к общему числу опкодов class-файла или java-приложения.

$$c_l = \frac{C_L}{N},$$

где N – общее количество операторов программного кода.

Следовательно, для class-файла или java-приложения без стеговложения q_1 и class-файла или java-приложения со стеговложением q_2, результаты сравнения на основе метрики Джилба должны быть идентичны. Class-файл со стеговложением и без него, должны иметь одинаковую относительную сложность.

$$c_l(q_1) = c_l(q_2)$$

Метрика Чепина для оценки информационной прочности отдельно взятого class-файла. Вложение цифрового водяного знака в class-файлы java-приложения за счет изменений, которые производятся с данным class-файлом. Разумно предположить, что если после вложения цифрового водяного знака class-файл не изменился в количестве команд, опкодов в объеме, в отображении исходного кода или интерфейсов, вычислительно и функционально, то возможно изменения затронули такие характеристики, как прочность.

У любого class-файла или java-приложения есть входные и выходные параметры, переменные, которые разбиваются на 4 группы:

- Входные переменные (P), которые используются в качестве ввода информации в java-приложение,
- Локальные или создаваемые и редактируемые в процессе работы java-приложения переменные (M),
- Управляющие переменные (C) те, которые участвуют в управлении логикой работы java-приложения,
- Переменные «мертвого кода» или не используемые (T).

Для каждой части метрики имеется свое значение, которое подставляется в итоговое выражение.

$$Q = a_1 P + a_2 M + a_3 C + a_4 T\ ,$$

где a_1, a_2, a_3 и a_4 являются весовыми коэффициентами, которые используются для отражения влияния каждой группы переменных на сложность java-

приложения. Самый наибольший вес, равный значению 3, имеет группа управляющих переменных (C) в связи с сильным влиянием на поток управления java-приложения. Остальные коэффициенты распределены следующим образом: a_1=1; a_2=2; a_3=0.5. Таким образом, учитывая весовые коэффициенты выражение принимает вид:

$$Q = P + 2M + 3C + 0{,}5T.$$

У class-файла со стеговложением не должен измениться результат вычисления каждой группы переменных по сравнению с class-файлом без стеговложения, соответственно, итоговое значение также должно остаться неизменным.

Соотношение объема class-файла к объему вложения

Как было описано выше, формат исполняемых файлов менее пригоден для вложения больших объемов информации, в сравнении с медиа-форматами. Объем допустимого вложения, который позволит скрытно произвести вложение цифрового водяного знака в структуру байт-кода class-файла, не нарушив его работоспособность, зависит от количества строк исходного кода, количества операционных команд, подлежащих эквивалентным заменам и в среднем, варьируется в диапазоне 0,2-3% от общего объема class-файла.

Class-файл, используемый в данном примере, имеет следующие характеристики:

- Объем: 1006 байт (8048 бит)
- Количество строк кода (LOC): 36
- Количество операционных команд (строк) в байт-коде: 78
- Количество допустимых к безопасному редактированию/эквивалентной замене опкодов: 11

Цифровой водяной знак, используемый в данном примере, имеет следующие характеристики:

- Количество символов: 4
- Объем цифрового водяного знака: 4 байта (32 бита)
- Количество произведенных эквивалентных замен для вложения цифрового водяного знака: 11

Виртуальная машина Java оперирует операционными командами – опкодами. Один опкод представляет собой одну инструкцию для JVM. Таким образом, количество опкодов в class-файле определяет количество действий, которые затратит виртуальная машина Java для исполнения данного class-файла. Один операционный код байт-кода представляет собой один байт. Следовательно, число возможных перестановок операционных кодов в class-файле равно N!, где N – число операционных кодов виртуальной машины Java пригодных для эквивалентных замен, присутствующих в байт-коде данного class-файла. Соответственно, число байт class-файла, доступных к замене, таким образом равно:

$$\log_2(N!) = N \log_2 N$$

Однако не все операционные команды пригодны для эквивалентных замен, а, следовательно, не все class-файлы могут содержать опкоды пригодные для эквивалентных замен. Таким образом, число возможных эквивалентных замен в java-приложении равно $\sum_{i=1}^{n} \mathrm{Oi!}$, где n – число class-файлов, содержащих пригодные для эквивалентных замен опкоды, O_i – число опкодов пригодных для эквивалентных замен в class-файле. Следовательно, число опкодов к использованию для эквивалентных замен и вложения ЦВЗ равно:

$$\log_2(\sum_{i=1}^{n} O_i!)$$

Таким образом, произведя расчеты, получив 11 опкодов для возможных эквивалентных замен, произведя вложение ЦВЗ объемом 4 байта за 11 эквивалентных замен, возможно установить соотношение цифрового водяного знака и class-файла из примера, описанного в данном блоке, является равным 2 к 503. В процентном соотношении: 0,397 ~ 0,4%.

В данном примере, производилась работа с узкой выборкой опкодов байт-кода на основе которых формировалась бинарная последовательность. Продемонстрирован упрощенный пример, включающий в себя, работу только с отобранными опкодами, однако в автоматическом режиме и на больших проектах с большим количеством class-файлов, будут задействованы бинарные представления всех опкодов содержащихся в байт-коде каждого class-файла для того, чтобы полноценно устранить риск удаления цифрового водяного знака, при переносе блоков кода, содержащих в себе операторы условного перехода, в другое место исходного class-файла. Таким образом, в бинарное представление будут переведены все опкоды байт-кода class-файла, а работа по эквивалентной замене будет по-прежнему производится на определенной выборке опкодов. Такой подход повысит устойчивость против атак обфускацией [154].

Выводы

Раздел содержит основные научные результаты, полученные в работе и направленные на разрешение актуальной научной задачи исследования.

1) Проведен анализ структуры class-файла, спецификации виртуальной машины Java и ее взаимосвязей основных операционных байт-кодов. Обоснован выбор class-файлов в качестве контейнера для вложения цифрового водяного знака. Показано, что у определенных типов операционных команд байт-кодов существует между собой эквивалентность, что позволяет производить эквивалентные замены в байт-коде class-файла. Рассмотрены возможнные вариации эквивалентных замен опкодов и результаты таких замен. Показано, что возможно использование расширенного набора опкодов для эквивалентных замен, что позволяет произвести

большее количество эквивалентных замен в class-файле. Результаты этих исследований опубликованы в работах [31, 34, 35].

2) В качестве научного результата, интегрально объединяющего вышеизложенные результаты, и позволяющего обеспечить достижение цели исследования была разработана методика создания и вложения цифрового водяного знака увеличенного объема в байт-код class-файлов за счет использования расширенного набора опкодов для эквивалентных замен. Увеличение объема цифрового знака осуществляется путем большего количества эквивалентных замен в одном class-файле. К элементам научной новизны данной методики относится следующее:

а) для вложения цифрового водяного знака используется структура байт-кода class-файла, а не сепциально добавленные методы, что позволяет цифровому водяному знаку являться не отделяемой частью class-файла;

б) для вложения цифрового водяного знака используется расширенный набор операционных команд пригодных для эквивалентных замен;

Данная методика и ее отдельные элементы опубликованы в работах [31, 35, 41, 153, 154, 155].

Разработанная в главе 2 методика создания и скрытого вложения цифрового водяного знака увеличенного объема в байт-код class-файла основанная на использовании дополнительных наборов операционных кодов байт-кода. Методика в отличие от известных, позволяет производить скрытое вложение цифрового водяного знака увеличенного объема в байт-код class-файлов на 60-80%. Методика доведена до реализации алгоритма программы для ЭВМ.

Разработанная методика:

- Позволяет производить увеличенного объема вложение в исполняемые class-файлы
- Не нарушать логику работы исполняемых class-файлов во время вложения ЦВЗ
- Не увеличивает объем class-файлов
- Возможность вложения, тиражируемого цифрового водяного знака

- Возможность автоматизации процесса

За счет проведенных экспериментов продемонстрировано, что объем успешно вложенного цифрового водяного знака в class-файл может составлять 0,4 - 5,7% от общего объема class-файла, что в среднем на 30% больше по сравнению с известными методиками и зависит от class-файла.

Материалы главы 2 были представлены в материалах международных и российских научно-технических конференциях [36, 44, 153, 154, 155] и опубликованы в изданиях, включенных в перечень ВАК России [31, 35, 41, 47], а также в изданиях Scopus [153, 154, 155]. Статьи «Методика нахождение величины наиболее выгодного контейнера в форматах исполняемых файлов» и «Методика создания и скрытого вложения цифрового водяного знака в байт-код class-файла на основе не декларированных возможностей виртуальной машины Java» опубликованы в изданиях, включенных в перечень ВАК России, без соавторства.

Результатом работы, проведенной в главе 2, является разработанная автором программа для ЭВМ «Модификатор байт-кода java-программы для скрытого вложения цифрового водяного знака посредством автоматического редактирования байт-кода class-файла», номер регистрации 2020664301 от 11.11.2020 г., копия свидетельства о регистрации программы для ЭВМ представлена в приложении А.

Положения главы 2 настоящего диссертационного исследования были внедрены в учебный процесс курса «Разработка защищенных приложений» в СПБГУТ им. проф. М.А. Бонч-Бруевича.

По результатам главы 2 подготовлено учебно-методическое пособие «Разработка защищенных приложений» (Россия, Санкт – Петербург, СПБГУТ им. проф. М.А. Бонч-Бруевича).

Глава 3. Методика создания и вложения устойчивого к атакам декомпиляцией цифрового водяного знака в class-файлы java-приложения

3.1 Модель нарушителя java-приложения

Существует большое количество java-приложений, которые являются цельными исполняемыми файлами формата jar и поставляемыми пользователю. Иными словами, java-приложение представляет собой набор class-файлов взаимосвязанных друг с другом и представленных в виде единого исполняемого jar-файла. Таким образом, java-приложение, которым является jar-файл, передается конечному пользователю полностью. Обычно java-приложение содержит в себе все, что ему необходимо для запуска и работы – class-файлы, файлы конфигураций внутренних вспомогательных библиотек, описание экранных форм и другие составляющие полноценного десктопного приложения (программы для ЭВМ).

Исходя из вышесказанного следует, что злоумышленник, который получил java-приложение, как пользователь, при необходимых знаниях и подготовке, получает доступ ко всему исходному коду приложения, которые содержится в class-файлах java-приложения. Данная ситуация позволяет злоумышленнику получить исходный код java-приложения, внести в него изменения, заменить экранные формы, настроить файлы конфигураций на свои банковские счета и выдавать результат за собственную разработку, возможно, с последующей перепродажей третьим лицам.

Таким образом, поднимается проблема защиты авторских прав на разработанное программное обеспечение на языке программирования Java. Кроссплатформенность Java, при наличии инструментов, позволяет произвести декомпиляцию class-файлов даже не подготовленному пользователю, тем самым получив доступ к исходному коду [4]. После чего, произведя минимальные изменения интерфейса и файлов конфигураций, скомпилировать его по новой, получив «совершенно другое java-приложение».

Одной из основных проблем создания и вложения цифровых водяных знаков в байт-код class-файлов java-приложения является то, что язык программирования Java, его компилятор и виртуальная машина Java, постоянно обновляются и дорабатываются авторами. В силу различных сложных связей и изменений, нередко возникают ситуации, когда существует вероятность выбора компилятором Java отличных друг от друга или эквивалентных операционных команд при интерпретации исходного кода в байт-код при использовании различных версий компилятора. При этом функционал и внутренняя логика java-приложения остается без изменений, и работа происходит по заданному разработчиком алгоритму. Таким образом, большинство существующих цифровых водяных знаков возможно уничтожить еще на этапе перекомпиляции java-приложения. Возможная модель действий злоумышленника представлена на рисунке 3.1.

Рисунок 3.1 – Возможная модель действий нарушителя

Таким образом, злоумышленник предполагает, что своими действиями повреждает или уничтожает цифровой водяной знак, находящийся в class-файлах [40]. После минимальных изменений class-файлов подверженных декомпиляции, злоумышленник компилирует их другой версией компилятора JDK [31], в

зависимости от той, которую он смог узнать из class-файлов. Финальным этапом является архивирование в исполняемый jar-файл и использование полученного java-приложения в любых целях.

В связи с вышеизложенной проблематикой защиты авторских прав на java-приложения, необходима разработка методики, которая позволяет создать и вложить цифровой водяной знак, который будет, помимо подтверждения авторства, устойчив к направленным атакам декомпиляцией [34].

3.2 Методика создания и вложения цифрового водяного знака в class-файлы java-приложения устойчивого к атакам декомпиляцией

Разработанная методика охватывает весь цикл разработки программного обеспечения в виде java-приложения и представляет собой ряд шагов, предпринимаемых на различных этапах разработки java-приложения. Разработанная методика состоит из трех этапов:

- Подготовительный этап
- Этап создания цифрового водяного знака
- Этап вложения цифрового водяного знака

Разработанная методика с разбиением этапов на действия представлена на рисунке 3.2.

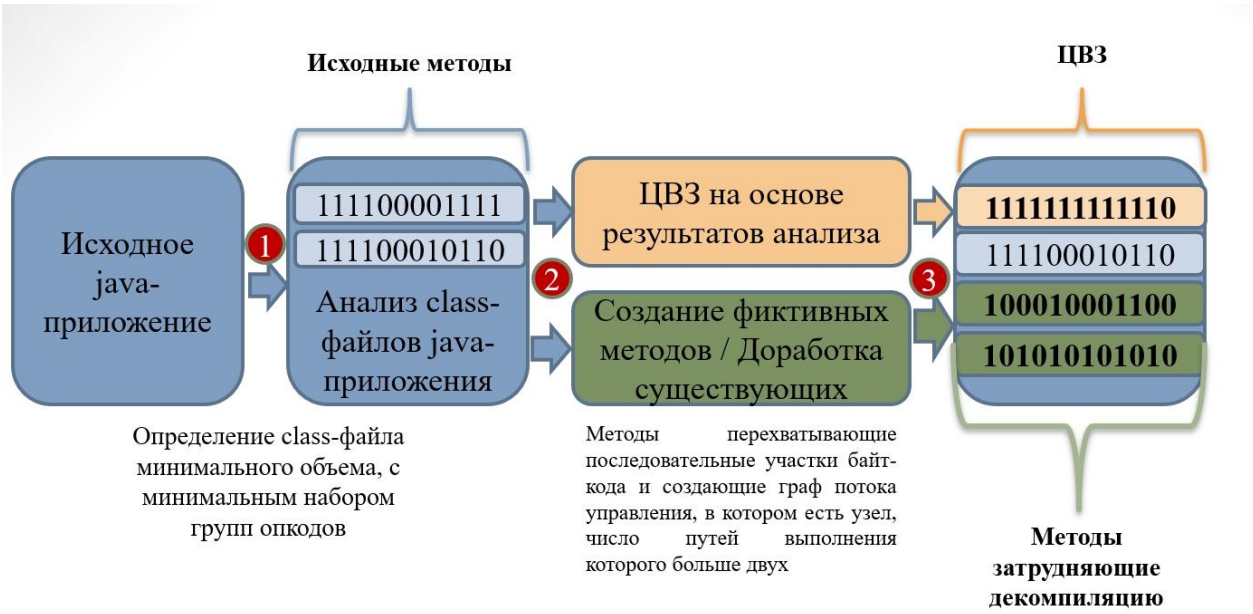


Рисунок 3.2 – Методика создания и вложения ЦВЗ в class-файлы java-приложения

Подготовительный этап разработанной методики включает:

- Анализ class-файлов java-приложения
- Выявление наименьшего по объему и количеству возможных к заменам операционных кодов class-файла
- Выделение class-файлов, содержащих в себе ключевую логику java-приложения
- Внесение в class-файлы, содержащих ключевую логику java-приложения, функций, которые из-за специфики представления и интерпретации их в байт-коде не могут быть декомпилированны логически верно, что нарушает работоспособность декомпилируемого class-файла (данное действие, возможно применять в момент написания исходного кода java-приложения, чтобы в последствии не добавлять фиктивные функции, а иметь работающие функции, встроенные в процесс работы java-приложения и заранее написанные в нужной парадигме). Действия подготовительного этапа методики продемонстрированы на рисунке 3.3.

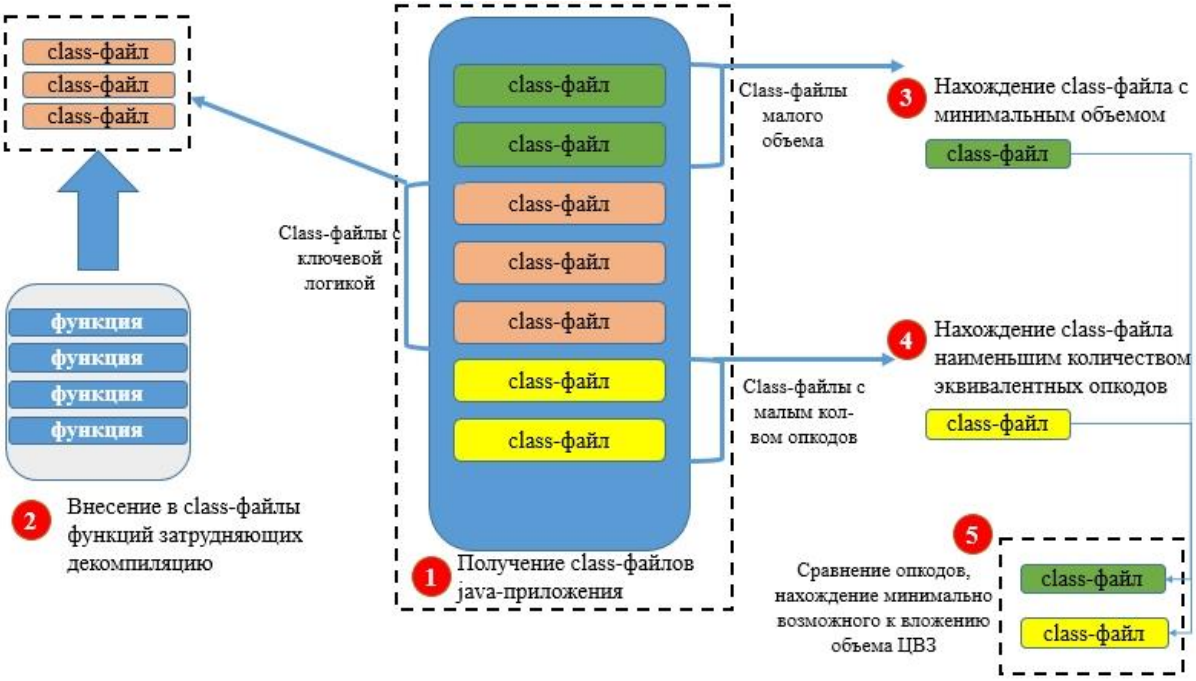


Рисунок 3.3 – Действия подготовительного этапа методики

Этап создания цифрового водяного знака включает:

- Сбор данных о необходимом виде ЦВЗ: криптографическая хеш-функция, текстовые данные, представляющие наименование компании и т.д.
- При необходимости: доработка существующих методов или создание новых фиктивных, использующих в качестве логики, возможности языка программирования Java, сложно поддающиеся декомпиляции или повышающие риск ошибки обработки class-файла декомпилятором
- Проверка минимально возможного количества раз, которое можно вложить цифровой водяной знак в class-файл минимального объема в java-приложении
- Проверка минимально возможного количества раз, которое можно вложить цифровой водяной знак в class-файл с минимальным количеством возможных к эквивалентным заменам операционных кодов
- Выбор вида ЦВЗ наиболее удовлетворяющему текущему java-приложению

- Перевод криптографической хеш-функции / текстовой информации в двоичное представление с разделением на битовые блоки.

Действия этапа создания цифрового водяного знака разработанной методики продемонстрированы на рисунке 3.4.

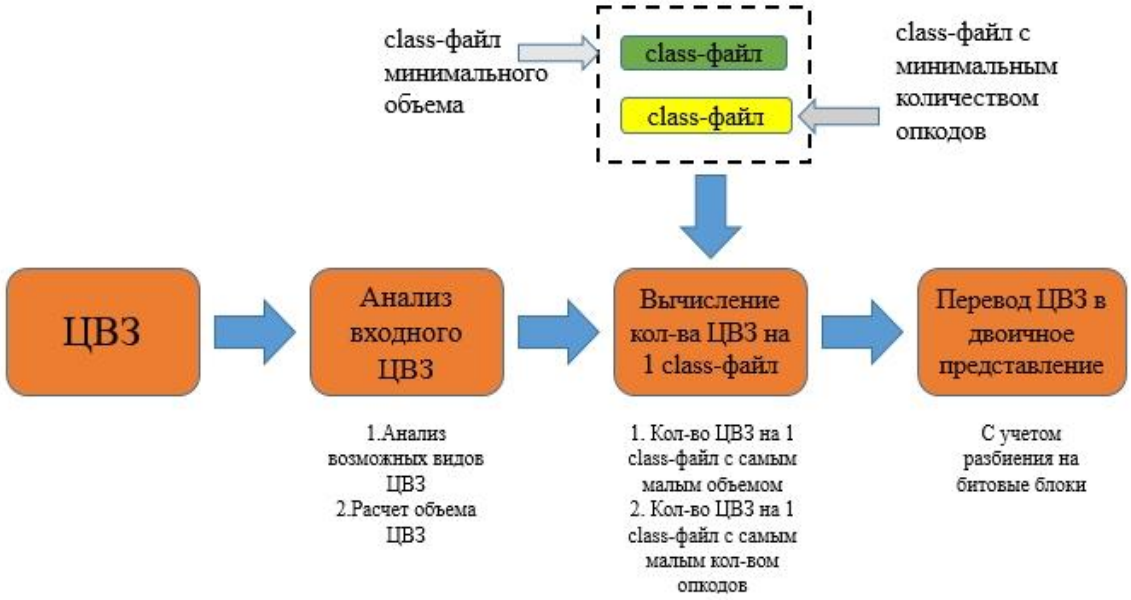


Рисунок 3.4 – Действия этапа создания цифрового водяного знака

Этап вложения цифрового водяного знака включает:

- Циклический обход class-файлов java-приложения с целью редактирования байт-кода каждого class-файла
- Если код не был создан с использованием функций языка программирования Java затрудняющих декомпиляцию, на данном этапе в class-файлы внедряется данный функционал. Посредством создания фиктивных методов или редактирования существующих методов в class-файле без изменения их внутренней логики, но с изменениями используемых практик и функций языка программирования Java для реализации данной логики
- Редактирование опкодов байт-кода на эквивалентные происходит при соблюдении критериев для ЦВЗ

- Эквивалентные замены производятся до тех пор, пока не достигается уникальная комбинация битовой последовательности опкодов class-файла, которая содержит в себе максимальное количество тиражируемо вложенных копий цифрового водяного знака, полученного на предыдущем этапе.

В упрощенном виде, действия этапа вложения цифрового водяного знака продемонстрированы на рисунке 3.5.

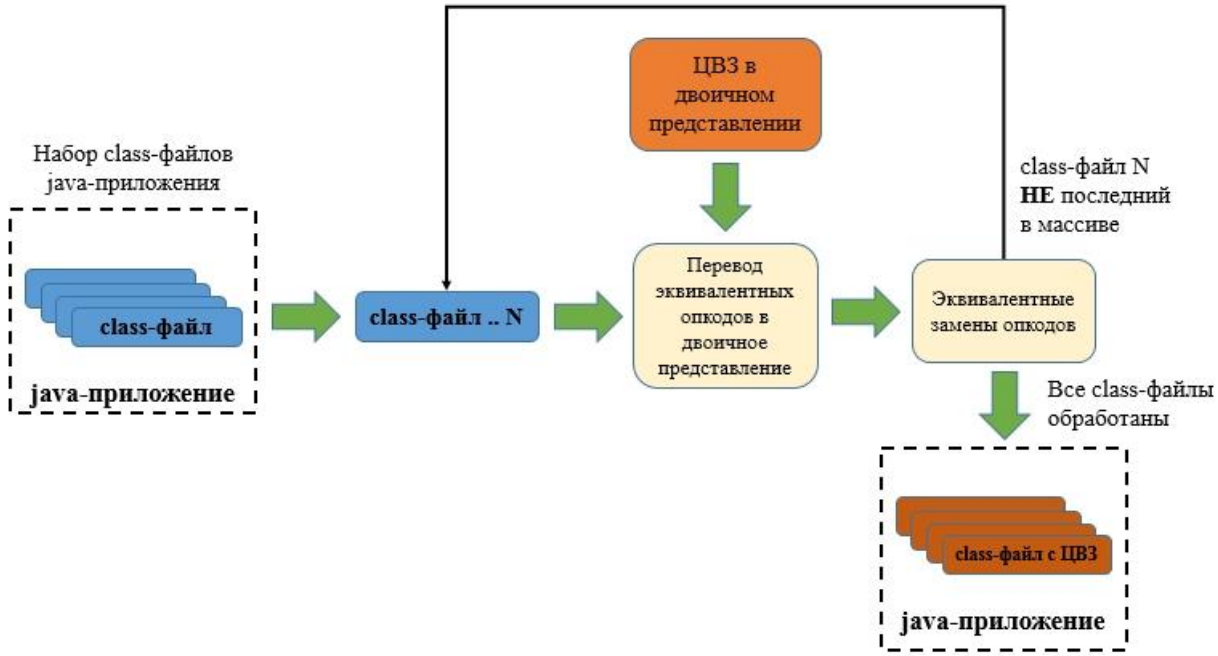


Рисунок 3.5 – Действия этапа вложения цифрового водяного знака

Таким образом, совокупность эквивалентных замен нескольких групп опкодов и паттернов, функций программирования не однозначно декомпилируемых или декомпилируемых неверно, полностью покрывает риск уничтожения или повреждения цифрового водяного знака посредством атак перекомпиляцией [46]. Также дополнительным фактором являются методы, препятствующие декомпиляции, вызывающие ошибку в работе декомпилятора или получение некорректного исходного кода после работы декомпилятора.

Также, разработанная методика доведена реализации алгоритма программы.

3.2.1 Пример алгоритма на основе методики

В связи с тем, что разработанная методика предполагает глубокий анализ class-файлов java-приложения, разделение class-файлов на выборки для различных действий методики и вложение цифрового водяного знака в каждый class-файл в зависимости от количества в нем опкодов, то сложность алгоритма будет выше, по сравнению с алгоритмом из Главы 2, следовательно, блок-схема подобного алгоритма объемней.

Таким образом, возможный пример алгоритма построенного на основе разработанной методики будет разделен на несколько блок-схем для удобства разбора и демонстрации отдельных блоков, которые могут быть вынесены, как самостоятельные алгоритмы. Однако все представленные блок-схемы следует воспринимать, как демонстрацию алгоритмов каждого отдельного модуля, которые в своей совокупности являются одним алгоритмом разработанным на основе шагов методики, процесс выполнения которого является единым.

Неотъемлемой частью алгоритма является работа с базой данных которая будет хранить все необходимые сущности во время работы алгоритма. Таким образом, первым шагом в разработке алгоритма, будет являться разработка структуры базы данных для хранения class-файлов, цифровых водяных знаков, их представлений. Разработанная структура базы данных представлена на рисунке 3.6.

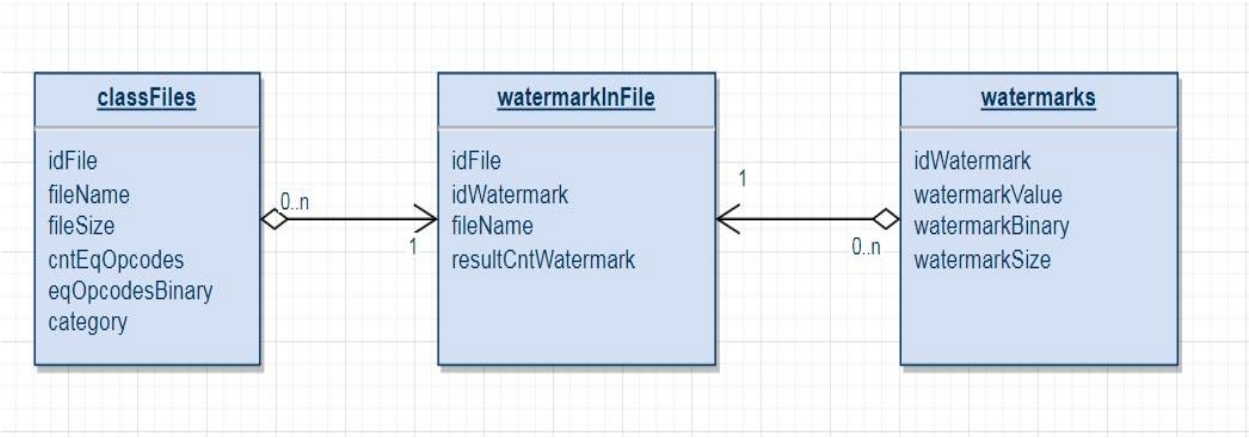


Рисунок 3.6 – Структура базы данных

Разработанная база данных состоит из 3 таблиц:

- **classFiles** – данная таблица отвечает за все обработанные алгоритмом class-файлы java-приложения. Также в ней хранится характеристика данных файлов:

уникальный id class-файла, наименование, размер, количество эквивалентных опкодов, эквивалентные опкоды переведенные в бинарное представление, категория class-файла (ключевая логика, минимальный объем, минимальное кол-во эквивалентных опкодов, общий)

- **watermarks** – данная таблица отвечает за хранение цифровых водяных знаков, вложение которых предполагается в class-файлы java-приложения. Таблица хранит характеристики цифровых водяных знаков: уникальный id цифрового водяного знака, значение, бинарное представление, объем цифрового водяного знака (в битах)

- **watermarkInFile** – ключевая таблица, связывающая две предыдущие. Столбцы **idFile** и **fileName** обеспечивают связь с таблицей **classFiles,** а столбец **idWatermark** обеспечивает связь с таблицей **watermarks.** Столбец **resultCntWatermark** хранит в себе количественный результат вложения цифрового водяного знака в class-файл, демонстрирующий количество цифровых водяных знаков, которые удалось вложить в class-файл с помощью алгоритма.

Алгоритм разделен на три условных блока, которые представлены каждый своей блок-схемой. Каждая блок-схема представляет один условный блок алгоритма и взаимодействует с одной из таблиц в базе данных. Пример возможной блок-схемы на основе методики демонстрирующей работу с class-файлами представлена на рисунке 3.7.

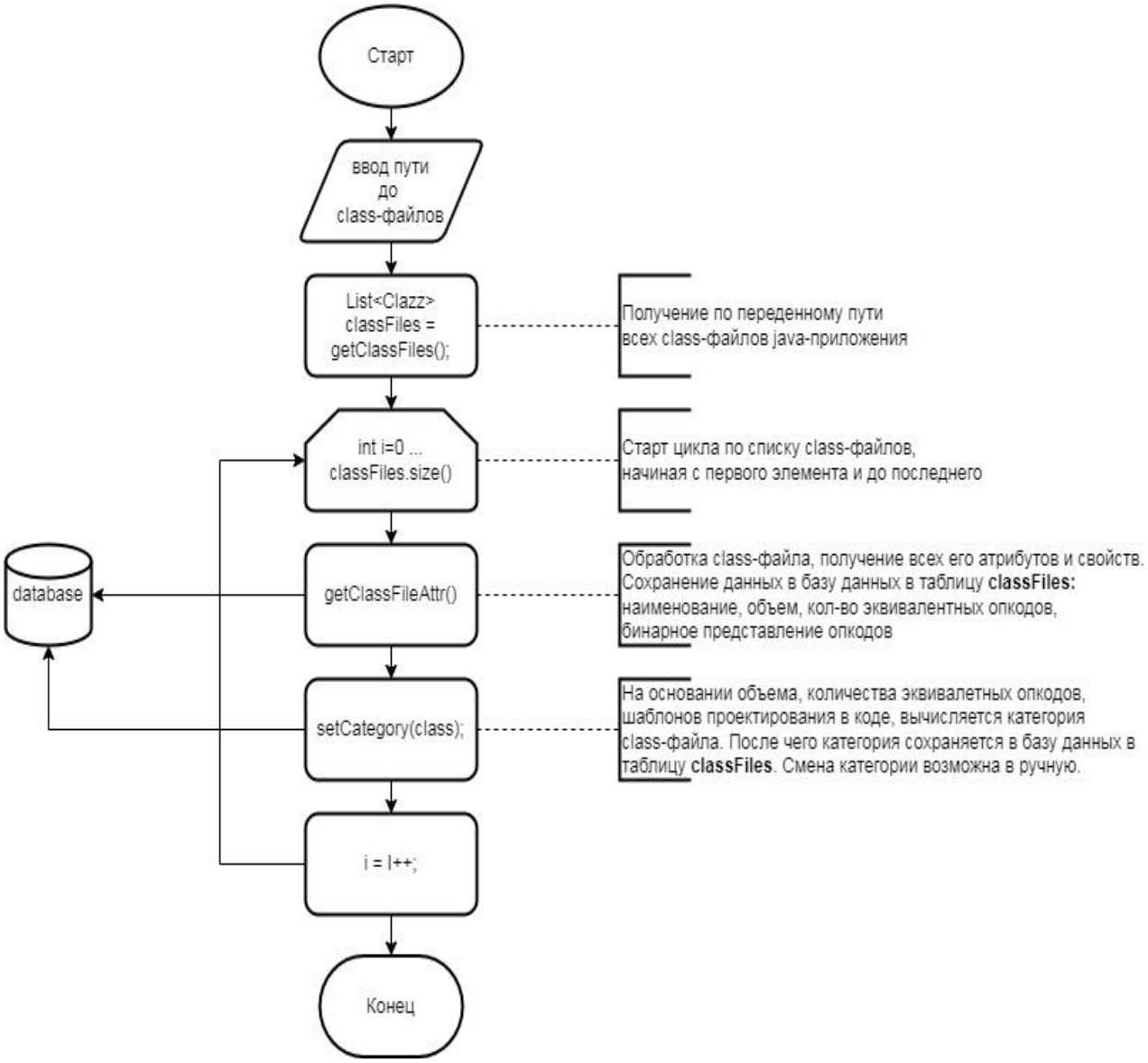


Рисунок 3.7 – Пример блок-схемы части алгоритма анализа class-файлов java-приложения

Блок-схема, продемонстрированная на рисунке 3.7 описывает анализ class-файлов java-приложения. По переданному пути, алгоритм в цикле анализирует каждый найденный class-файл java-приложения. Анализируется объем class-файла, количество опкодов, которые он содержит, количество эквивалентных опкодов, а также их бинарное представление. Также, на основе полученных данных class-файлу присваивается категория. После этого все данные полученные посредством анализа сохраняются во внутреннюю базу с целью дальнейшего повторного использования.

Второй частью алгоритма являются функции, осуществляющие использование в полученных class-файлах с ключевой логикой, которые были промаркированы в базе данных в таблице **classFiles** в столбце **category** ранее, с целью использования функций, затрудняющих декомпиляцию в class-файлах. Производится перебор всех функций с анализом возможности их использования в конкретном class-файле, после чего осуществляется переход к следующему class-файлу. После завершения алгоритма, производятся записи в базу данных. Блок-схема демонстрирующая пример работы алгоритма на основе разработанной методики, представлена на рисунке 3.8.

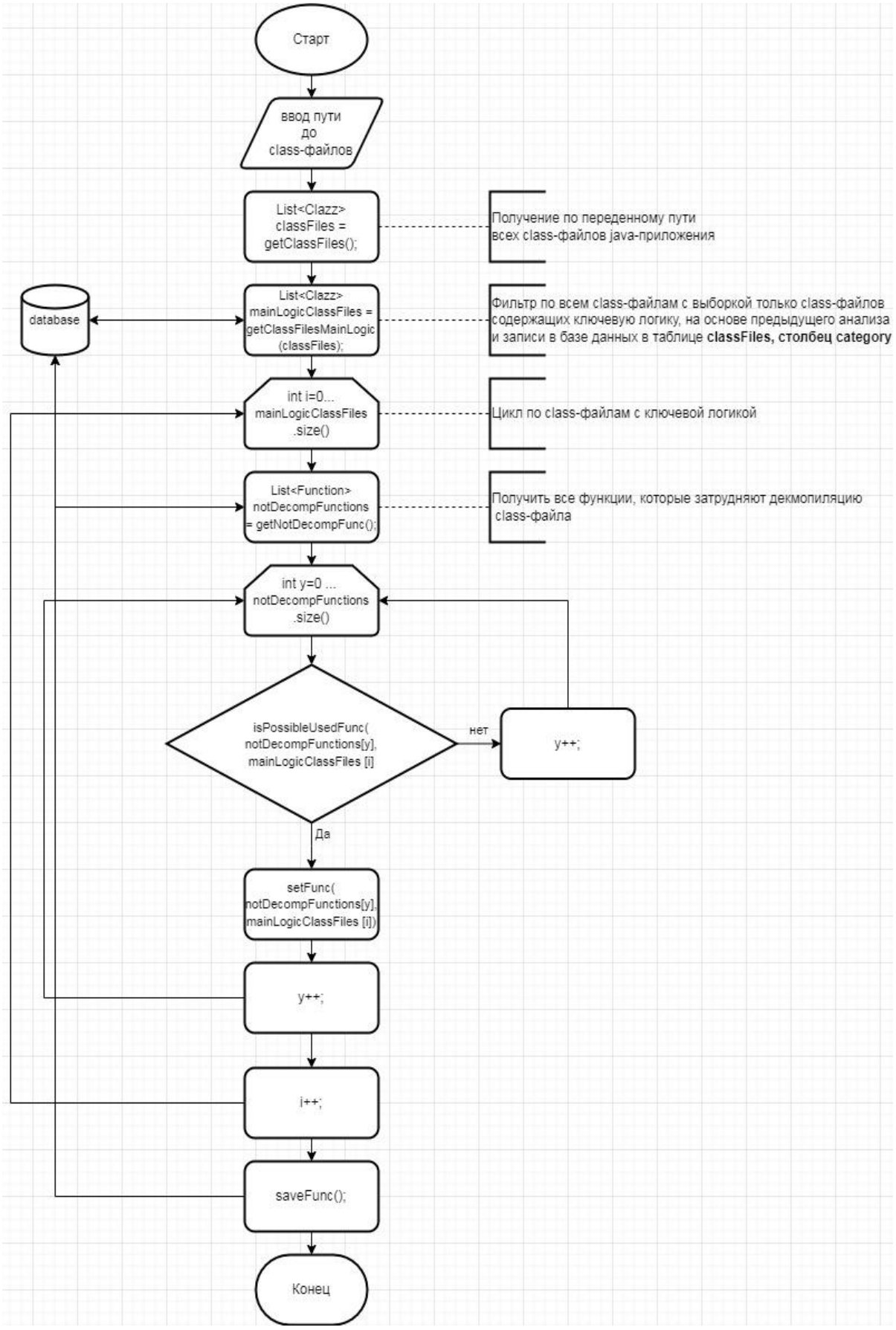


Рисунок 3.8 – Пример блок-схемы части алгоритма реализации затрудняющих декомпиляцию функций в class-файлах java-приложений

На рисунке 3.9 продемонстрирована завершающая часть алгоритма на основе разработанной методики в виде блок-схемы, реализующая функцию вложения цифрового водяного знака в байт-код class-файлов java-приложения.

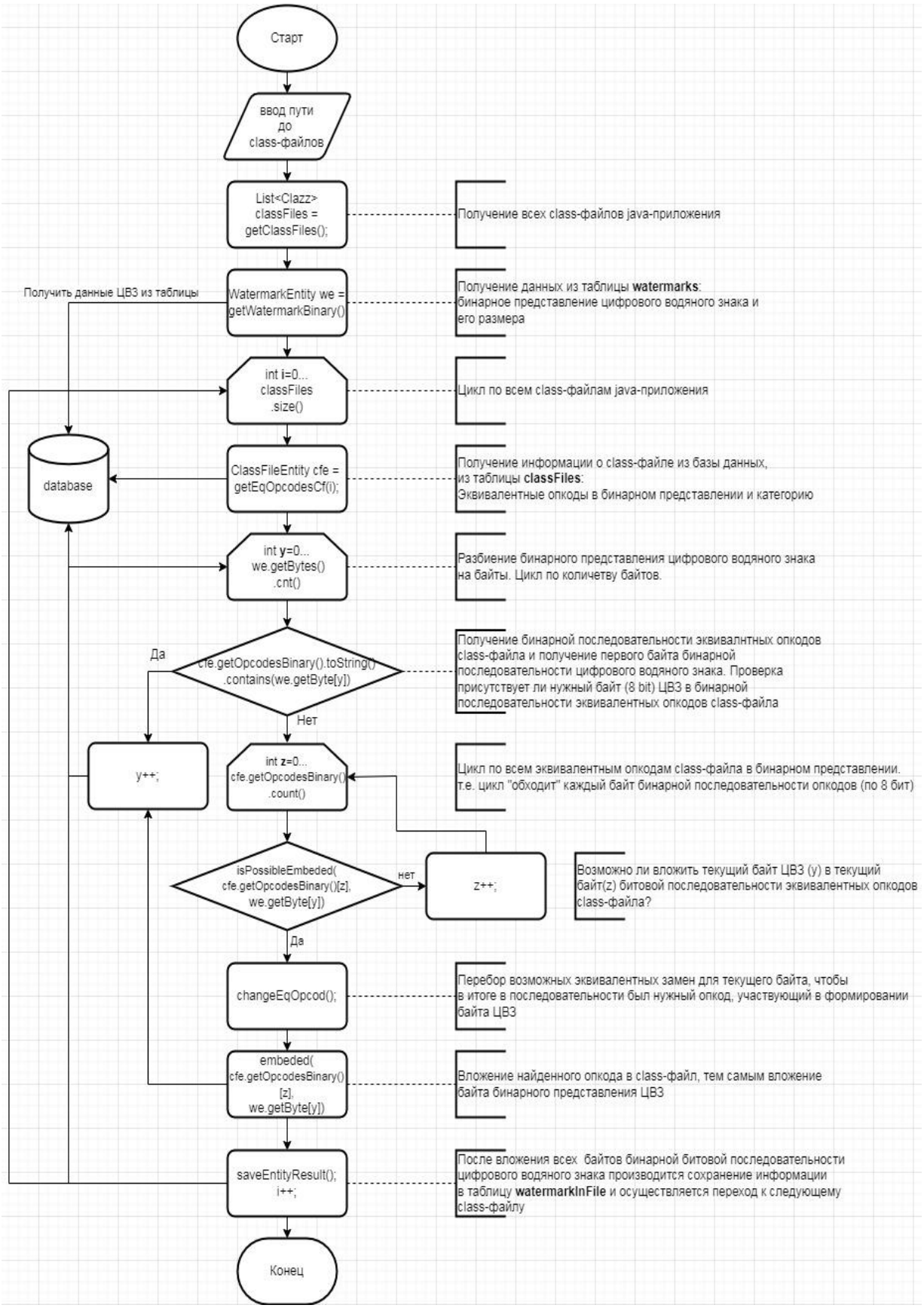


Рисунок 3.9 – Пример блок-схемы части алгоритма реализации вложения цифрового водяного знака в class-файлы java-приложения

Блок-схема, представленная на рисунке 3.9 является цикличной. В связи с тем, что при вложении цифрового водяного знака, каждое действие происходит частями, то необходим цикл по каждому набору частей. Таким образом, главный цикл, который обрабатывает каждый class-файл java-приложения, содержит в себе побочный внутренний цикл. Второй цикл, вложенный в цикл, обрабатывающий class-файлы, является обработчиком цифрового водяного знака, приведенного в бинарное представление и разделенного на байтовые блоки, которыми оперирует цикл. Третий цикл, являющимся вложенным в цикл, обрабатывающий ЦВЗ, обрабатывает каждый байт битовой последовательности опкодов, доступных к эквивалентным заменам, представленных в бинарном формате и производит вложение байта цифрового водяного знака в байт битовой последовательности опкодов class-файла, производя эквивалентные замены опкодов.

Таким образом, сложносоставная блок-схема демонстрирует в трех частях реализацию алгоритма на основе разработанной методики создания и вложения цифрового водяного знака в class-файлы java-приложения устойчивого к атакам декомпиляцией [45].

3.2 Обзор существующих декомпиляторов class-файлов Java

Перед тем, как переходить к анализу эффективности атак с помощью декомпиляции на class-файлы с внедренным в него цифровым водяным знаком, необходимо произвести обзор имеющихся в открытом доступе декомпиляторов, разобрать их основные свойства и возможности [23].

Декомпилятор Java преобразует class-файл в исходный код языка Java. Другими словами, декомпилятор преобразует байт-код в читаемый формат *.java. В силу различия алгоритмов работы внутреннего java-компилятора и алгоритмами декомпиляторов, которые де-факто производят попытки восстановить оригинальный синтаксис языка Java исходя из его мнемонического представления в виде операционных команд байт-кода, находящегося в class-файле, исходный код

Java, из которого был сгенерирован class-файл, не будет точно соответствовать исходному коду Java, сгенерированному декомпилятором Java [90]. Но большая часть кода останется прежней.

Также, необходимо уточнить, что в последнее время, кампания Oracle (разрабатывает Java и соответственно JVM) участила количество релизов новых версий, как языка программирования, так и виртуальной машины Java. Так как в основном, все декомпиляторы Java, которые находятся в открытом доступе, разрабатываются независимыми некоммерческими командами, зачастую декомпилятор не поддерживает последние версии языка программирования или виртуальной машины Java, что создает артефакты, дефекты и риски получить некачественный или нерабочий код, при попытках восстановить исходный код class-файла, который был скомпилирован с помощью более новых алгоритмов, нежели те, которые используются в декомпияторе.

Также, необходимо обратить внимание на тот факт, что большинство доступных в открытом доступе декомпиляторов Java, представляют собой графический интерфейс для библиотеки, которая восстанавливает исходный код Java из одного или нескольких class-файлов. Количество графических интерфейсов гораздо больше, чем количество библиотек, которые производят декомпиляцию class-файлов. Графические интерфейсы в основном различаются пользовательскими свойствами, подсветкой определенных команд и других не влияющих на качество декомпиляции class-файла функциях.

JDProject [101]. Другое название – Java Decompiler. Один из наиболее часто используемых декомпиляторов Java в автономном режиме. Разработан для декомпиляции Java 5 или более поздних версий (на данный момент, версии Java 10). Декомпилятор доступен для Windows, Mac OS и Linux. Также, данный декомпилятор имеет автономную графическую утилиту – JD-GUI, которая отображает исходные коды Java class-файлов. Внутри проекта Java Decompiler и соответственно, графической утилиты JD-GUI используется библиотека **JD-Core** в качестве ядра программы ответственного за декомпиляцию class-файлов.

Procyon [119, 145]. Это набор инструментов мета программирования Java, ориентированный на генерацию и анализ кода. В данный проект входит несколько инструментов, среди них присутствует декомпилятор Java. Данный декомпилятор разрабатывается с октября 2019 года и не всегда работает стабильно. Классы, скомпилированные с помощью внутреннего компилятора Java javac, могут давать неоптимальные результаты, особенно для таких конструкций, как switch-операторы на основе String. Отличительными особенностями данного декомпилятора от других, являются:

- Стабильная декомпиляция перечислений Java (Enum)
- Стабильная декомпиляция перечислений (Enum) и операторов условного перехода реализованными на основе «переключателя» (Switch) для версии Java 7.
- Декомпиляция аннотаций
- Возможность декомпиляции class-файлов, использующих лямбда-выражения (Java 8 Lambdas)

Данный декомпилятор является библиотекой-ядром, на основе которого созданы несколько графических интерфейсов:

- SecureTeam Java Decompiler [77]. Интерфейс декомпилятора на основе JavaFX.
- Luyten [130]. Интерфейс с открытым исходным кодом.
- Bytecode Viewer [56, 62]. Пакет для декомпиляции, дизассемблирования и отладки Java с открытым исходным кодом. Он может создавать декомпилированные исходные коды из нескольких современных декомпиляторов Java, включая **Procyon, CFR и FernFlower**.
- Helios [57, 95]. Независимый проект, в котором вместо Swing используется SWT. Графический интерфейс. Работает только с версией Java 8. В данный момент более не поддерживается, был объединен с проектом **FernFlower** для среды разработки IntelliJ IDEA.

Cavaj Java Decompiler [63]. Это графический интерфейс. Бесплатное автономное приложение только для операционной системы Windows, которое преобразует файлы байт-кода (class-файлы) в исходный код Java. Не требует загрузки и установки Java, это отдельная программа. Cavaj Java Decompiler использует устаревший и более не поддерживаемый (по состоянию на август 2011 года) **JAD** [99] в качестве механизма декомпиляции Java [64].

DJ Java Decompiler [79]. Отдельное приложение, работающее без предустановленной Java Development Kit или Java Runtime Edition. Также, как и Cavaj Java Decompiler, разрабатывался для операционной системы Windows. Является графическим интерфейсом, использующим для декомпиляции библиотеку JAD. Основным преимуществом DJ Java Decompiler является то, что существует возможность производить декомпиляцию более одного class-файла одновременно. DJ Decompiler позволяет сохранять, распечатывать, редактировать и компилировать сгенерированный код Java. Это не только декомпилятор и дизассемблер Java, но и полнофункциональный редактор Java кода, использующий графический пользовательский интерфейс с подсветкой синтаксиса. Позволяет производить низкоуровневое исследование class-файла, так как имеется встроенный в графическую утилиту HEX-редактор [134].

JBVD - Java Bytecode Viewer & Decompiler [107]. Простое графическое приложение для просмотра байт-кода class-файла. Обновления перестали выпускаться после 2017 года. Использует в качестве ядра библиотеку декомпиляции javassist [68, 69, 103], которая поддерживается до сих пор и включена в продукты компании JBoss [106]. Разработка Javassist была частично поддержана программами финансирования PRESTO и CREST Японского агентства науки и технологий.

CFR [65]. Расшифровывается как Class File Reader. Самодостаточная библиотека для декомпиляции class-файлов java-приложения. Полностью написана на версии Java 6, однако, поддерживает весь функционал последних версий Java, включая Java 12 и 14. Есть инструмент командной строки. Закрытый исходный код.

FernFlower [83]. Аналитический декомпилятор для Java. Он используется для декомпиляции расширений файлов class, zip и jar. Находится в стадии разработки. С 2017 года является составной частью среды разработки IntelliJ IDEA [97].

Krakatau [120]. Синтаксис ассемблера Krakatau в основном является надмножеством синтаксиса Jasmin с некоторыми незначительными несовместимостями, но, в отличие от Jasmin, Krakatau имеет полную поддержку всех функций Java 14 и даже поддерживает некоторые недокументированные функции, обнаруженные в старых версиях JVM. В отличие от других декомпиляторов, Krakatau был специально разработан для работы с **обфусцированным** кодом и может легко обрабатывать искажения, которые нарушают работу других декомпиляторов. Однако декомпилятор Krakatau не поддерживает некоторые функции Java 8 и более поздних версий, такие как лямбда-выражения, поэтому он лучше всего работает со старым кодом. Написан и работает полностью на Python.

Recaf [67, 148]. Один из новых и крайне быстро развивающихся проектов по декомпиляции class-файлов. Является самодостаточным ядром декомпиляции с GUI-интерфейсом на JavaFx. Включает в себя подсветку синтаксиса, редактор байт-кода, создание class-файлов на основе байт-кода. Работает с самыми свежими версиями Java. Recaf будет работать с той версией Java, которая установлена в системе по умолчанию. Осуществляется проверка байт-кода в реальном времени при редактировании class-файла [72].

Для языка программирования Java создано множество декомпиляторов под разные платформы [94, 139]. Многие разработки являются устаревшими, не поддерживаются или выкуплены крупными IT-компаниями. После обзора существующих декомпиляторов для Java был произведен их анализ, результат которого представлен ниже в таблице 3.1.

Таблица 3.1 – Сравнение существующих Java декомпиляторов

Декомпилятор	**Ядро для декомпиляции**	**Является графическим интерфейсом?**
JDProject	JD-Core	-
Procyon	Procyon	- не стабилен
SecureTeam Java Decompiler	Procyon	+
Luyten	Procyon	+
Bytecode Viewer	Procyon	+
Helios	Helios	- не поддерживается объединен с FernFlower
Cavaj Java Decompiler	JAD	+
JAD	JAD	-
DJ Java Decompiler	JAD	+
Java Bytecode Viewer & Decompiler	javassist	+
javassist	javassist	- включен в JBoss
JBoss	javassist	+
CFR	CFR	-
FernFlower	FernFlower	- включен в IntelliJ IDEA
IntelliJ IDEA	FernFlower	+
Krakatau	Krakatau , Jasmin	- устарел
Recaf	Recaf	- присутствует графический интерфейс

Таким образом, анализируя полученные данные, которые агрегированы в таблице 3.1 можно сделать вывод о том, что при большом выборе декомпиляторов, в реальных задачах можно ограничиться небольшим набором или одним. Исходя

из данных в таблице выше, производится выборка доступных к использованию и удовлетворяющих условиям исследования декомпиляторов.

При анализе таблицы и поиска, подходящего под текущее исследование декомпилятора, из таблицы исключаются все декомпиляторы, которые являются графическим интерфейсом для другой библиотеки декомпиляции. Остается 8 наименований. Из этих 8 наименований, которые являются самодостаточными библиотеками декомпиляции, исключаются на текущий момент не поддерживаемые, не стабильные или устаревшие проекты. Остается 6 наименований:

- JD-Core
- JAD
- javassist
- CFR
- FernFlower
- Recaf

Из данного списка возможно исключить javassist и FernFlower, в связи с тем, что данные проекты прекратили быть некоммерческими или их некоммерческие версии устарели, имеют ограниченный функционал по сравнению с коммерческими версиями, имеющими закрытый исходный код. Оставшиеся наименования в выборке, при детальном анализе, демонстрируют примерно одинаковый результат работы и скорость выполнения декомпиляции.

3.3 Категоризация цифровых водяных знаков в class-файлах при применении разработанной методики

Прежде, чем приступать к экспериментальной проверке предлагаемой методики создания и вложения цифрового водяного знака в class-файл java-приложения, необходимо определить возможные виды цифровых водяных знаков, которые могут обсуждаться в дальнейшем.

Как обсуждалось в главе 2 текущей работы, методика создания и вложения цифрового водяного знака основана на различных действиях с опкодами байт-кода в class-файле. В основном это действия эквивалентной замены опкодов различных функций ядра языка программирования Java. Например, существуют эквивалентные опкоды отвечающие за ветвление кода, переход по условиям, циклы, инициализацию констант и другие функции, которые используются компилятором Java в зависимости от его версии, операционной системы, конфигураций, сложности и нелинейности исходного кода. Таким образом, по определенной внутренней логике, компилятор при различных условиях, использует различные опкоды для условно одинаковых действий в исходном Java-коде.

После того, как компилятор Java, произвел компиляцию исходного кода и интерпретировал его в операционные команды, которые проанализирует виртуальная машина Java, class-файл можно запускать на выполнение. Иными словами, виртуальная машина Java, при проверке байт-кода встроенным верификатором ничего не знает о логике компилятора и проверяет лишь правильность байт-кода. Виртуальная машина Java не может проверить соответствие исходного кода и операционной команды, которую для него выбрал компилятор Java.

В связи с этим осуществляется одна из ключевых частей работы методики, так как в спецификации виртуальной машины Java содержится большое количество эквивалентных опкодов. Также, в совокупности со знанием исходного кода, можно добиться эффективного вложения цифрового водяного знака практически в любые class-файлы. Например, при знании исходного кода, точнее его логики работы, которую можно получить, воспользовавшись любым декомпилятором, существует возможность проследить использование каждой переменной в программе и определить действительно ли изначально заданный ей тип при инициализации и объявлении переменной, необходим [137]. Если в приложении некоторая переменная при всех возможных исходах использования не принимает значение

больше 50, то в class-файле операционную команду, ответственную за объявление переменной и ее типа, можно заменить с int на short.

В связи с этим, появляются различные возможные пути использования данной методики, что порождает классификацию видов цифрового водяного знака, вложение которого возможно в байт-код class-файла посредством использования предлагаемой методики. Ниже рассматривается классификация возможных видов цифрового водяного знака.

Цифровые водяные знаки в байт-коде class-файла java-приложения изначально возможно разделить на две крупные категории по типам редактирования байт-кода для вложения: структурированные и не структурированные.

Структурированные цифровые водяные знаки.

Структурированным цифровым водяным знаком следует считать цифровой водяной знак, который был подготовлен и создан исходя из структуры java-приложения или class-файла. Предварительно производится анализ java-приложения и составляющих его class-файлов на предмет наличия в class-файлах необходимых переменных, устойчивых конструкций, паттернов, позволяющих произвести вложение цифрового водяного знака. На основе полученной информации производится выборка class-файлов пригодных для вложения цифрового водяного знака выбранной методикой. Среди полученной выборки определяется class-файл с самым малым конфидентом эффективности вложения, данный class-файл берется за основу. Производится допущение, что объем, который возможно вложить в данный class-файл, будет возможно вложить в остальные class-файлы выборки с более широкой ветвистостью кода, наличием условий, циклов, переменных.

Таким образом, основа цифрового водяного знака заключается в минимально допустимом объеме вложения выборки class-файлов в единичный элемент данной выборки. При использовании качестве контейнера для вложения цифрового водяного знака class-файла с большим коэффициентом эффективности вложения,

цифровой водяной знак до обогащается информацией, однако, все также основываясь на конкретном class-файле.

Иными словами, структурированные цифровые водяные знаки – это вложение информации в class-файл на основе знания о структуре контейнера, полученной заранее в ходе анализа доступных к вложению контейнеров (class-файлов). Цифровой водяной знак создается под определенную структуру и под нее же редактируется.

Не структурированные цифровые водяные знаки

Данные цифровые водяные знаки являются противоположностью структурированных. Данная категория цифровых водяных знаков используется, когда необходимо ускорить вложение информации в class-файлы java-приложения. Сокращение времени и экономия ресурсов происходит за счет того, что весь проект java-приложения не подвергается анализу. Заранее подготовленный цифровой водяной знак, который является самодостаточным и не основан на структуре контейнера покрывающего сообщения.

Таким образом цифровые водяные знака данного типа основаны не на структуре, они являются готовыми словосочетаниями, фразами, годами, цифрами и другой информацией, которая создается на основе других факторов отличных от структуры кода. Например, торговое наименование, усложненные фразы покрывающие определенные риски и прочее. Вложение производится без анализа структуры java-приложения и составляющих его class-файлов. При невозможности вложения всего цифрового водяного знака в конкретный class-файл, производится вложение части или не производится совсем. В некоторых случаях, возможна подготовка нескольких цифровых водяных знаков разного объема, на случай возникновения ситуации, когда class-файл обладает более сложной структурой исходного кода и благодаря этому существует возможность произвести вложение цифровой водяной знак полностью или большего объема.

Иными словами, не структурированные цифровые водяные знаки – это вложение заранее подготовленной информации, в отрыве от структуры контейнера,

в который предполагается вложение. Благодаря этому экономится время на процедуре анализа java-приложения, но появляются риски наличия некоторых class-файлов java-приложения, которые в силу своей отличающейся структуры кода, не смогли вместить в себя цифровой водяной знак частично или полностью, что в свою очередь может снизить эффективность цифрового водяного знака и его устойчивость к атакам.

Представленное выше разделение типов цифровых водяных знаков для class-файлов java-приложения на категории, является верхнеуровневым и не исчерпывающим, так как каждую из представленных выше категорий, возможно разделить на дополнительные подкатегории, позволяющие произвести более углубленную и разветвленную классификацию видов цифровых водяных знаков, используемых в методике, предлагаемой в данной работе. Ниже приведена дополнительная классификация цифровых водяных знаков, демонстрирующая разделение каждой из вышеописанных категорий на подкатегории, но также не являющейся исчерпывающей и имеющей возможности к еще более глубокой категоризации. Однако в данной работе в такой углубленной категоризации цифровых водяных знаков нет необходимости.

Контекстно-информационные цифровые водяные знаки

Данная подкатегория цифровых водяных знаков может относится, как к основанным на структуре, так и без нее. Контекстно-информационные цифровые знаки содержат в себе некоторый информационный набор в контексте текущего java-приложения и всего к нему относящегося. Информацией может быть любое слово, фраза, словосочетание, номер телефона, номер заявки на патент, наименование бренда, компании и другие данные однозначно указывающие на правообладателя. Таким образом владелец программного обеспечения стремится оставить свой собственный цифровой маркер в его программном продукте в виде цифрового водяного знака.

В связи с тем, что цифровые водяные знаки данной категории – это конечный набор опознавательных фраз или одна единственная, то их возможно отнести к разным категориям первого уровня в зависимости от способа их вложения в class-файлы.

Например, если рассматривать случай, когда компания, в лице ее руководства, хочет в кратчайшие сроки использовать свое собственное наименование в качестве цифрового водяного знака и ничто другое, то данный случай явный пример контекстно-информационного цифрового знака, основанного не на структуре кода. Или иными словами *«не структурированный контекстно-информационный цифровой водяной знак»*. Рассмотрим подробнее определение классификации цифрового водяного знака для данного случая:

- руководство компании заранее подготовило фразу (зарегистрированное наименование компании) для создания на ее основе цифрового водяного знака
- руководство компании готово рассматривать в качестве базиса для создания цифрового водяного знака только эту фразу
- руководство компании сжато в сроках по каким-то причинам

Перечисленные выше пункты позволяют сделать вывод, что специалистам, которые будут производить вложение цифрового водяного знака на основе методики предлагаемой в данной работе, придется ее упрощать, а, следовательно, не выполнять часть действий в каждой из фаз. Таким образом, специалисты будут создавать цифровой водяной знак на основе заранее заготовленной фразы и производить попытку вложения в class-файл java-приложения, а при неудачной попытке переходить к следующему class-файлу. Как обсуждалось ранее, данный подход менее ресурсоемкий и более быстрый, но менее эффективный, так как class-файлами, в которые не удастся вложить даже частично такой цифровой водяной знак, могут быть class-файлы, отвечающие за ключевую логику работы приложения или за ее часть.

Рассматривая данный пример с другой стороны и с другими вводными данными от руководства компании, цифровой водяной знак можно классифицировать как контекстно-информационный цифровой водяной знак, основанный на структуре. Или *«структурированный контекстно-информационный цифровой водяной знак»*. Это будет возможным при:

- отсутствии ограничения по времени
- предварительном анализе всех class-файлов java-приложения
- на основе полученной, в ходе анализа, информации создать набор основных фраз для использования их в качестве основы для цифрового водяного знака, в зависимости от сложности и ветвистости логики class-файла и его объема.

Адаптационно-случайные цифровые водяные знаки

Данная подкатегория цифровых водяных знаков относится только к категории не структурированных. Цифровые водяные знаки данной категории не создаются заранее, так как для их создания и вложения используется информация, которую возможно получить из уже рабочего и скомпилированного class-файла java-приложения или его байт-кода, после чего производятся замены опкодов случайным образом с составлением карты вложений.

Таким образом нельзя заранее предугадать, каким будет цифровой водяной знак, в связи с тем, что алгоритмы компилятора по выбору опкодов работают по-разному в различных ситуациях, что говорит о некоторой псевдо-случайности. А так как данная подкатегория цифровых водяных знаков основывается на уже готовом наборе опкодов class-файла и не может выйти за пределы допустимого количества доступных к эквивалентной замене опкодов, то это может говорить о некоторой адаптивности.

Цифровые водяные знаки данной подкатегории вариативны в своем объеме в зависимости от конкретного class-файла работа с которым осуществляется в данный момент времени. Обычно для вложения цифровых водяных знаков данной подкатегории не требуется квалификации и дополнительных умений, так как используются уже готовое программное обеспечение, которое автоматически

сканирует каждый class-файл и производит замены опкодов внутри него на основе имеющейся, у данного модификатора class-файлов, таблицы эквивалентности опкодов и введенных данных пользователя. Чаще всего от пользователя требуется только выбрать способ эквивалентной замены опкодов – абсолютно все в каждом class-файле или каждый третий, например.

Использование данной подкатегории цифровых водяных знаков является самым простым и наименее ресурсоемким, однако, также является самым малоэффективным и критически не устойчивым к атакам по причинам:

- практически ничего не мешает использовать злоумышленнику такое же программное обеспечение для того, чтобы еще раз поменять все опкоды в class-файлах на эквивалентные. Данное действие гарантировано повредит и с большой вероятностью уничтожит цифровой водяной знак.
- после окончания работы программное обеспечение для модификации составляет карту покрытия цифровым водяным знаком class-файлов java-приложения, где указано в каком class-файле какие замены произведены и каким должен быть порядок основных (критически важных для проверки) опкодов в каждом из class-файлов. Если злоумышленникам удастся выкрасть class-файлы, то риск того, что данная карта будет найдена и также украдена велик.

Данные цифровые водяные знаки являются адаптационно-случайными и не основываются на структуре class-файла, так как при их создании и вложении не производится анализ class-файла в понимании категории структурированных цифровых водяных знаков. Или иначе – «*не структурированный адаптационно-случайный цифровой водяной знак*».

Иерархия категорий цифровых водяных знаков представлена на рисунке 3.10, а карта зависимости различных категорий цифровых водяных знаков от способности устойчивости к атакам и скорости реализации, представлена на рисунке 3.11.

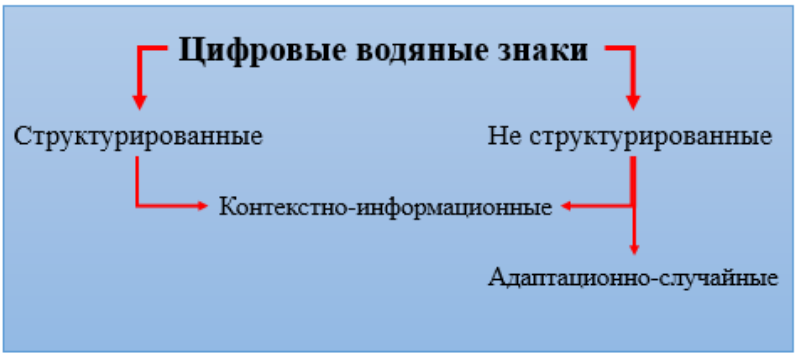


Рисунок 3.10 – Иерархия категорий цифровых водяных знаков

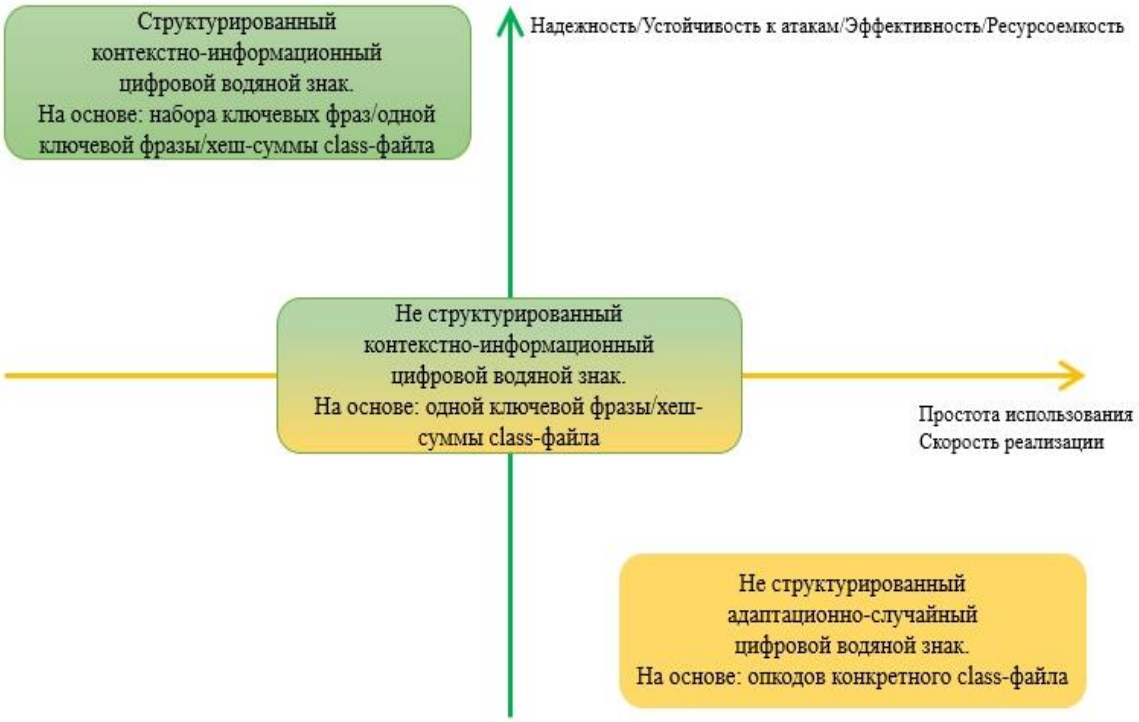


Рисунок 3.11 – Карта зависимости категорий цифровых водяных знаков

Необходимо отметить, что представленная в данном блоке классификации цифровых водяных знаков, разделенных на категории по типам редактирования байт-кода class-файлов и типам подготовки информационных сообщений для последующего вложения, не является сугубо локальной и применимой непосредственно только к предлагаемой методике скрытого вложения цифровых водяных знаков в байт-код java-приложения. При необходимости, данную классификацию возможно использовать при обсуждении вложения цифровых

водяных знаков с использованием других методик и на основе выходных файлов, и особенностей работы других языков программирования.

3.4. Исследование устойчивости цифрового водяного знака в class-файлах java-приложения к атакам декомпиляцией

В данном блоке будет осуществлена проверка устойчивости цифрового водяного знака, вложенного в class-файл java-приложения по описанной в главе 2 методике, к атакам, основанным на декомпиляции class-файла сторонним программным обеспечением, приводящим к разрушению или уничтожению цифрового водяного знака.

Необходимо дать пояснение принципам атаки декомпиляцией. Данная атака заключается в том, что злоумышленник нелегально получает доступ к class-файлу, чаще всего компрометируя всю информационную систему java-приложения [37]. После чего, злоумышленнику необходимо изменить исходный код java-приложения или информационной системы таким образом, чтобы, например, работа модуля оплаты была основана на подставных реквизитах злоумышленника. Соответственно, злоумышленник декомпилирует class-файлы, реализующие логику работы модуля оплаты, вносит изменения в конфигурации и указывает их в исходном коде, полученным после декомпиляции. После чего производит повторную компиляцию исходного кода, получая в результате совершенно другой class-файл, который:

- содержит в себе перенаправления способов оплаты на реквизиты злоумышленника
- имеет отличную от первоначальной хеш-сумму class-файла
- продолжает работать согласно заявленной спецификации для модуля оплаты
- предположительно, очищен от возможных меток, маркеров, цифровых водяных знаков, в силу того, что перекомпиляция происходила на другом оборудовании, другой версии виртуальной машины Java и других условиях, что

приводит к разрушению большинства цифровых водяных знаков, вложенных в class-файл посредством существующих методик [55, 92, 93, 124].

Заключительным шагом является замена class-файлов модуля оплаты java-приложения на class-файлы злоумышленника. Данный вид атаки на class-файлы java-приложения может быть использован, как один из пунктов, составляющих процедуру фарминга [61, 113]. Более подробная модель нарушителя будет рассмотрена в последующих блоках данной работы.

3.4.1. Компиляция class-файла

Для того, чтобы исследование было максимально приближено к реальной логике возможного class-файла модуля оплаты java-приложения, в данном блоке вся работа будет производиться с одним class-файлом, имеющим распределенную логику исходного кода, основанную на ветвлении кода, что часто встречается в реальных банковских системах. Исходный код будущего class-файла с упрощенным каркасом логики работы java-приложения, основанным на реальном банковском приложении представлен в листинге 3.1.

Листинг 3.1 – Исходный код class-файла из модуля оплаты

```
public class TestWatermark {
    public static void main(String[] args) {
        int a = 1;
        int b = 2;
        int c = 3;
        int sum = 0;
        System.out.println("Start application");
        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
        System.out.println(sum);
```

```
        if (a < b) {
            System.out.println("a < b");
        } else {
            System.out.println("b < a");
        }
        if (c != a) {
            System.out.println("c != a");
        } else {
            System.out.println("c = a");
        }
        if (a != b) {
            sum = a + b;
        } else {
            sum = c;
        }
        if (sum > c) {
            System.out.println("sum > c");
        } else {
            System.out.println("sum a + b = c");
        }
        System.out.println("End application");
    }
}
```

Так как в дальнейшем, исследование будет происходить на основе программного кода, представленного в листинге 3.1, необходимо дать пояснения по ключевым аспектам. В силу того, что данный код имеет упрощенный характер, то основные переменные участвующие в логике работы не передаются на вход данному коду, а уже находятся внутри него в виде переменных [a, b, c, sum]. Далее логика представленного кода представлена «ветвлениями», что позволяет принять

решение по поводу плательщика на основе входных параметров, после чего вывести результат принятия решения.

С точки зрения виртуальной машины Java и операционных команд байт-кода, которыми она оперирует, в данном фрагменте кода представлены четыре оператора условных переходов, как минимум трех категорий, согласно спецификации виртуальной машины Java. Таким образом, возможно осуществить допущение, что в байт-коде скомпилированного class-файла будет иметься, как минимум четыре операционных команды, отвечающих за операции условных переходов, что позволит произвести замены на эквивалентные операторы, с соблюдением процедуры предлагаемой методики и предъявляемых критериев к цифровым водяным знакам в исполняемых файлах java.

Для реализации действия компиляции приведенного выше кода, будет использоваться стандартный компилятор java, входящий в любую поставку Java Development Kit. Таким образом, необходимо текстовый файл, содержащий исходный код представить в расширении *.java, что позволит компилятору java начать работу с данным файлом. Операция компиляции текстового файла с расширением *.java в исполняемый class-файл виртуальной машины Java, представлена в листинге 3.2.

Листинг 3.2 – Команда для запуска компиляции исходного кода java

```
javac TestWatermark.java
```

Данная команда вызывает компилятор java для компиляции переданного исходного кода в class-файл, с которым в дальнейшем уже возможна работа в виртуальной машине Java, при отсутствии ошибок компиляции. Полученный class-файл возможно запустить на выполнение, чтобы убедиться в правильности работы логики предложенного кода. Команда для запуска class-файла TestWatermark.class представлена в листинге 3.3.

Листинг 3.3 – Команда для запуска скомпилированного class-файла

```
java TestWatermark
```

Таким образом, за два действия был скомпилирован class-файл, в который в последствии будет произведено вложение цифрового водяного знака, после чего применена атака декомпиляцией. Результат выполнения class-файла продемонстрирован на рисунке 3.12.

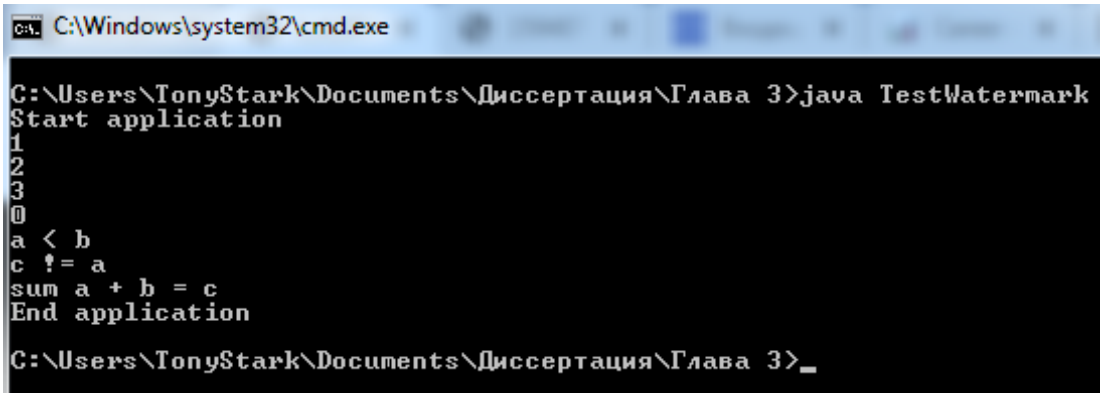


Рисунок 3.12 – Результат выполнения скомпилированного class-файла

3.4.2 Анализ структуры байт-кода полученного class-файла

Следующим шагом необходимо произвести анализ полученной структуры байт-кода class-файла, чтобы убедиться в ранее озвученных предположениях.

Также, необходимо определить соглашение о наименованиях java-файлов и class-файлов, в рамках текущего исследования, во избежание ошибок и повышения отличительных характеристик. Таким образом, исходный код на языке программирования java из блока 3.4.1 далее будет именоваться «базовый java-файл». Полученный class-файл после компиляции базового java-файла, далее именуется, как «чистый class-файл». Class-файл с ЦВЗ, будем называть, как «грязный class-файл». Исходный Java-код, полученный в результате декомпиляции «грязного class-файла» с вложенным ЦВЗ, будем называть, как «вторичный java-файл». Class-файл, полученный в результате компиляции java-кода из вторичного java-файла, именуется далее, как «перекомпилированный грязный class-файл».

Для анализа, полученного в блоке 3.4.1 базового class-файла, необходимо воспользоваться специализированным инструментарием. Таким инструментарием может служить, как стандартная утилита из Java Development Kit «javap –c», так и более сложный инструмент в виде отдельного программного обеспечения, позволяющего производить модификации байт-кода class-файла, помимо его отображения. Так как результат выполнения идентичен и в данном исследовании необходимо производить редактирование байт-кода class-файла, будет использоваться специализированное программное обеспечение с графическим интерфейсом, для наглядной демонстрации. Получение байт-кода class-файла будет производиться с помощью программного обеспечения Java Bytecode Editor [105].

После загрузки чистого class-файла в Java Bytecode Editor, появляется возможность просмотра байт-кода данного class-файла и его редактирования. Подтверждаются предположения, описанные в блоке 3.4.1. Байт-код исполняемого class-файла содержит 4 опкода условных операторов внутри логики базового java-файла. Полный листинг байт-кода чистого class-файла приведен в листинге 3.4.

Листинг 3.4 – Байт-код оригинального class-файла без цифрового водяного знака

```
iconst_1
istore_1
iconst_2
istore_2
iconst_3
istore_3
iconst_0
istore 4
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "Start application"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
```

```
getstatic java/lang/System/out Ljava/io/PrintStream;
iload_1
invokevirtual java/io/PrintStream/println(I)V
getstatic java/lang/System/out Ljava/io/PrintStream;
iload_2
invokevirtual java/io/PrintStream/println(I)V
getstatic java/lang/System/out Ljava/io/PrintStream;
iload_3
invokevirtual java/io/PrintStream/println(I)V
getstatic java/lang/System/out Ljava/io/PrintStream;
iload 4
invokevirtual java/io/PrintStream/println(I)V
iload_1
iload_2
if_icmpge 31
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "a < b"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
goto 34
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "b < a"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
iload_3
iload_1
if_icmpeq 41
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "c != a"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
goto 44
getstatic java/lang/System/out Ljava/io/PrintStream;
```

```
ldc "c = a"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
iload_1
iload_2
if_icmpeq 52
iload_1
iload_2
iadd
istore 4
goto 54
iload_3
istore 4
iload 4
iload_3
if_icmple 61
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "sum > c"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
goto 64
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "sum a + b = c"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "End application"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
return
```

В байт-коде чистого class-файла присутствуют операционные команды, отвечающие за операторы условных переходов в исходном java-коде. Ниже, в

таблице 3.2, приведены данные опкоды и операторы условных переходов исходного кода, за которые они ответственны.

Таблица 3.2 – Соответствие команд байт-кода операторам программного кода

Наименование опкода в байт-коде	**Соответствующее действие в программном коде**
if_icmpge	a < b
if_icmpeq	c != a
if_icmpeq	a != b
if_icmple	sum > c

Таким образом, в данном class-файле имеется четыре явных точки для вложения цифрового водяного знака. Однако с учетом того, что в логике используются переменные, то можно предположить, что помимо явных, для предлагаемой методики, точек вложения цифрового водяного знака, присутствуют также и вероятные точки вложения. Такими вероятными точками для вложения цифрового водяного знака могут быть: типы переменных, выводимая текстовая информация.

3.4.3 Изменения размера class-файла при редактировании байт-кода

С целью определения максимальной возможности скрытого редактирования байт-кода class-файла, необходимо произвести поэтапные изменения байт-кода с возрастающей сложностью. Таким образом, первым этапом будет осуществлено грубое редактирование всех опкодов ответственных за операторы условных переходов. После этого, дополнительно, будет произведена замена типов переменных и способов их записи и вызова из пула констант.

При увеличении объемов выводимой текстовой информации увеличивается размер файла, а при уменьшении уменьшается. Данные изменения заметны, как при проверке работоспособности программного обеспечения, так и при анализе

байт-кода. В связи с этим, изменения байт-кода затрагивающие отображаемую и выводимую текстовую информацию, не будут производиться.

Таким образом, будет производиться сравнение трех class-файлов. Чистого class-файла, с измененными операторами условного перехода и с измененными типами переменных совместно с операторами условных переходов. Сравнение основано на двух параметрах: результат выполнения class-файла и итоговый размер class-файла, после произведенных изменений.

При первом редактировании, все операционные коды, отвечающие за условные переходы в class-файле, будут грубо заменены на операционный код одного типа. Иными словами, все операционные коды чистого class-файла, представленные в таблице 3.2, будут заменены опкодом if_icmpgt, означающим условный переход вида value1 <= value2. При втором редактировании, будет производится редактирование class-файла из первого случая. Таким действием реализуется возрастающая сложность изменений чистого class-файла. Во время второго редактирования, помимо ранее измененных опкодов условных переходов, дополнительно изменены типы переменных и способы их сохранения на стек виртуальной машины Java. Детальные изменения байт-кода чистого class-файла продемонстрированы в таблице 3.3.

Таблица 3.3 – Детальное описание изменений опкодов чистого class-файла

Исходные опкоды	**Редактирование 1. Измененные опкоды.**	**Редактирование 2. Измененные опкоды.**
if_icmpge	if_icmpgt	-
if_icmpeq	if_icmpgt	-
if_icmple	if_icmpgt	-
iconst_1, istore_1, iload_1	-	lconst_1, lstore_1, lload_1
iconst_2, istore_2, iload_2	-	-
iconst_3, istore_3, iload_3	-	iconst_3, fstore_3, fload_3
iconst_0, istore 4, iload_4	-	dconst_0, dstore 4, dload_4
iadd	-	ladd

invokevirtual java/io/PrintStream/println(I)V	-	Несколько разных по типам: …println(J)V/ println(F)V/ println(D)V

Результат двухэтапного усложняющегося редактирования байт-кода чистого class-файла продемонстрирован на рисунке 3.13.

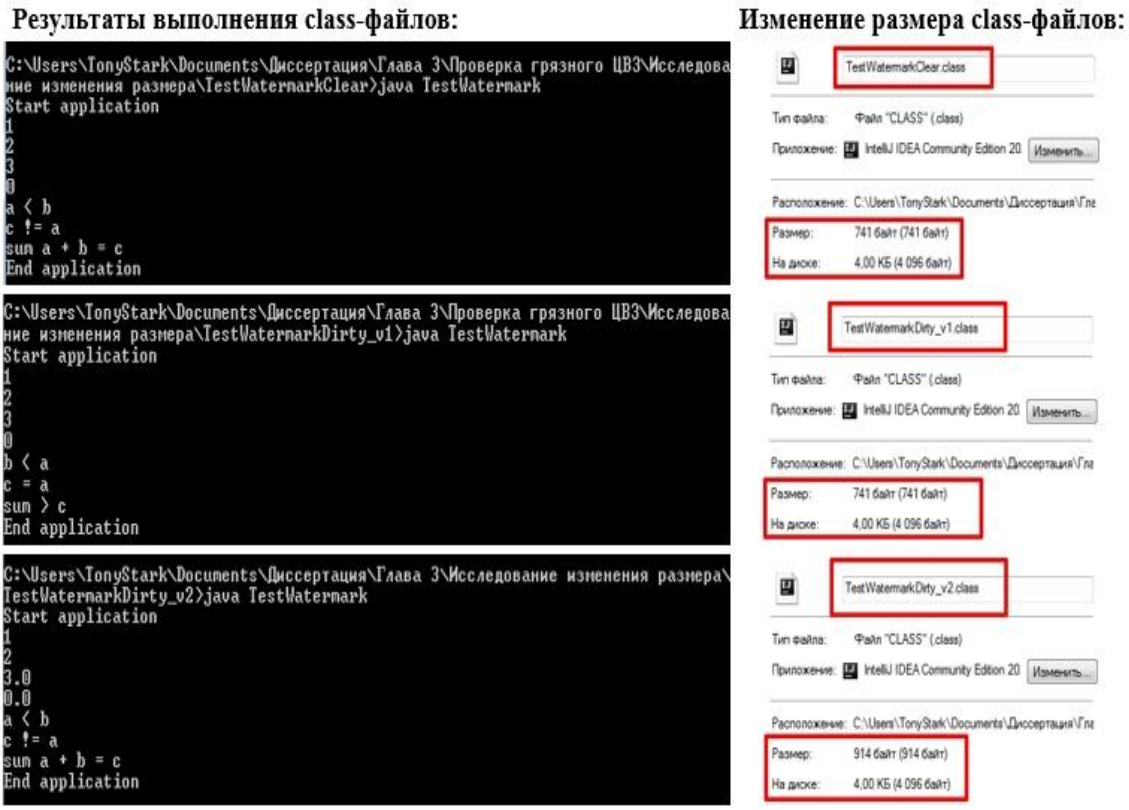


Рисунок 3.13 – Результат редактирования байт-кода class-файла

Из рисунка 3.13 видно, что при грубом изменении типов переменных class-файла его размер увеличивается. Из чего сделан вывод, что для редактирования типов переменных и их сохранения на стек виртуальной машины Java, требуется более сложный и глубокий анализ каждого class-файла по отдельности, что означает экспоненциальный рост сложности скрытого вложения цифрового водяного знака в java-приложении, в зависимости от количества содержащихся в нем class-файлов.

3.4.4 Вложение цифрового водяного знака в чистый class-файл

Перед тем, как производить вложение цифрового водяного знака, необходимо дать пояснение касательно процедуры вложения и используемого типа цифрового водяного знака. В связи с тем, что пример исходного кода основан на реальном банковском java-приложении, но не является им в силу ряда причин, необходимость во всестороннем анализе чистого class-файла не является необходимым действием.

Таким образом, исходя из ранее продемонстрированной карты зависимостей категорий цифровых водяных знаков в блоке 3 данной главы, делается вывод о том, что при вложении в class-файл цифрового водяного знака с низкими характеристиками устойчивости к атакам перекомпиляции, в случае его устойчивости к атаке, следует устойчивость к данному виду атак цифровых водяных знаков с более высокими характеристиками устойчивости к атакам перекомпиляции.

Исходя из терминологии, представленной в блоке 3 данной главы, следует, что для того, чтобы подтвердить устойчивость цифрового водяного знака к атакам перекомпиляцией, созданного и вложенного в class-файл с помощью предлагаемой методики, достаточно подтвердить устойчивость к атаке перекомпиляцией цифрового водяного знака категории «Адаптационно-случайных цифровых водяных знаков», обладающего минимальными характеристиками устойчивости к атакам.

Следовательно, для данной категории цифровых водяных знаков не требуется предварительный анализ байт-кода class-файла и создание цифрового водяного знака исходя из его структуры. Цифровой водяной знак создается во время вложения в class-файл на основе тех операционных кодов, которые заменяются на эквивалентные, с сохранением их номеров, для дальнейшей проверки. Таким образом, в чистый class-файл производится вложение без информационного не структурированного адаптационно-случайного цифрового водяного знака. Данный цифровой знак будет состоять из сохранённого состояния class-файла после вложения, наименования эквивалентных опкодов, на которые

была произведена замена, а также не будет основан на структуре чистого class-файла.

Для упрощения проверки данного выше предположения, будет произведена эквивалентная замена одного опкода, ответственного за операцию условного перехода, с изменением выгрузки со стека виртуальной машины Java переменных, участвующих в логическом выражении для ветвления данного блока кода. Для вложения цифрового водяного знака в байт-код чистого class-файла, который полностью приведен в листинге 3.4, был выбран опкод if_icmpge, отвечающий в исходном коде за проверку логического выражения «a < b». Что является, в свою очередь, сравнением двух переменных, значения которых равны 1 и 2, соответственно. С последующей проверкой, что переменная «a», содержащая в себе значение 1, меньше переменной «b», содержащей в себе значение 2. Таким образом, результат выполнения class-файла должен вывести в консоль строчку текста «a < b».

Операционный код if_icmpge, будет заменен на эквивалентный ему if_icmple, что означает изменение условия, содержащегося в условном переходе исходного кода с «a < b» на «b > a». Также, для сохранения логики работы class-файла была произведена замена порядка добавления на стек переменных JVM, которые используются для проверки логического выражения. На рисунке 3.14, продемонстрировано выделением цветом опкода, посредством которого происходит вложение цифрового водяного знака в байт-код чистого class-файла без перекомпиляции.

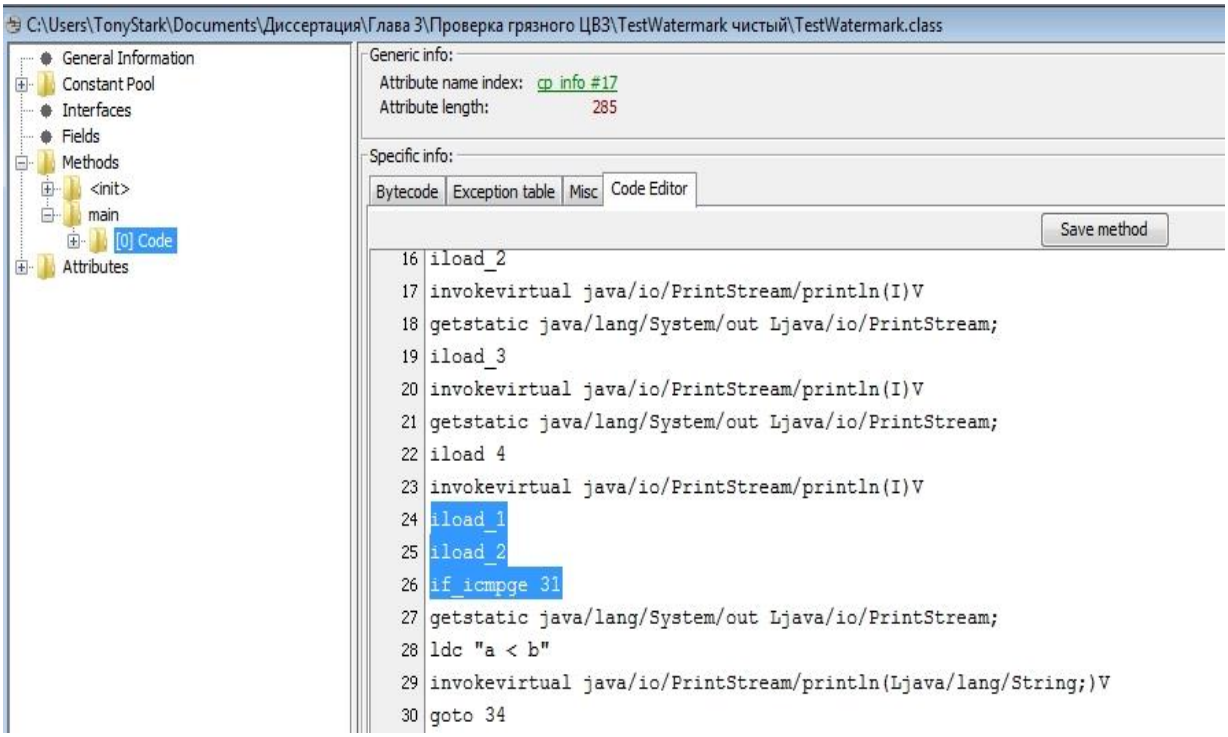


Рисунок 3.14 – Переменные и опкод чистого class-файла для вложения цифрового водяного знака

После того, как все изменения байт-кода чистого class-файла, связанные с вложением цифрового водяного знака завершены, результатом данных действий является грязный class-файл, то есть class-файл, который содержит в себе примитивный цифровой водяной знак. Байт-код грязного class-файла с выделением цветом опкодов, претерпевших изменение, продемонстрирован на рисунке 3.15.

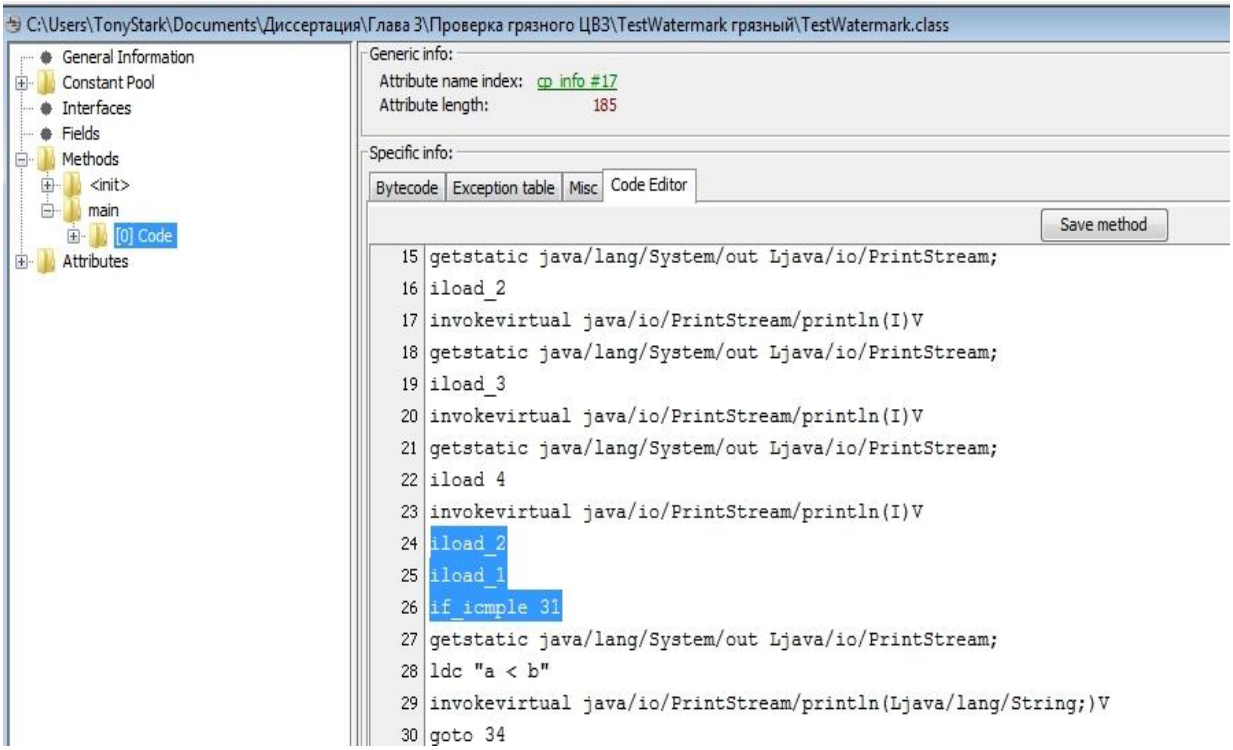


Рисунок 3.15 – Переменные и опкод грязного class-файла после вложения цифрового водяного знака

Данный ЦВЗ содержится в байт-коде в строках 24-26. Для его проверки необходимо проанализировать изначальный class-файл на наличие данных команд в его байт-коде. Данные команды должны быть на тех же строках 24-26 и в той же последовательности. Данная операция подтвердит неизменность байт-кода, а, следовательно, и примитивного цифрового водяного знака. Полный листинг байт-кода грязного class-файла приведен в листинге 3.5.

Листинг 3.5 – Байт-код class-файла с вложенным цифровым водяным знаком

```
iconst_1
istore_1
iconst_2
istore_2
iconst_3
istore_3
```

```
iconst_0
istore 4
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "Start application"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
getstatic java/lang/System/out Ljava/io/PrintStream;
iload_1
invokevirtual java/io/PrintStream/println(I)V
getstatic java/lang/System/out Ljava/io/PrintStream;
iload_2
invokevirtual java/io/PrintStream/println(I)V
getstatic java/lang/System/out Ljava/io/PrintStream;
iload_3
invokevirtual java/io/PrintStream/println(I)V
getstatic java/lang/System/out Ljava/io/PrintStream;
iload 4
invokevirtual java/io/PrintStream/println(I)V
iload_2
iload_1
if_icmple 31
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "a < b"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
goto 34
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "b < a"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
iload_3
iload_1
if_icmpeq 41
```

```
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "c != a"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
goto 44
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "c = a"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
iload_1
iload_2
if_icmpeq 52
iload_1
iload_2
iadd
istore 4
goto 54
iload_3
istore 4
iload 4
iload_3
if_icmple 61
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "sum > c"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
goto 64
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "sum a + b = c"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "End application"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
```

return

Изменение порядка выгрузки переменных со стека переменных виртуальной машины Java не повлияло на результат выполнения class-файла. Если бы был заменен единственный опкод условного перехода на эквивалентный, то логика выполнения class-файла была бы изменена, что привело бы к нарушению требований к цифровым водяным знакам в исполняемых файлах и их критериев.

3.4.5 Атака перекомпиляцией на class-файл с цифровым водяным знаком

Атака перекомпиляцией включает в себя две фазы, исходя из ее наименования. Первая фаза атаки заключается в том, что злоумышленник, используя декомпилятор class-файлов, производит декомпиляцию скомпрометированных class-файлов. Дополнительно, перед фазой декомпиляции class-файлов может быть проведена процедура сбора данных о оборудовании, на котором работает java-приложение, например, с помощью социальной инженерии. Данный шаг позволяет понять на каком оборудовании точно нельзя производить декомпиляцию скомпрометированных class-файлов. Потому что существует вероятность идентичности работы виртуальной машины Java и декомпиляторов с компилятором Java на идентичном оборудовании, что повышает для злоумышленника риск сохранения цифрового водяного знака в class-файле.

Также, перед проведением декомпиляции, злоумышленник, с помощью стандартных утилит из Java Development Kit, может воспользоваться командой «javap -v class-file | findstr major». Принцип работы данной команды заключается в том, что она позволяет получить версию Java Development Kit, с помощью которой был скомпилирован указанный class-файл. Пример выполнения данной команды для class-файла с цифровым водяным знаком, используемым в данном исследовании, и ее результат приведен в листинге 3.6 и на рисунке 3.16, соответственно.

Листинг 3.6 – Получение версии JDK компилировавшей указанный class-файл

```
javap -v TestWatermark | findstr major
```

Рисунок 3.16 – Результат получения версии Java Development Kit, с помощью которой происходила компиляция грязного class-файла

Рисунок 3.16 демонстрирует, что старшая версия сборки Java Development Kit, является 55-й. Данная информация является открытой и находится в спецификации виртуальной машины Java, на официальном сайте Oracle [166]. Таким образом, из спецификации виртуальной машины Java, возможно выяснить, что данный class-файл был скомпилирован Java Development Kit версии 55, которая присутствовала в релизе Java 11, следовательно, чтобы повысить вероятность уничтожения цифрового водяного знака злоумышленнику необходимо произвести декомпиляцию данного class-файла на версии Java Development Kit, которая отличается от той, на которой он был скомпилирован.

Также, проведение описанных выше подготовительных действий, облегчает существование облачных онлайн-сервисов, которые предоставляют в аренду функционал web-сайта содержащий в себе несколько видов декомпиляторов Java с возможностью выбора системы и оборудования и ряда других настроек.

Исходя из данных полученных в блоке 2 текущей главы и описанных выше возможных действий злоумышленника, для проведения локальной декомпиляции грязного class-файла был выбран декомпилятор DJ Java Decompiler [79]. Также дополнительно, для повышения эффективности проводимого эксперимента, будут использованы онлайн версии декомпиляторов на основе других библиотек в качестве ядра декомпилятора, чтобы полноценно имитировать возможные действия злоумышленников.

Таким образом, загружая грязный class-файл с вложенным цифровым водяным знаком в декомпилятор, совершается первая фаза. Результат первой фазы атаки перекомпиляцией продемонстрирован на рисунке 3.17.

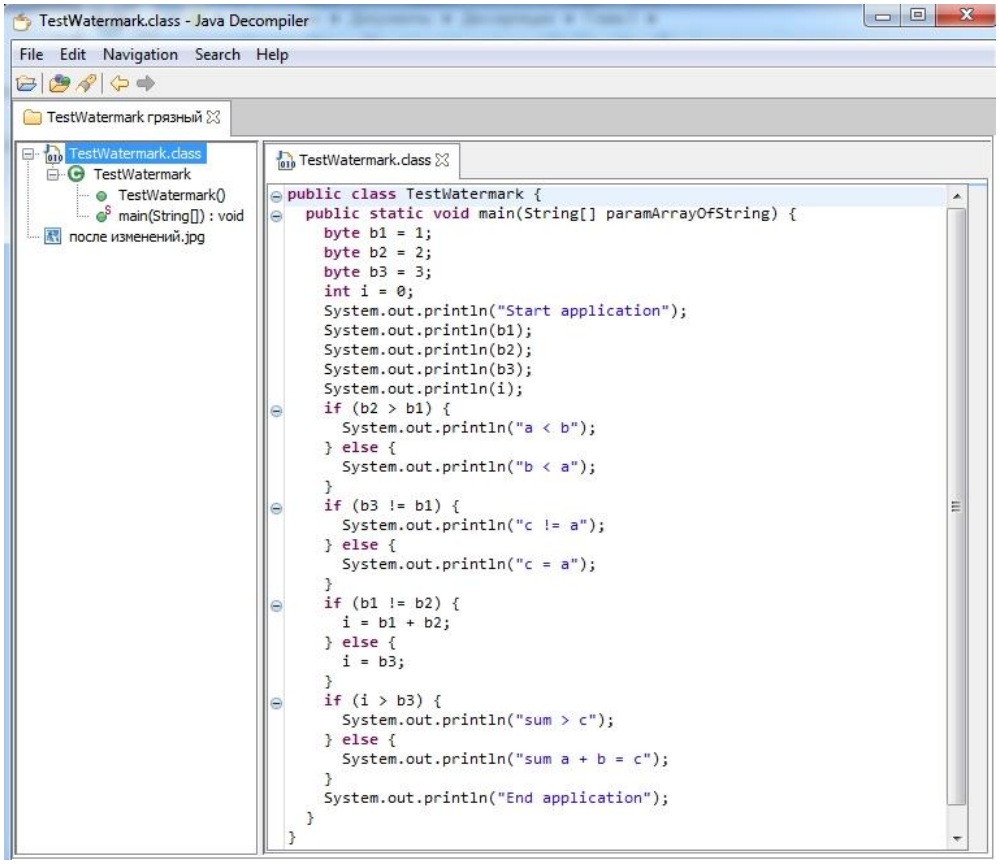


Рисунок 3.17 – Результат декомпиляции грязного class-файла посредством DJ Java Decompiler

Как видно из рисунка 3.17, за исключением изменения наименования переменных, логика class-файла, по сравнению с оригинальным java-файлом сохранена, следовательно, декомпиляция успешна и не нарушила логику работы class-файла. Листинг java-кода полученного после декомпиляции грязного class-файла локальным декомпилятором DJ Java Decompiler представлен в листинге 3.7.

Второй фазой атаки перекомпиляцей будет являться повторная компиляция полученного кода, посредством декомпиляции. Таким образом, код полученный в декомпиляторе DJ Java Decompiler и представленный в листинге 3.7, копируется в новый java-файл, который будет являться вторичным по отношению к базовому

java-файлу. После создания вторичного java-файла, производится процедура его компиляции, согласно шагам, приведенным в блоке 3.4.1 и листинга 3.2. После чего, злоумышленник допускает, что все метки и цифровые водяные знаки, которые находились в грязном class-файле в момент его компрометации, являются уничтоженными. Результат анализа байт-кода перекомпилированного грязного class-файла, полученного после компиляции вторичного java-файла, представлен на рисунке 3.18.

Листинг 3.7 – Код class-файла полученный с помощью декомпилятора

```
public class TestWatermark {
public static void main(String[] paramArrayOfString) {
    int b1 = 1;
    int b2 = 2;
    int b3 = 3;
    int i = 0;
    System.out.println("Start application");
    System.out.println(b1);
    System.out.println(b2);
    System.out.println(b3);
    System.out.println(i);
    if (b2 > b1) {
        System.out.println("a < b");
    } else {
        System.out.println("b < a");
    }
    if (b3 != b1) {
        System.out.println("c != a");
    } else {
        System.out.println("c = a");
    }
```

```
if (b1 != b2) {
        i = b1 + b2;
} else {
        i = b3;
}
if (i > b3) {
        System.out.println("sum > c");
} else {
        System.out.println("sum a + b = c");}
System.out.println("End application");}}
```

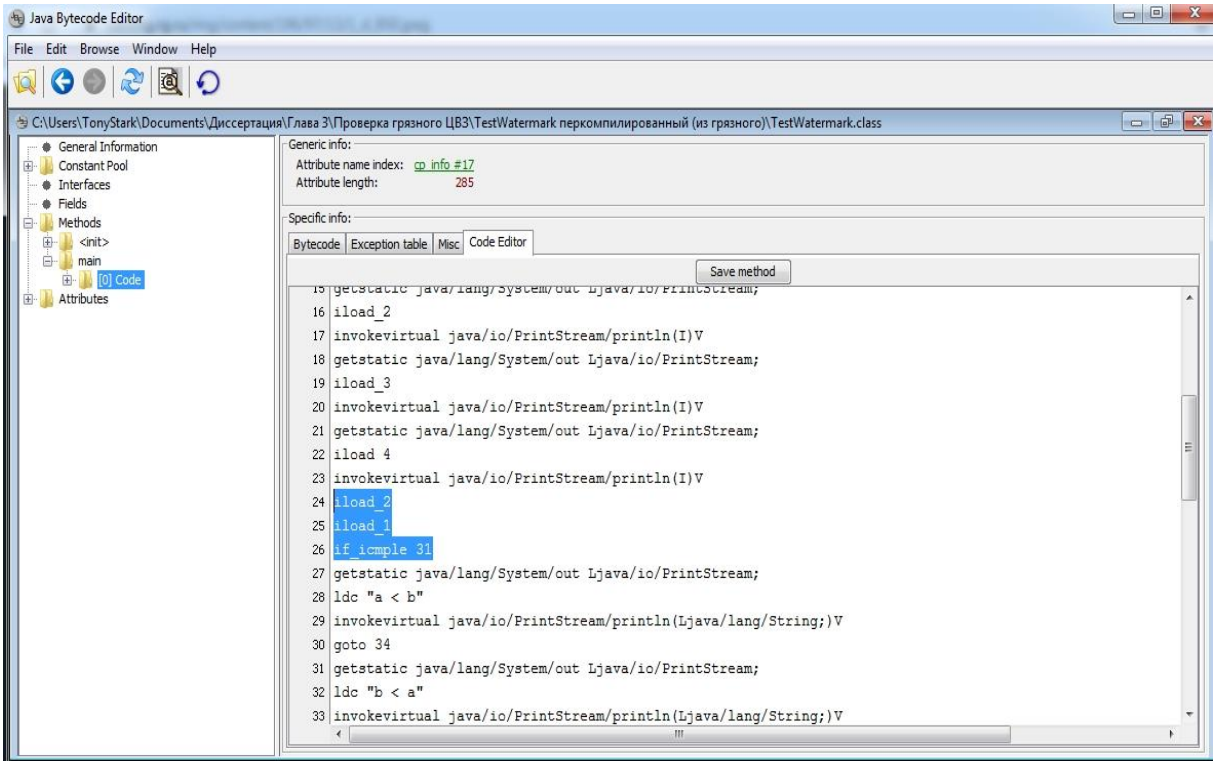


Рисунок 3.18 – Результат анализа байт-кода перекомпилированного грязного class-файла

Результат показывает, что опкоды находятся на ожидаемых позициях, в ожидаемом порядке, замены осуществлены и остались неизменны. Для репрезентативности выборки результатов, грязный class-файл, был декомпилирован с помощью онлайн-сервисов декомпиляции на различных

библиотеках ядра с различными алгоритмами декомпиляции. При декомпиляции «грязного» class-файла с помощью различных онлайн-сервисов предоставляющих открытый доступ к декомпиляторам, были задействованы декомпиляторы:

- Procyon
- CFR
- JDCore
- Jadx
- Fernflower
- JAD (веб-версия ядра DJ Java Decompiler)

После декомпиляции, каждым представленным выше веб-декомпилятором, полученный java-код компилировался. После чего проводился анализ байт-кода, полученного перекомпилированного грязного class-файла. Каждый полученный результат был идентичен предыдущему и все они идентичны результату, описанному выше, полученному с помощью атаки перекомпиляцей через локальный декомпилятор - DJ Java Decompiler.

Результат, отличный от остальных, был получен только при использовании декомпилятора Jadx. Данный декомпилятор не справился с загружаемым в него грязным class-файлом и вывел ошибку о невозможности декомпиляции.

Таким образом, повторяя шаги, описанные в блоке 3.4.1 и листинге 2, злоумышленник создает, по его мнению, совершенно другой class-файл. В какой-то мере, это действительно правда, так как:

- декомпилятор изменил наименования переменных, что означает была применена самая простая обфускация
- новый, перекомпилированный class-файл создан на *основе* java-кода грязного class-файла, однако, не из самого грязного class-файла
- в следствии пунктов выше, изменилась хеш-сумма перекомпилированного class-файла, а также MD5 checksum.

Однако, как демонстрирует рисунок 3.18, цифровой водяной знак, который

- без информационный, не несущий никакую смысловую нагрузку
- не привязанный к структуре class-файла
- созданный во время редактирования байт-кода, без предварительного анализа class-файла
- являющийся адаптационно-случайным, что означает практически отсутствие устойчивости к атакам
- являющийся минимальным, так как была произведена эквивалентная замена только 1 опкода во всем class-файле (около 10% из гарантированно доступных вариантов опкодов к вложению цифрового водяного знака, описанных в блоке 3.4.3, и менее 5% от всех возможных вариантов опкодов доступных к вложению)

является устойчивым к атакам перекомпиляцией на различном оборудовании, различных версиях Java Development Kit, а также различных версиях декомпиляторов. Данные результаты позволяют сделать вывод, о том, что при использовании цифровых водяных знаков более сложных категорий, обладающих более высокими характеристиками устойчивости к атакам, гарантирует сохранение работоспособности программного обеспечения и невозможности уничтожения или повреждения цифрового водяного знака, вложенного в class-файл посредством предлагаемой методики, любыми атаками перекомпиляцией, с помощью находящихся в открытом доступе декомпиляторов Java или в ручном режиме.

3.5. Устойчивость class-файлов при использовании затрудняющих конструкций в коде к декомпиляции

В блоке 3.4 были рассмотрены атаки на цифровой водяной знак посредством перекомпиляции. Полученные результаты позволяют сделать вывод о устойчивости цифрового водяного знака к атакам перекомпиляцией с помощью

декомпиляторов и разности подходов, семантика остается неизменной, а, следовательно, и цифровой водяной знак.

Разработанная методика также включает в себя действие по внедрению шаблонов и функций языка программирования Java, которые затрудняют работу инструментов декомпиляции, в связи с тем, что неверно распознаются декомпиляторами при разборе байт-кода class-файлов. В методике предлагается два варианта использования этого шага: на этапе создания исходного кода, когда разработка сразу ведется с использованием подобных функций и на этапе вложения цифрового водяного знака, когда редактируются существующие методы под затрудняющие функции без изменения логики или создаются фиктивные методы, использующие внутри себя подобные функции.

Не существует правил, обеспечивающих вложенность диапазонов на уровне байт-кода, а, следовательно, они могут перекрываться или даже совместно использовать код с кодом обработчика. Таким образом, один из способов ввести декомпиляторы в заблуждение — перехватывать последовательные участки байт-кода, которые не обязательно являются последовательными в исходном коде Java. Прекрасным примером данного подхода является конструкция переключателя (switch-case). В исходном коде оператор switch инкапсулирует различные блоки кода в качестве целей переключения. Однако в байт-коде нет ничего, что явно привязывало бы инструкцию переключения к различным блокам кода (т. е. явной инкапсуляции не было).

Если функционал оператора switch помещен в диапазон функционала try-catch (с помощью которого возможно избежать внезапного прерывания выполнения приложения в силу того, что ошибка будет «погашена» оператором catch) вместе с частью блоков кода, которые связаны в качестве его целей, то автоматический декомпилятор не сможет вывести семантически эквивалентный код, который хоть как-то похож на исходный код. Он просто воспроизведет данную ловушку для стандартных ошибок в исходном коде в той или иной форме, возможно, путем дублирования кода.

Конструкция switch в байт-коде Java предоставляет эффективный инструмент обфускации потока управления. Она является единственным органичным способом (за исключением структуры try-catch), который позволяет создать граф потока управления, в котором имеется узел с более чем двумя последователями. Это значительно усложняет метод и гарантирует, что никакой декомпилятор не сможет правильно воспроизвести исходный код.

Ниже рассмотрены наиболее часто встречаемые конструкции.

Функционал: Целые числа разбиваются на операции XOR и обрабатываются в стеке с помощью swap, dup_x1и pop2.

Последствия: Декомпилятор не может правильно проанализировать стек, что приводит к изменению кода, а иногда и к недопустимому коду.

Функционал: Разделение блока кода на два переупорядоченных подблока и заключение первого блока в фиктивный блок try-catch.

Последствия: Некоторые декомпиляторы могут произвести декомпиляцию подобного метода, но объединяют второй блок кода в блок try с первым блоком кода, что приводит к изменению поведения. Также, некоторые декомпиляторы не могут восстановить исходные код, что приводит к циклам циклов в результате декомпиляции.

Функционал: Вставка блока try-catch с типом обработчика null (используется для реализации ключевого слова finally).

Последствия: Декомпиляторы не могут распознать использование типа null в блоке catch, что приводит к неточному или совершенно неверному коду с измененной логикой.

Функционал: Для версий JVM 1.8 и более поздних существует возможность добавлять к class-файлам JAR-файла суффикс «/», чтобы вызвать исключение в

инструментах декомпиляции, принуждая их рассматривать class-файл в качестве каталога.

Последствия: Декомпиляторы пропускают такие class-файлы не анализируя их.

Функционал: Перехват инструкции getstatic и пропуск полученного значение в вызов метода.

Последствия: Декомпилирует метод, но перемещает вызов метода внутрь захваченного диапазона. Это приводит к изменению логики.

Функционал: Использование конструкции try-catch для создания потока управления либо с помощью явного оператора throw, либо путем вставки оператора, который, создает исключение. Эта обфускация использует известный факт Java: вызов метода экземпляра для нулевого объекта всегда приводит к исключению NullPointerException.

Последствия: Полное искажение логики приложения и как следствие недостижимый или некорректный код.

Более подробный набор функций затрудняющих декомпиляцию class-файлов java-приложения представлен в таблице 3.4.

Таблица 3.4 – Фрагмент набора функций, затрудняющих декомпиляцию

Функционал	**Описание функционала**
throw null	Перехват инструкции getstatic и передача полученное значение в вызов метода
Добавление к class-файлу суффикса «/»	Инструмент декомпиляции рассматривает данный class-файл, как каталог
Добавление циклического избыточного кода к ZipEntry	JVM не проверяет CRC ZipEntry. Невозможность распаковки jar
Добавление блока try-catch с типом обработчика null	Вызов ошибки в блоке try, которая передает null в пустой блок catch

Разбивка блока кода на два, оба внутри try-catch	Разбивает блок кода на два переупорядоченных под блока и заключает первый блок в фиктивный обработчик
Обращение к несуществующим данным	Добавление в код зависимостей и вызовов к данным, которых не существует
XOR	Целые числа разбиваются на операции XOR
Смещение байтов архива JAR	Изменение заголовков байтов MANIFEST.MF, устанавливая смещения байтов JAR с 0x6 и 0x7 на 0x0 и 0x8
Упаковка локальных переменных в битовые поля	Объединение некоторых переменных и упаковка их в одну переменную с большим количеством бит (например, long)
Замена инструкций if(non)null блоками try-catch	Замена инструкций if(non)null блоками try-catch
Добавление операторов переключения недостижимого кода	Недостижимый блок case содержит блок try-catch
goto в try-catch	Переход через инструкции goto , которые находятся в специальном блоке try
Расширение ресурсов - *.class	Имена неклассовых ресурсов заканчивается на *.class
Несоблюдение соглашений конструктора класса	Оборачивание супер вызова родительского конструктора класса в блок try

Таким образом, возможно сделать вывод о том, что подавляющее большинство конструкций обладает полноценной устойчивостью к декомпиляции в силу того, что обратная интерпретация байт-кода в исходный код несовершенна при наличии сложных конструкций. Декомпиляторы не в состоянии восстановить исходную логику работы class-файла или произвести попытку декомпиляции в силу аварийного завершения выполнения декомпилятора во время анализа class-файла перед процедурой декомпиляции.

3.6. Анализ эффективности методики

Модель Нельсона для оценки надежности java-приложения с ЦВЗ и без ЦВЗ. Java-приложения имеют внутреннюю логику, которая представляет некий результат своей работы. Приложения производят расчеты математических функций, собирают данные, выбирают наилучшие из найденных и различные другие функции. Все java-приложения так или иначе работают с наборами данных внутренними или внешними (входными) E_i.

Таким образом, можно дать следующее математическое определение надежности java-приложения. Надежность java-приложения – это вероятность правильного и безошибочного выполнения n запусков на исполнение java-приложения. Поэтому один запуск на одном наборе входных данных является единичным испытанием java-приложения.

Для оценки надежности необходимо использовать случайные наборы данных среди равновероятных наборов данных. Однако для сравнения надежностей java-приложения до стеговложения и со стеговложением, необходимо производить расчет надежности на строго определенных наборах данных E_i. Вероятность P того, что на выбранном наборе данных E_i при исполнении class-файла или java-приложения будет получен приемлемый результат будет определяется выражением:

$$P = 1 - \sum_{i=1}^{n} e_i y_i \,,$$

где e_i – вероятность или частота использования i-го набора входных данных, y_i – динамическая переменная, которая принимает нулевое значение, если запуск на исполнение заканчивается приемлемым результатом, и значение 1, если прогон заканчивается рабочим отказом или аварийным завершением.

Для сравнения надежности java-приложения со стеговложением и без него достаточно сравнения результатов запусков на исполнение class-файла или java-

приложения с одними и теми же наборами данных, равное количество раз. Таким образом выражение примет вид:

$$P = 1 - \sum_{i=1}^{n} y_i \, .$$

Надежность R(n) java-приложения П будет равна вероятности того, что в последовательности из n запусков на исполнение ни один из них не закончится ошибкой времени выполнения, программной ошибкой или дефектом исполнения java-приложения.

$$R(n) = (1 - P_1)(1 - P_2) \dots (1 - P_n) = \prod_{j=1}^{n} (1 - P_j)$$

Выражение возможно представить в виде:

$$R(n) = \exp\left(\sum_{j=1}^{n} \ln(1 - P_j) \right).$$

Следовательно, при сравнении надежности R(n), class-файл или java-приложение без стеговложения n_1 и class-файл или java-приложение со стеговложением n_2 должны иметь равные надежности $R(n_1) = R(n_2)$. Однако равная надежность class-файла или java-приложения со стеговложением и без не дает полной гарантии неизменности. Соответственно, помимо равенства надежностей необходимо, чтобы была идентичность результатов каждого запуска на исполнение на одном и том же наборе входных данных.

Исходя из выражений, полученных выше и используя входные данные для class-файла, исходный код которого представлен в листинге 3.1, был произведен расчет надежности на тех же входных данных для class-файла, байт-код которого представлен в листинге 3.5. Расчетные значения надежностей class-файла до вложения ЦВЗ и после равны. Также равны результаты запуска class-файлов на исполнение.

Оценка эффективности методики

Для оценки эффективности методики и ее совокупности действий, представляющих собой анализ class-файлов java-приложения, создание ЦВЗ, вложение ЦВЗ, добавление конструкций, затрудняющих декомпиляцию, были выбраны декомпиляторы представленные в блоке 3.2.

Таким образом, в исходный код, представленный в листинге 3.1 блока 3.4, были добавлены различные конструкции, затрудняющие декомпиляцию и после компиляции class-файла, произведено вложение цифрового водяного знака. Манипуляции с исходным кодом проводились несколько раз с разными конструкциями языка программирования Java и получением каждый раз нового class-файла, с целью репрезентативности выборки. В конечном итоге анализ проводился на 200 class-файлах, содержащих в себе различные конструкции, затрудняющие декомпиляцию и цифровой водяной знак.

Эффективность методики оценивалась комплексно, по четырем ключевым параметрам:

- Успешность перекомпиляции class-файла (успешность декомпиляции и возможность повторной компиляции полученного кода)
- Процент искаженной или не компилируемой логики, после успешной декомпиляции (без ошибок в работе декомпилятора)
- Процент случаев аварийного завершения работы декомпилятора
- Процент случаев в которых был удален или поврежден цифровой водяной знак

Все полученные 200 class-файла подвергались процедуре декомпиляции каждым ранее представленным декомпилятором. Результат анализа представленный в таблице 3.5, демонстрирует процентное соотношение декомпиляторов не справившихся с декомпиляцией class-файлов по различным причинам к общему числу используемых декомпиляторов.

Таблица 3.5 – Анализ эффективности разработанной методики

Функционал	**Успешность перекомпиляция**	**Процент искаженной / не компилируемой логики**	**Аварийное завершение работы декомпилятора**	**ЦВЗ удален / поврежден**
throw null	0 %	90 %	10 %	0 %
Добавление к class-файлу суффикса «/»	0 %	0 %	100 % игнорирование class-файлов	0 %
Добавление Циклического избыточного кода к ZipEntry	30 %	0 %	70 %	0 %
Добавление блока try-catch с типом обработчика null	0 %	90 %	10 %	0 %
Разбивка блока кода на два, оба внутри try-catch	0 %	45 %	55 %	0 %
Обращение к несуществующим данным	50 %; Возможно аварийное завершение работы JVM	0 %	50 %	0 %
Разбиение XOR	10 %	90 %	0 %	0 %
Расширение ресурсов - *.class	18 %	40 %	42%	0 %
Смещение байтов архива JAR	14 %	0 %	86 %	0 %
Упаковка локальных	0 %	70 %	30 %	0 %

переменных в битовые поля				
Замена инструкций if(non)null блоками try-catch	0 %	95 %	5 %	0 %
Добавление операторов переключения недостижимого кода	0 %	90 %	10 %	0 %
goto в try-catch	0 %	100 %	0 %	0 %
Несоблюдение соглашений конструктора класса	0 %	100 %	0 %	0 %

В таблице 3.5 цветом выделены функции или языковые конструкции, использование которых может повлечь за собой успешную перекомпиляцию class-файла, в исключительных случаях. Однако таких конструкций 5 из 14 и у каждой из них высокий процент аварийного завершения работы декомпилятора. В совокупности с другими конструкциями эффект будет усилен, что позволит гарантировать невозможность корректной декомпиляции class-файлов java-приложения. Также, необходимо заметить, что ни одним из представленных в исследовании декомпиляторов разрушить или повредить цифровой водяной знак не представляется возможным.

Точное сравнение с другими методиками затруднительно, однако, на основе имеющихся результатов исследования, можно сделать предположение о том, что среди существующих методик будет большой процент успешной декомпиляции и уничтожения или повреждения цифрового водяного знака, в связи с тем, что подавляющее большинство методик не учитывает фактор перекомпиляции на различных версиях JDK, а также не используют принцип создания цифрового

водяного знака на основе байт-кода с использованием конструкций затрудняющих декомпиляцию.

Выводы

Раздел содержит основные научные результаты, полученные в работе и направленные на разрешение актуальной научной задачи исследования.

1) Рассмотрена возможная модель действий нарушителя для компрометации class-файлов java-приложения. Проведен анализ доступных в открытых источниках декомпиляторах Java файлов. Исследована возможность применения паттернов проектирования на этапе разработки java-приложения или этапе вложения цифрового водяного знака. Результаты этих исследований опубликованы в работах [34, 44].

2) Проанализирован скомпилированный class-файл, содержащий в себе измененный байт-код. Доказана неизменность в логике работы java-приложения, после редактирования байт-кода class-файла и его повторной компиляции. Исполняемый файл соответствует заявленным характеристикам, сохраняется размер. Проанализированы определенные конструкции и шаблоны проектирования кода. Продемонстрировано, что при наличии в исходном коде определенных конструкций повышаются вероятность: некорректной декомпиляции class-файлов java-приложения, невозможности повторной компиляции декомпилированного кода, аварийное завершение работы декомпилятора. Результаты этих исследований опубликованы в работах [45, 46].

3) В качестве научного результата, интегрально объединяющего вышеизложенные результаты, и позволяющего обеспечить достижение цели исследования была разработана методика создания и вложения цифрового водяного знака в байт-код class-файлов java-приложений устойчивого к атакам декомпиляцией, за счет использования расширенного набора опкодов для эквивалентных замен и реализации посредством них конструкций и шаблонов проектирования в коде, которые препятствуют декомпиляции class-файла и

удалению цифрового водяного знака из java-приложения. К элементам научной новизны данной методики относится следующее:

а) для вложения цифрового водяного знака в байт-код class-файлов java-приложения используется комплексный анализ всех class-файлов и их взаимосвязей в java-приложении;

б) используются шаблоны проектирования и методы языка программирования Java в совокупности, не позволяющие произвести декомпиляцию java-приложения или разрушить цифровой водяной знак в его class-файлах атакой декомпиляцией;

в) для вложения цифрового водяного знака используется расширенный набор операционных команд пригодных для эквивалентных замен, что позволяет цифровому водяному знаку являться неотъемлемой частью java-приложения.

Данная методика и ее отдельные элементы опубликованы в работах [31, 34, 45, 46, 153].

Разработанная в главе 3 методика создания и вложения устойчивого к атакам декомпиляцией цифрового водяного знака в class-файлы java-приложения позволяет снизить вероятность успешной декомпиляции java-приложения и его class-файлов на 60-80%, позволяет увеличить количество отказов при нелегитимной декомпиляции java-приложения, увеличить устойчивость цифрового водяного знака к атакам перекомпиляцией до 80-95% от общего объема цифрового водяного знака.

Разработанная методика позволяет:

- Произвести вложение цифрового водяного знака устойчивого к разрушению атаками декомпиляцией class-файлов
- Проводить оценку целостности class-файлов на уровне java-приложения
- Использовать расширенный набор эквивалентных опкодов, для вложения цифрового водяного знака в class-файлы java-приложения

- Учитывает, при проведении анализа class-файлов, количество эквивалентных опкодов и объем class-файла, что позволяет производить вложение ЦВЗ адаптационно под конкретный class-файл
- Минимизирует вероятность пропуска class-файла с частью ключевой логики
- Использовать легитимные шаблоны и функции написания методов на языке программирования Java для затруднения декомпиляции class-файлов java-приложения
- Может быть преобразована в алгоритм для программы на ЭВМ

За счет проведенных экспериментов продемонстрировано:

- Устойчивость class-файлов к атакам перекомпиляцией
- Устойчивость ЦВЗ к атакам декомпиляцией

Материалы главы 3 были представлены в материалах международных и российских научно-технических конференциях [153] и опубликованы в изданиях, включенных в перечень ВАК России [31, 34, 45, 46], а также в изданиях Scopus [153]. Статьи «Исследование устойчивости цифрового водяного знака к атакам направленной декомпиляции составных частей приложения Java» и «Методика обфускации байт-кода java-приложения с целью его защиты от атак декомпиляцией» опубликованы в изданиях, включенных в перечень ВАК России, без соавторства.

Глава 4. Методика создания и вложения устойчивого к атакам обфускацией цифрового водяного знака в class-файлы информационной системы

4.1 Модель злоумышленника информационной системы

Для понимания возможных угроз со стороны злоумышленника, необходимо построить его модель. Под моделью нарушителя понимается перечень вариантов действий возможного нарушителя для достижения им цели компрометации информационной системы, а также его квалификация, возможные инструменты и ресурсы.

Под информационной системой понимается некий комплекс исполняемых java-приложений и class-файлов, выполняющих различные функции, например, функцию подсчета суммы операции или функцию оплаты [15]. Например, любой веб-сервис, предоставляющий услуги, попадает под вышеописанное определение информационной системы.

Первым приоритетом злоумышленника может являться не вся информационная система, а ее конкретный сервис, java-приложение или даже несколько конкретных class-файлов java-приложения, которое отвечает за проведение платежей. Возможная модель злоумышленника информационной системы продемонстрирована на рисунке 4.1.

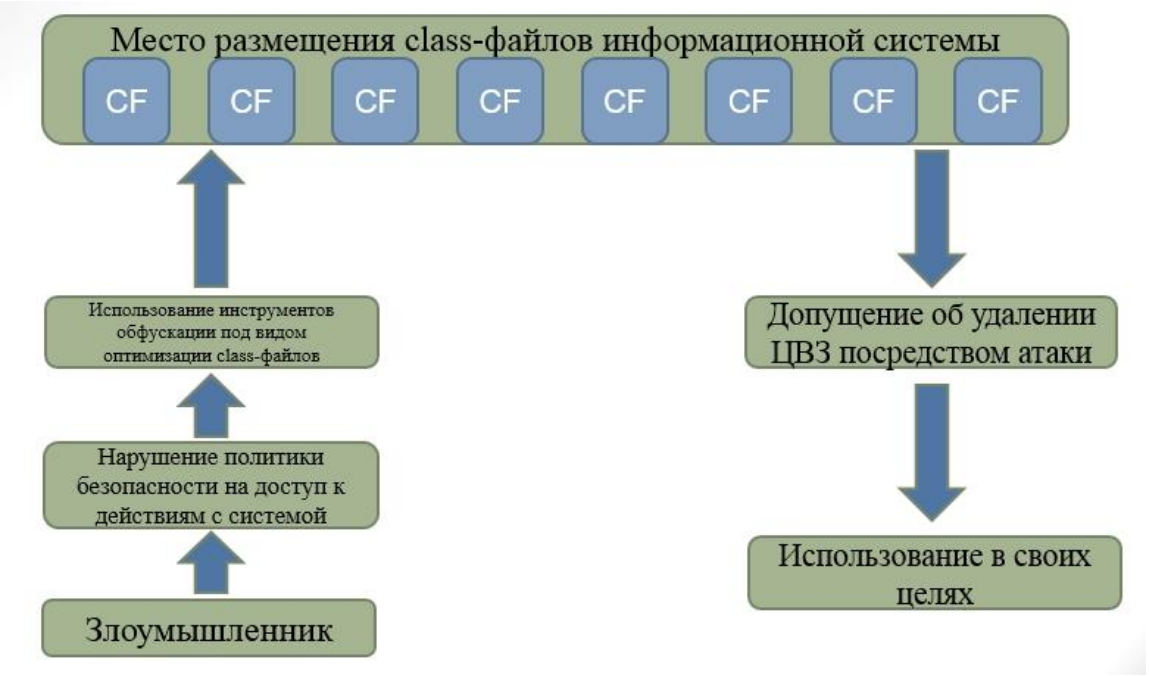


Рисунок 4.1 – Модель злоумышленника информационной системы

Нарушив каким-либо образом политики безопасности и получив доступ к информационной системе, конкретному java-приложению и его class-файлам, отвечающим за функции оплаты, злоумышленник может произвести атаку обфускацией под видом санкционированной оптимизации кода с целью увеличения скорости исполнения java-приложений или отдельных class-файлов. Такие процедуры являются вполне распространенной и легитимная практикой уже функционирующих информационных систем. В некоторых случаях, нет необходимости получения доступа к отдельным class-файлам, достаточно наличия прав доступа на запуск инструмента обфускации кода с возможностью выбора параметров и целей обфускации в ручном режиме. Таким образом предприняв попытку очистить украденные class-файлы от различных меток и цифровых водяных знаков.

Во время атаки обфускацией злоумышленник изменяет исходный java-код, предполагая, что параметры инструмента обфускации, выставленные с заметным превышением необходимого уровня сложности процедуры, повреждают или удаляют цифровой водяной знак компании, вложенный в class-файлы java-приложения информационной системы [33].

После успешной атаки обфускацией, злоумышленник имеет возможность произвести кражу java-приложений или конкретных class-файлов их составляющих. В скомпрометированных class-файлах подменяя оригинальные функции оплаты на свои, а также изменяя конфигурации. После чего производится внедрение скомпрометированного java-приложения или его составных class-файлов в информационную систему. Также, распространенным методом является операция фарминга, когда java-приложение размещается на другом хостинге, а в информационной системе, после компрометации оставляется незаметное вложение-изменение, перенаправляющее пользователей информационной системы на интернет адрес, содержащий java-приложение оплаты злоумышленника [156].

Рассмотрим каждый блок модели нарушителя в отдельности.

1. Злоумышленник получает доступ к информационной системе и к действиям с ней, например, посредством различных инструментов.

2. Злоумышленник, используя полученные несанкционированным образом права и инструменты взаимодействия на код информационной системы, совершает атаку обфускацией на интересующие его части информационной системы.

3. Совершается предположение, что продвинутые способы обфускации и превышение допустимой сложности обфускации повреждает или удаляет цифровой водяной знак в class-файлах java-приложений.

4. Кража class-файлов или конкретных java-приложений информационной системы.

5. Изменение байт-кода class-файлов или java-приложений с целью замены исходных функций на функции, созданные злоумышленником.

6. Возможно, при необходимости, дополнительная атака перекомпиляцией. В таком случае злоумышленник предполагает большую вероятность уничтожения цифрового водяного знака, а также используется менее ресурсоёмкий способ редактирования исходного кода class-файла.

7. Завершающим действием будет являться внедрение скомпрометированной части информационной системы обратно или её использование для процедуры фарминга.

Для успешности действий злоумышленника, необходимо, чтобы жертва решила воспользоваться именно данной информационной системой, которую изменил злоумышленник для своих нужд. В связи с этим злоумышленники не производят изменений пользовательского интерфейса для неотличимости от оригинальной информационной системы, а производят подмены во внутренних алгоритмах информационной системы. После чего запускается процедура фарминга, перенаправляющая запросы от оригинальной информационной системы в информационную систему, реализованную злоумышленником. Таким образом, пользователь попадает в копию официальной информационной системы, которая только выглядит, как оригинальная. Любые действия пользователя с такой информационной системой влекут риски, тем более при использовании функций оплаты или передачи реквизитов.

Таким образом, исходя из вышеописанной модели злоумышленника, можно сделать предположения. Злоумышленнику, вне зависимости от целей атаки на информационную систему необходимо получить доступ к class-файлам или java-приложениям информационной системы, после чего произвести обфускацию, редактирование, подмену необходимого количества. После чего проводить процедуру фарминга или любую другую. Исходя из вышеописанного, делается вывод, что при любой атаке на информационную систему с целью фарминга, злоумышленнику придется производить манипуляции с class-файлами java-приложений данной информационной системы. Таким образом, необходимо, чтобы существовало, как минимум одно стороннее java-приложение осуществляющее постоянный мониторинг информационной системы и ее составных частей на наличие цифрового водяного знака.

Пример взаимодействия java-приложения, осуществляющего функцию проверки и периодического мониторинга и структуры информационной системы,

ее составных частей, на наличие цифрового водяного знака, представлен на рисунке 4.2.

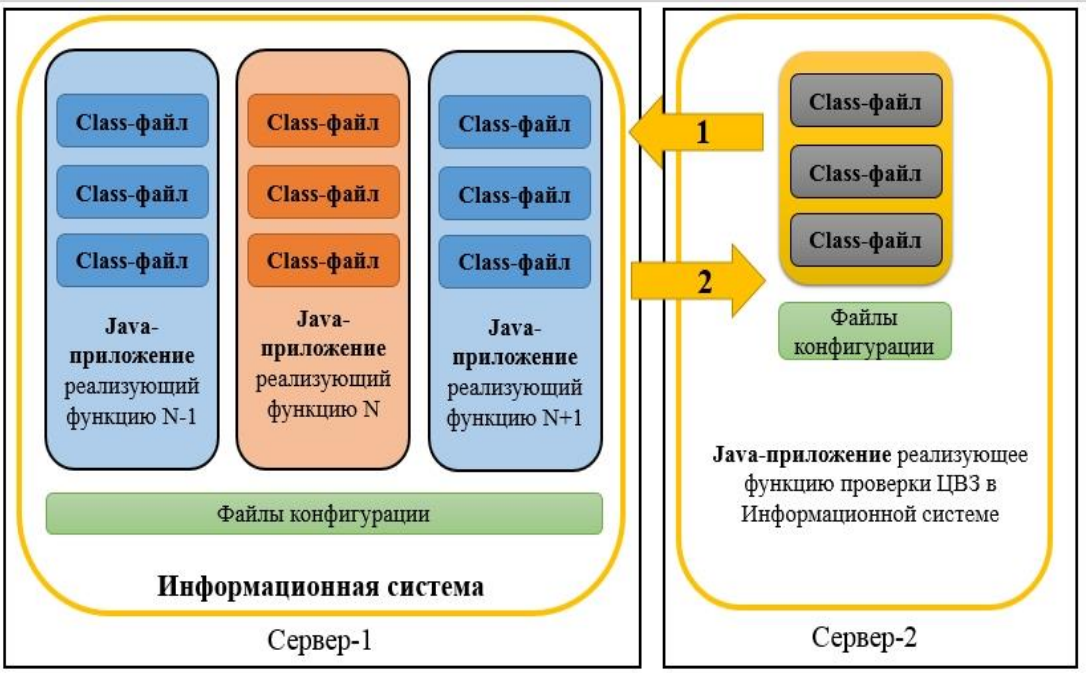


Рисунок 4.2 – Пример взаимодействия java-приложения, реализующего функцию проверки наличия ЦВЗ в структуре информационной системы и ее составных частях.

Следовательно, при сохранении после атаки обфускацией самого примитивного, не структурированного адаптационного-случайного цифрового водяного знака или невозможность информационной системы, ее части в виде java-приложения в дальнейшем функционировать, как прежде с той же скоростью, эффективностью или вообще функционировать, будет возможным вывод о устойчивости информационной системы к любым атакам обфускацией и подмены отдельных java-приложений или их class-файлов, которая содержит в себе цифровые водяные знаки с более высокими характеристиками устойчивости к атакам.

4.2 Методика создания и вложения устойчивого к атакам обфускацией цифрового водяного знака в class-файлы информационной системы

Для повышения сложности получения злоумышленником исходного java-кода class-файлов информационной системы необходимо увеличить устойчивость цифрового водяного знака, вложение которого производится в class-файлы информационной системы. Для этого возможно применить методику создания и вложения цифрового водяного знака устойчивого к атакам обфускацией направленных на деструктуризацию покрывающего сообщения или уничтожение вложения. Пример информационной системы, представляющей собой совокупность java-приложений, выполняющих различные функции, но объединенных между собой для выполнения конкретной цели, приведен на рисунке 4.3.

Таким образом, исходя из возможных моделей злоумышленника можно сделать предположение о том, что целью любых атак злоумышленника является некоторая часть информационной системы, которая позволит злоумышленнику получить выгоду в то или ином виде [143]. В связи с тем, что правообладатели и создатели информационной системы, могут оперировать только допущениями на основе своей экспертизы, о возможных действия и истинных мотивах злоумышленника, появляется необходимость защиты исходного кода каждого блока java-приложения информационной системы от кражи/подмены или редактирования.

Так как информационная система является совокупностью java-приложений, сконфигурированных для выполнения определенных задач, то меняется распределение ключевой логики в class-файлах. Поэтому ранее предложенные в данной работе методики могут быть менее эффективны для полноценной информационной системы, которая имеет распределенную логику и слабую связанность совокупности java-приложений.

Информационная система представляет собой набор java-приложений со слабой связью, с целью отказоустойчивости архитектуры, каждое из которых

реализует одну конкретную бизнес-функцию. В связи с вышеизложенным требуется дополнительный цифровой водяной знак, который будет связующим и подтверждающим звеном при проверке целостности информационной системы [3, 38]. Таким образом, необходимо модифицировать ранее разработанную методику под архитектуру информационной системы.

Как уже отмечалось выше, вложение тиражируемого цифрового водяного знака в class-файлы информационной системы может иметь существенные недостатки:

- Проверка целостности информационной системы будет проводится в один этап, за который будет осуществлена проверка абсолютно всех class-файлов каждого java-приложения информационной системы, что потребует большого количества ресурсов, как вычислительных, так и временных.
- Чаще всего, целью злоумышленников является один-два конкретных модуля информационной системы. С целью уничтожения в них цифрового водяного знака, компрометации, редактирования с подстановкой своих файлов конфигураций или функций и использование полученных модулей в фарминговых целях, либо подмена оригинальных в информационной системе.

Из вышеописанного следует, что необходимо осуществлять проверку целостности в два этапа, для более эффективной локализации проблемы. Исходя из принципа «от большего к меньшему». Таким образом, сначала проверяя цифровой водяной знак регистратор, который позволяет удостоверится в том, что при отсутствии деструктуризации данного цифрового водяного знака все java-приложения составляющие информационную систему не были подвержены разрушающим атакам или подменам [43]. В противном случае, при отсутствии цельного ЦВЗ-регистратора в информационной системе, необходимо запустить второй этап проверки, который включает анализ всех class-файлов тех java-приложений, в которых во время первого этапа проверки не было обнаружено частей регистрирующего цифрового водяного знака.

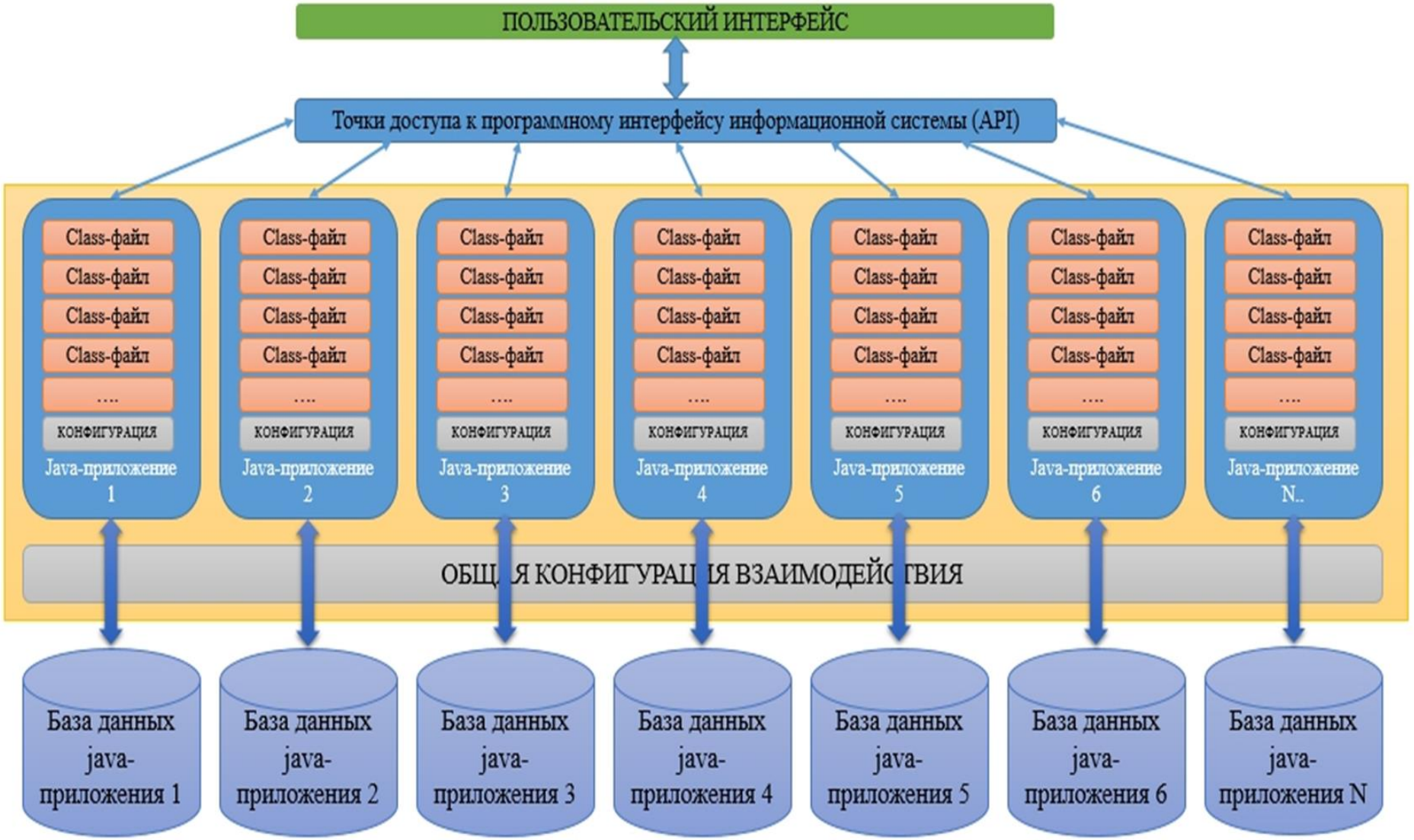


Рисунок 4.3 – Пример информационной системы java-приложений

Методика создания и тиражируемого вложения, устойчивого к атакам обфускацией направленных на деструктуризацию покрывающего сообщения или уничтожение, цифрового водяного знака в class-файлы информационной системы, представляемой совокупностью java-приложений, состоит из четырех этапов: анализ информационной системы, анализ каждого входящего в ИС java-приложения, создание и вложение ЦВЗ на основе анализа структуры class-файла java-приложения, создание и вложение цифрового водяного знака регистратора на основе структуры и интеграции ключевых java-приложений и их class-файлов. Методика продемонстрирована на рисунке 4.4.

Этап анализа информационной системы.

На данном этапе производится анализ информационной системы, как единого продукта, комплекса программных средств. Анализируются:

- интеграционные взаимосвязи компонентов информационной системы, java-приложений
- использование конфигурационных файлов
- интеграция java-приложений с базами данных
- шины данных передачи служебных сообщений
- взаимодействие внутренних сервисов информационной системы посредством архитектурного паттерна REST API.

Составляется карта взаимодействия компонентов информационной системы. Подробно исследуется взаимосвязь java-приложений внутри информационной системы, производится разделение на ключевые приложения и вспомогательные.

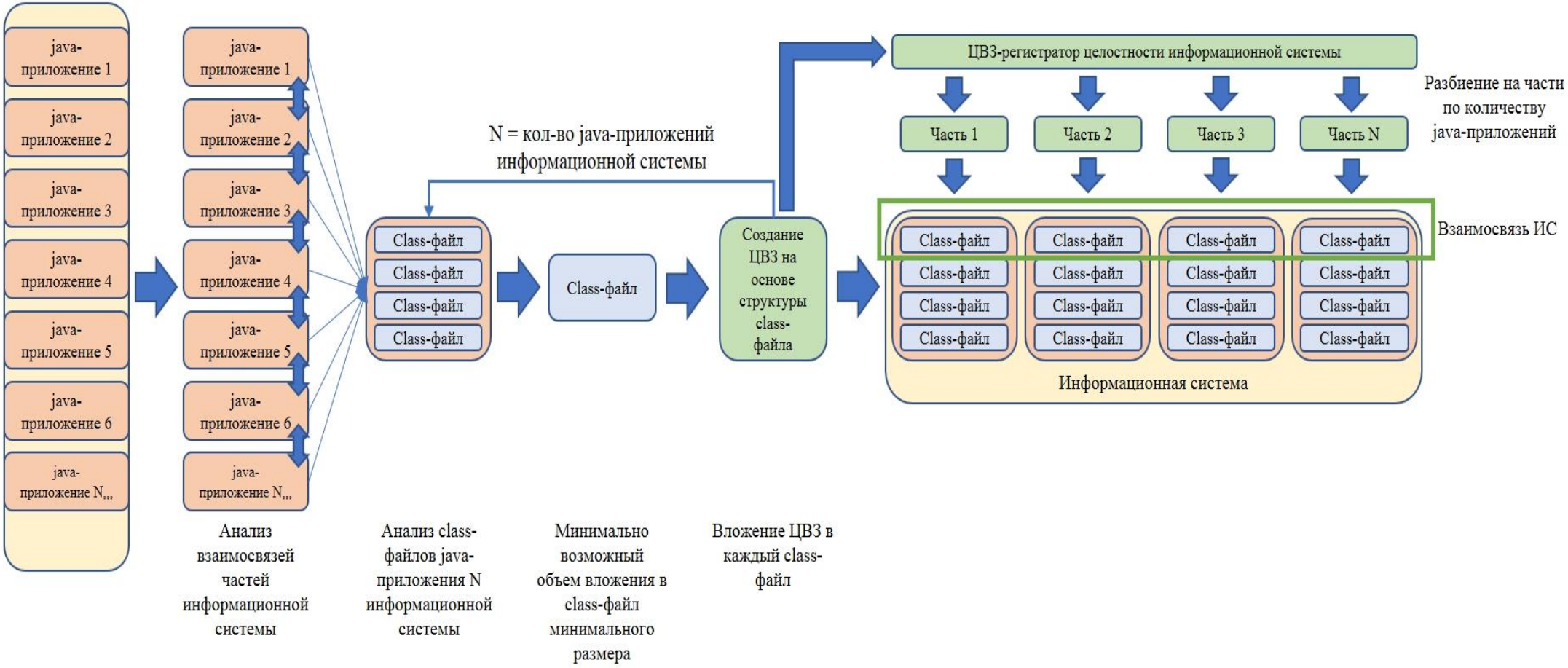


Рисунок 4.4 – Методика создания и вложения цифрового водяного знака регистратора в информационную систему

Этап анализа java-приложений информационной системы.

На данном этапе производится анализ каждого java-приложения, являющегося частью информационной системы, его бизнес-функций, логики и интеграции с взаимодействующими другими java-приложениями. Анализ включает:

- Исследование интеграционного взаимодействия java-приложения с другими java-приложениями информационной системы
- Выявление категории значимости java-приложения для информационной системы (содержит ключевую логику, влияет на работоспособность всей ИС, большое количество внешних интеграций, обрабатывает выходные данные ключевых java-приложений в исключительных моментах и т.д.)
- Анализ связей class-файлов в java-приложении
- Выявление class-файлов с ключевой логикой java-приложения
- Выявление class-файлов ответственных за интеграционное взаимодействие с ключевыми java-приложениями
- Ранжирование class-файлов по количеству опкодов доступных к эквивалентным заменам
- Нахождение маркерного class-файла (самый меньший объем в class-файле, минимально доступное к эквивалентным заменам количество опкодов в байт-коде).

Этап создания и вложения цифрового водяного знака на основе структуры байт-кода class-файла.

Этап создания цифрового водяного знака для class-файлов java-приложений информационной системы аналогичен этапам из ранее разработанных методик, но с отличиями в используемых опкодах. На данном этапе используется заранее подготовленное ключевое слово, хеш-сумма class-файла или любая другая информация, которую возможно привести к бинарному представлению. Опкоды,

которые доступны для осуществления операций эквивалентных замен представляются в виде последовательности бит, полученной из hex-представления данных опкодов в виртуальной машине Java. Из последовательности опкодов доступных для эквивалентных замен исключается заранее оговоренная часть: 1 опкод, 10% опкодов, в середине последовательности, в начале, в конце, но последовательно.

После чего битовая последовательность, полученная из цифрового водяного знака, делится на байтовые блоки. Каждый байтовый блок проверяется на наличие в битовой последовательности пригодных для эквивалентных замен группе опкодов class-файла. При удовлетворении условий наличия данного байта информации в последовательности бит и необходимого местоположения (после предыдущего бита) производится проверка следующего байта.

Те байты, которых не оказалось в битовой последовательности запоминаются. При наличии хотя бы одного не записанного внутрь битовой последовательности опкодов байта цифрового водяного знака, начинается цикличный перебор возможных эквивалентных опкодов каждого опкода битовой последовательности для получения битовой последовательности включающей все биты цифрового водяного знака. Данный этап эффективен при его реализации в виде алгоритма.

После получения битовой последовательности, которая содержит в себе битовую последовательность цифрового водяного знака, последовательность делится на байтовые блоки, которые в свою очередь преобразуются в мнемоническое представление операционных кодов, которые в той же последовательности необходимо эквивалентно заменить в исходном class-файле.

Этап создания и вложения цифрового водяного знака регистратора.

Данный этап основывается на class-файлах содержащих ключевую логику java-приложения. Цифровой водяной знак, который может являться цифровым следом информационной системы, зашифрованной последовательностью хеш-сумм каждого java-приложения информационной системы, криптографической

хеш-суммой, переводится в бинарное представление, полученная битовая последовательность делится на байтовые блоки, байтовые блоки поровну делятся между всеми java-приложениями информационной системы.

Отобранные на предыдущих этапах class-файлы включающие ключевую логику и логику интеграционного взаимодействия между java-приложениями анализируются на предмет возможности включения байтовых блоков цифрового водяного знака регистратора.

Если битовая последовательность class-файла в ранее оговоренной и не тронутой области, на этапе создания в вложения цифрового водяного знака в class-файлы, позволяет произвести эквивалентные замены для вложения части ЦВЗ-регистратора, то данные эквивалентные замены совершаются. В противном случае используется расширенный набор опкодов возможных к эквивалентным заменам и создаются фиктивные инициализации переменных в заранее определенном месте class-файлов, бинарное представление которых сразу отвечает требованиям текущего байтового блока ЦВЗ-регистратора. После чего производится переход в следующему java-приложению информационной системы.

Таким образом, разработанная методика использующая совокупность эквивалентных замен нескольких групп опкодов, двух разно уровневых цифровых водяных знаков обладающих малой связностью друг с другом, позволяет произвести в кратчайшие сроки проверку целостности информационной системы и локализовать скомпрометированную часть с точностью до class-файла. Также, в следующих блоках данного раздела будет продемонстрирована, что разработанная методика обеспечивает устойчивость цифрового водяного знака к атакам обфускации. Также, при использовании продвинутых инструментов и методов обфускации с целью уничтожения или повреждения цифрового водяного знака, значительно повышается риск приведения в неработоспособное состояние скомпрометированного class-файла.

4.3 Исследование эффективности атак обфускацией на цифровой водяной знак

Необходимо рассмотреть существующие на данный момент обфускаторы исходного кода class-файлов java-приложения объединенных в информационную систему. В связи с тем, что обфускаторы были придуманы для защиты исходного кода и затруднения его понимания или редактирования при краже злоумышленником, использование обфускаторов в качестве инструмента атак на части информационной системы содержащих цифровой водяной знак ранее не рассматривалось. Таким образом, необходимо осуществить обзор каждого, находящегося в открытом доступе, обфускатора на предмет возможности атак на цифровой водяной знак, содержащийся в class-файлах.

Оценка устойчивости ЦВЗ к атакам обфускацией

Вероятность наступления события «D» (Destruction), разрушения обфускацией ЦВЗ, рассчитывается по формуле Бернулли:

$$P_D = \mathrm{C}_N^K p^K (1-p)^{N-K},$$

где C_N^K – удачные комбинации замен опкодов байт-кода class-файла в бинарной последовательности пригодных к эквивалентным заменам опкодов,

p – вероятность наступления события,

K – количество вложений,

N – количество разрушенных ЦВЗ.

Для определения вероятности разрушения всех цифровых водяных знаков в class-файлах информационной системы, когда N == K, необходимо использовать формулу:

$$P_D = p^K (1-p)$$

Устойчивость цифрового водяного знака к разрушающим атакам обфускацией рассчитывается по следующей формуле:

$$P(Y) = 1 - \frac{D}{K},$$

где D – число экспериментов, в которых атаки приводили к разрушению ЦВЗ,

K – число экспериментов, в которых проводились атаки обфускацией на ЦВЗ в class-файлах информационной системы.

Метрика Мак-Кейба для оценки сложности потока управления java-приложения в информационной системе. Логику работы class-файла или java-приложения можно представить в виде графа потока управления. Для того, чтобы определить или сравнить сложность потока управления java-приложения с цифровым водяным знаком и без него, необходимо в качестве основы использовать представление java-приложения в виде графа управления, где операции определяются как вершины графов. Таким образом, формула для оценки цикломатической сложности потока управления, будет иметь вид:

$$V = e - n + 2p,$$

где e - количество ребер в графе, n - количество вершин (узлов) в графе, p - число компонент связности графа потока управления. Для правильного приложения с одной точкой входа и одной точкой выхода p = 1 (т.к. достаточно замкнуть граф потока управления одной дугой (вершина входа и вершина выхода из java-приложения соединяется одним ребром) из точки выхода в точку входа) формула будет иметь вид:

$$V = e - n + 2$$

Таким образом, если для java-приложения информационной системы с ЦВЗ q_1 и java-приложения информационной системы без ЦВЗ q_2 применить метрику Мак-Кейба, то итоговая цикломатическая сложность $V(q_1)$ и $V(q_2)$ не должна

различаться. Следовательно, после вложения цифрового водяного знака в class-файл или java-приложение у него не должно увеличиться или уменьшиться количество ребер, вершин и компонент связности графа потока управления.

Метрика Мак-Клура для оценки сложности class-файла информационной системы. Данная метрика основана на числе возможных путей исполнения java-приложения информационной системы, а также числе управляющих конструкций и переменных. Для расчета метрики выделяется три этапа: вычисление значения сложностной функции для каждой управляющей переменной class-файла информационной системы, вычисление сложностных функций для всех частей (модулей) информационной системы, вычисление общей сложности иерархической схемы разбиения информационной системы на части.

Цифровой водяной знак используется в первую очередь на уровне опкодов условных переходов, циклов, инициализации типов переменных. Также необходимо произвести точечную оценку изменений каждого class-файла ИС после стеговложения. Следовательно, нет необходимости использовать все три этапа расчета сложности метрики Мак-Клура. Для оценки сложности class-файла будем применять только расчеты первого этапа метрики.

$$C(o) = \frac{(D(o) + J(o))}{n},$$

где D(o) - величина, измеряющая сферу действия операционного кода o. J(o) - мера сложности взаимодействия функциональных частей class-файла или java-приложения с помощью данного опкода o, n - число отдельных частей в схеме разбиения class-файла на функциональные блоки.

Таким образом, после вложения цифрового водяного знака в class-файл все три параметра выражения должны остаться неизменны. Иными словами, цифровой водяной знак не должен нарушить логическую связность частей class-файла или ИС, не должен изменить сферу действия опкода и количество функциональных частей class-файла или ИС.

4.3.1 Анализ возможностей, существующих обфускаторов class-файлов Java

Существует два основных подхода к обфускации: простая обфускация исходного кода java-программы и обфускация исходного кода с изменением его логики. Обфускация с изменением логики включает в себя добавление в код ветвлений, циклов, различных «пустых» и обратных функций, вставки недостижимого кода и т.д. Таким образом, бинарный код class-файлов java-приложения прошедших через усложненный подход обфускации будут сильно различаться.

Существует несколько методик обфускации данных:

- **Обфускация имен.** Замена удобочитаемых наименований в коде трудными для расшифровки альтернативами
- **Обфускация потока управления.** Модификация логической структуры кода, чтобы сделать ее менее предсказуемой и отслеживаемой
- **Арифметическая обфускация.** Преобразование простых арифметических и логических выражений в сложные эквиваленты
- **Виртуализация кода.** Преобразование реализации метода в инструкции для случайно сгенерированных виртуальных машин.

Самые часто используемые и наиболее эффективные стратегии обфускации кода включают:

- Переименование классов, полей, методов, библиотек и т. д.
- Преобразование арифметических и логических выражений
- Шифрование строк, классов и т. д.
- Сокрытие вызовов конфиденциальных API и т. д.
- Удаление определенных метаданных
- Изменение структуры кода

Практически все существующие обфускаторы эффективно работают с методиками обфускации имен, арифметическими операциями (перестановки). Малый процент работает с методикой обфускации потока управления, модифицируя логику программы таким образом, чтобы результат работы файла, программы или информационной системы оставался прежним, но код был изменен [3, 15]. Процесс обфускации потока управления продемонстрирован на рисунке 4.5. Не находящиеся в открытом доступе и узкоспециализированные обфускаторы поддерживают работу по методике виртуализации кода.

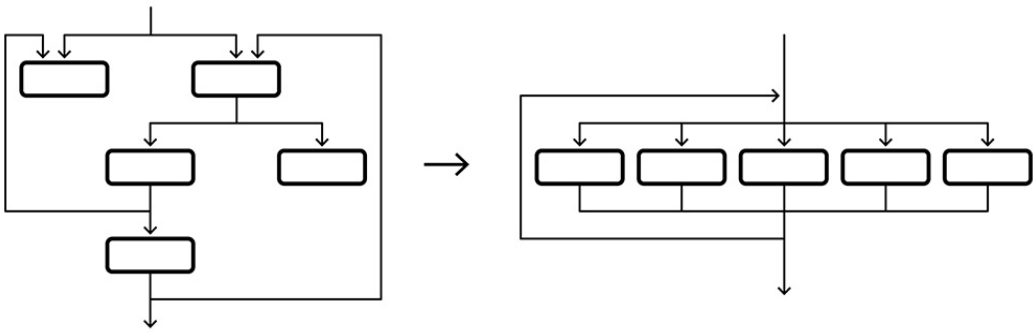


Рисунок 4.5 – Пример обфускации потока управления

ProGuard [89]. Наиболее популярный и бесплатный обфускатор. Изначально создавался, как инструмент для оптимизации исходного кода написанного на языке программирования Java. Инструмент командной строки, который уменьшает размер приложения за счет сокращения байт-кода и обфускации имен классов, полей и методов. В связи с тем, что инструмент производит оптимизацию кода происходит его «сжатие», таким образом, что приложение занимает меньший объем, чем до обфускации. Работа обфускатора основана на четырех этапах:

- На этапе «сжатия» обнаруживаются и удаляются неиспользуемые классы, поля, методы и атрибуты
- На этапе работы оптимизатора оптимизируется байт-код и удаляются неиспользуемые инструкции
- На этапе обфускации остальные классы, поля и методы переименовываются, используя короткие бессмысленные имена

- На этапе предварительной проверки к class-файлам добавляется информация о предварительной проверке, необходимая для Java Micro Edition, а также для Java 6 и более поздних версий.

Практически не имеет возможности эффективной работы с Reflections API. Не реализованы алгоритмы продвинутой обфускации, что позволяет рассматривать данный инструмент только, как высококлассную оптимизацию исходного кода [141, 143].

yGuard [84]. Обфускатор с открытым исходным кодом. С функцией «сжатия» кода и его оптимизации. Существует возможность обрабатывать class-файлы, скомпилированные любым стабильным JDK (до Java 17). Обфускатор yGuard предоставляет возможности для переименования классов, интерфейсов, методов и т. д., чтобы обфуцировать полученный архив, что значительно усложняет реверс инжиниринг. Также, в данном обфускаторе присутствует функция «сжатия» кода, которая уменьшает размер class-файла обеспечивая его более быстрый запуск за счет удаления мертвого кода или не используемого байт-кода.

Не переименовывает тип передаваемых в метод аргументов или объектов, что позволяет быстрее понять обфуцированный код. Отсутствует эффективная работа с логикой java-приложения, все функции обфускатора сосредоточены вокруг замен имен пакетов, классов, методов и полей на невыразительные символы. Логика остается неизменной, как и байт-код class-файлов java-приложений.

JODE [110]. Является java-пакетом, в первую очередь, для оптимизации кода. Способы работы JODE с исходным кодом java включают в себя поддержку операций:

- Переименование имен классов, методов, полей и локальных переменных в более короткие запутанные или уникальные наименования, или в соответствии с заданной таблицей перевода
- Удаление отладочной информации

- Удаление мертвого кода (классов, полей, методов) и константных полей
- Оптимизация размещения локальных переменных

На текущий момент устарел и не поддерживает новые версии JDK и JVM.

JavaGuard [102]. Обфускатор байт-кода общего назначения, разработанный для того, чтобы легко вписаться в процесс сборки и тестирования, обеспечивая уверенность в том, что код Java защищен от декомпиляции и других форм реверс инжиниринга.

На текущий момент обфускатор устарел и не поддерживается. Нет возможности работы с java-приложениями и class-файлами, скомпилированными на версиях JDK после 2016 года. Имеются ошибки при работе с функциями ввода-вывода информации в текстовые файлы.

jarg [100]. Оптимизатор java кода, с возможностью обфускации. Набор возможностей данного обфускатора является стандартным: удаление отладочной информации, переименование имен классов, полей, методов и т.д., удаление не используемого исходного кода и байт-кода. В качестве ядра обфускатора используется библиотека для редактирования байт-кода class-файлов BCEL [74]. В данный момент является устаревшим и не поддерживает работу с новыми выпусками JDK.

JBCO: the Java ByteCode Obfuscator [59, 104]. Обфускатор разработанный на базе платформы Soot [125, 133]. Возможности обфускации исходного кода:

- Упаковка переменных. Объединение объявления нескольких переменных в одну.
- Замена инструкций if(non)null блоками try-catch
- Вставка в код дополнительных функций таких как оператор switch и косвенных операторов условных переходов (if)
- Переупорядочивание инструкций load
- Переименование полей, методов, классов, пакетов

- Несоблюдение соглашений конструктора классов (нарушение правил работы с кодом Java).

Невозможность обфускации class-файлов, скомпилированных с помощью JDK 7 и выше. На более ранних версиях существует вероятность искажения исходного java-кода class-файла, что приводит к его неработоспособности. Необходима ручная выборка элементов программы, которые необходимо обфуцировать, что делает практически невозможной атаку обфускацией на class-файлы всей информационной системы [96].

GuardIT [146]. Это проприетарный обфускатор от компании Arxan, который не только выполняет обфускацию кода, обфускацию имен и шифрование строк, но также выполняет накладные расходы и в других местах, чтобы поместить исходный код java-приложения в систему активной защиты, способную обнаруживать несанкционированное вмешательство в режиме реального времени [112]. Эти методы работают с байт-кодом, что делает Java несовместимой с компиляторами AOT.

Сложные преобразования обфускации кода перемещают, разбивают, вставляют и создают функционально эквивалентные последовательности кода, которые препятствуют декомпиляции и обратной разработке самого базового байт-кода.

Zelix KlassMaster [171]. Обфускатор для защиты байт-кода Java от декомпиляции и реверс инжиниринга. Используются технологии обфускации потока, шифрования строк, шифрования целочисленных констант, обфускация параметров метода, обфускация ссылок и изменения параметров метода.

Большим недостатком данного инструмента для обфускации кода является геометрическое увеличение сложности исходного кода, увеличение нагрузки и большие потери в производительности в обфуцированном java-приложении [131, 132]. Такие же недостатки, но в более меньших величинах имеются у обфускатора **DashO-Pro** [144].

Allatori [53]. Представитель обфускаторов реализующих строковое шифрование в class-файлах. Позволяет произвести обфускацию имен, потоков, отладочной информации. Удаляет трассировку стека.

Строковое шифрование никаким образом не влияет на функционал class-файлов информационной системы, а, следовательно, не разрушает цифровой водяной знак. Обфускация потока управления и удаление трассировки стека наносят больше вреда, чем пользы, в связи с тем, что увеличивают нагрузку на JIT компилятор виртуальной машины Java тем самым значительно увеличивая время обработки команд и запуска class-файлов. Удаление трассировки стека не позволяет в будущем произвести разбор работы программы, как для злоумышленника, так и для системных администраторов, производящих обслуживание информационной системы.

JObfuscator [109]. JObfuscator переименовывает все переменные и методы, преобразовывает поток кода в нелинейную версию и шифрует все строки с помощью механизма шифрования полиморфных строк [51]. Крайне нестабилен для class-файлов, скомпилированных на новейших версиях Java Development Kit.

Stringer Java Obfuscator [164]. Исходя из названия, данный обфускатор специализируется исключительно на шифровании строк в режиме реального времени работы информационной системы. Производит шифрование строк и ресурсов java-приложения, включая нативные библиотеки. Сокрытие вызовов методов, типов полей и доступа к полям. Защита от отладки Java-кода (удаление стектрейса логирования). Скрытие вызовов методов позволяет защитить критические места приложения от анализа и модификации. Этот механизм маскирует вызовы библиотечных методов и методов приложения специально построенными динамическими функциями. Он также скрывает типы полей и доступ к полям.

Разработчики обфускатора придерживаются мнения, что самая критическая, чувствительная и конфиденциальная информация современных java-приложений и информационных систем находится в строковом представлении, в переменных типа String (строка). Исходя из этого, данный обфускатор не вмешивается

напрямую в логику работы программы и не меняет поток управления, что гарантирует целостность любого цифрового водяного знака находящегося в байт-коде class-файла информационной системы, подвергнутого атакой обфускации с помощью данного инструмента. Таким образом, данный инструмент выгодно использовать совместно с методикой вложения цифрового водяного знака.

Результаты анализа обфускаторов представлены в таблице 4.1. Результаты, продемонстрированные в таблице, дают наглядное представление о функциональных возможностях, рассмотренных обфускаторов.

Устаревшие обфускаторы исключаются из анализа таблицы. Обфускаторы, которые имеют стандартный функционал, не влияющий на внутреннюю логику приложения и не изменяющие коренным образом большое количество байт-кода, не способны повредить или уничтожить цифровой водяной знак. Обфускаторы, обладающие функциями продвинутой обфускации при использовании специалистами имеют возможность повредить и в некоторых случаях уничтожить цифровой водяной знак. Однако это не гарантирует достижение целей злоумышленника – доступ к исходному коду, его редактирование и использование в своих целях.

В связи с тем, что функции продвинутой обфускации кардинальным образом меняют граф потока управления приложения, то компилятор Java будет затрачивать большее количество времени и ресурсов на запуск и поддержание работы java-приложения или информационной системы. Также на текущий момент, после использования методов продвинутой обфускации, при декомпиляции class-файлов в подавляющем большинстве случаев результат является ложноположительным, иными словами, продвинутая обфускация имеет большую вероятность привести class-файлы java-приложения или информационной системы в неработоспособный вид.

При детальном анализе, исключающем устаревшие версии обфускаторов, возможно сформировать вывод о том, что на текущий момент находящиеся в открытом доступе обфускаторы, не могут в должной мере использованы в качестве инструмента для проведения атак на class-файлы информационной системы с

вложенным цифровым водяным знаком. Даже в малом проценте случаев, когда злоумышленнику удастся удалить цифровой водяной знак, с большой вероятностью будет получен неработоспособный, тяжеловесный и избыточный код, который необходимо восстанавливать до работоспособного состояния, затрачивая временные усилия в разы, превосходящие написание собственного кода специалистом. При этом, после применения методик обфускации, злоумышленник не может наверняка утверждать, что его действия привели к уничтожению цифрового водяного знака.

Таблица 4.1 – Результаты анализа обфускаторов

Наименование обфускатора / Функционал	Переименование идентификаторов	Удаление отладочной информации	Шифрование строк	Изменение потока управления	Вставка поврежденного/ недостижимого кода	Удаление неиспользуемого кода, оптимизация	Гибкость управления обфускацией	Реконструкция трассировки стека	Работа с Reflection API	Потеря производительности	Устарел	Платный
ProGuard	+	+	–	–	–	+ –	Высокая	+	–	–	–	–
yGuard	+	–	–	–	+	+	Высокая	–	–	–	–	–
JODE	+	+	–	–	–	+	Низкая	–	–	–	+	–
JavaGuard	+	–	–	–	–	+	Низкая	–	–	–	+	–
jarg	+	+	–	–	–	+	Низкая	–	–	–	+	–
JBCO: the Java ByteCode Obfuscator	+	+	–	+	+	+	Низкая	–	+	+	+ –	–
GuardIT	+	+	+	–	–	+	Высокая	–	–	+ –	–	++
Zelix KlassMaster	+	+	+	+	+	+	Высокая	+	–	+++	–	+
DashO-Pro	+	+	–	–	–	+	–	–	–	++	–	+++
Allatori	+	+	+	+	–	+	Низкая	–	–	++	–	+
JObfuscator	+	–	+	+	+	+ –	Низкая	–	–	+	+ –	–
Stringer Java Obfuscator	+	–	+	–	–	+	Высокая	–	–	– +	–	+++

4.3.2 Влияние обфускации на class-файлы java-приложения

В разделе 4.2.1 были разобраны наиболее эффективные и часто используемые обфускаторы, которые могли иметь успех в атаках на цифровой водяной знак. Как продемонстрировано в анализе, практически все приведенные обфускаторы имеют схожий набор функций, которые позволяют изменять байт-код class-файлов тем или иным образом. Однако абсолютно все проанализированные обфускаторы имеют ряд недостатков, которые экспоненциально влияют на целостность и работоспособность программы по мере усиления эффекта обфускации исходного кода. Таким образом, возможно сделать вывод, что базовые функции обфускации любого инструмента не смогут полноценно произвести уничтожение цифрового водяного знака.

Функции более сложного характера в результате совокупности воздействий на байт-код могут нанести существенный вред, искажающий цифровой водяной знак, однако, приведут к необратимым изменениям в исходном коде, после чего он будет недоступен для использования или модификации [39]. Функции продвинутой обфускации, такие как изменение потока управления, помимо возможного искажения логики работы информационной системы, производят ряд нежелательных изменений исходного кода, такие как:

- Невозможность поддержки информационной системы и ее исходного кода
- Невозможность обработки ошибок работы информационной системы из-за удаления отладочной информации
- Нагрузка на виртуальную машину Java
- Увеличение времени старта и загрузки исходного кода
- Увеличение времени выполнения кода.

По умолчанию компилятор Java записывает имена исходных файлов и, при необходимости, информацию о номерах строк в результирующие class-файлы. Эти данные необходимы для получения трассировок стека. Обфускатор может полностью удалить эту информацию или изменить имена файлов на

бессмысленные строки. Для правильной работы многих сторонних библиотек и фреймворков требуется информация о трассировке стека. Одним из примеров является Apache log4j. Таким образом, удаление трассировки стека моментально приводит в неработоспособное состояние часть внутренней логики java-приложения или информационной системы.

Многие обфускаторы предоставляют механизм, позволяющий разработчикам управлять обфускацией определенных областей программы, используя регулярные выражения для указания определенных классов, полей или методов. Это необходимо в связи с тем, что производить обфускацию всего java-приложения или информационной системы не только тратит большое количество ресурсов, но и опасно из-за риска снижения производительности или приведения к неработоспособности. При использовании функций продвинутой обфускации удаление цифрового водяного знака, созданного и вложенного по разработанной методике, не гарантируется, но гарантируется значительное снижение скорости выполнения исходного кода. Результаты демонстрируют, что среднее замедление скорости выполнения исходного кода достигает 35%, а увеличение размера одного class-файла на 300% [151].

Более подробный анализ конкретных случаев показал, что некоторое снижение производительности было связано с увеличением времени, необходимого JIT-компиляторам для анализа сложного потока управления, созданного обфускациями. Следовательно, обфускации создают проблемы для таких инструментов, как компиляторы и декомпиляторы. Причина воздерживаться от изменения потока управления кода с помощью обфускации заключается в том, что некоторые из этих изменений не позволяют виртуальной машине Java эффективно оптимизировать код, что ухудшит производительность java-приложения или class-файлов информационной системы [58].

Также, обфускация кода и потока управления данными java-приложения или информационной системы имеет ряд недостатков:

- Наименования стандартных классов Java API, которые входят в состав JRE, невозможно скрыть с помощью обфускации, поэтому все использования этих class-файлов остаются ясно видимыми в коде, полученном после декомпиляции;
- При получении доступа к объектам во время выполнения исходного кода через Reflection API или JNI не представляется возможным их переименовать. В связи с тем, что исходный код исполняется в реальном времени динамически, невозможно определить осуществимость доступа к классам, компонентам или методам;
- Имена сериализуемых классов и классов, связанных с технологией RMI (такие как классы с суффиксами _Stub и _Skel, а также классы, расширяющие java.rmi.Remote), обычно исключаются из процесса обфускации при использовании большинства обфускаторов. Имена сериализуемых классов часто используются внутри этих процессов, и обфускация их имен может нарушить правильное восстановление объектов. Аналогично, в технологии RMI классы с суффиксами _Stub и _Skel используются для генерации кода, необходимого для удаленного вызова методов. Имя класса _Stub соответствует удаленному объекту, а _Skel соответствует серверной стороне. Если эти имена будут обфусцированы, то вызовы методов RMI могут перестать работать должным образом. Обфускаторы, предназначенные для Java, обычно учитывают особенности сериализации и RMI, и автоматически пропускают эти классы через механизм исключения, чтобы сохранить правильную работу сериализации объектов и RMI;
- Перегруженные классы могут не пройти более строгую проверку байт-кода верификатором виртуальной машины Java;
- Обфускация потока управления отрицательно влияет на производительность;
- Набор инструкций JVM состоит из инструкций более высокого уровня по сравнению с реальными процессорами, такими как x86 или ARM, поэтому дизассемблированный код Java легче проанализировать, разобрать и понять, чем дизассемблированный код C++, что сводит к минимуму всю эфективность обфускации, как инструмента.

Обычно обфускация используется для защиты кода от обратной инжиниринга и попыток несанкционированного доступа. В обфускаторах могут применяться различные техники, включая замену инструкций на менее понятные или нестандартные формы. Однако, несмотря на это, исходный код, обфусцированный с помощью обфускатора, по-прежнему остается допустимым и исполняемым Java-кодом. Процесс декомпиляции обфусцированного кода может привести к созданию псевдокода с большим количеством меток и операторов goto, что может привести к неработоспособности кода, замедлению иего исполнения или нарушению логики. Таким образом, атака обфускацией на цифровой водяной знак в class-файлах java-приложения или информационной системы, не гарантирует его удаление, но с большой вероятностью приведет к критическим искажениям логики или ее полной неработоспособности [111].

Существуют исследования, подтверждающие ложноположительную обфускацию, что создает только видимость, но не реальный результат. Таким образом, не все инструменты обфускации способны произвести ее с реальным работающим кодом. Подобные исследования дополняют информацию о том, на сколько много необходимо ресурсов для проведения атаки обфускацией без гарантии удаления цифрового водяного знака [169].

4.4. Статистический анализ

В предыдущих блоках исследовались возможности различных инструментов обфускации, их влияние на производительность class-файлов java-приложений в информационной системе, вероятность повреждения или удаления цифрового водяного знака, а также недостатки продвинутой обфускации при необходимости получения исходного кода.

Исходя из анализа обфускаторов представленного в блоке 4.3, возможно сделать вывод о том, что цифровой водяной знак созданный и вложенный в class-файлы информационной системы, устойчив к атакам обфускаторов, ключевые функции которых основаны на алгоритмах оптимизации исходного кода.

Продвинутые обфускаторы, ключевые функции которых основаны на алгоритмах изменения графа потока управления приложением, позволяют с большей вероятностью повредить или уничтожить цифровой водяной знак. Однако вероятность полноценного уничтожения цифрового водяного знака посредством атак продвинутой обфускацией прямо пропорциональна вероятности приведения исходного кода в неработоспособное состояние или состояние критических изменений, которые в результате всех действий обфускатора не являются исходным кодом и имеют колоссальные потери производительности в силу значительно переусложненной логики. Рисунок 4.6 демонстрирует данную зависимость.

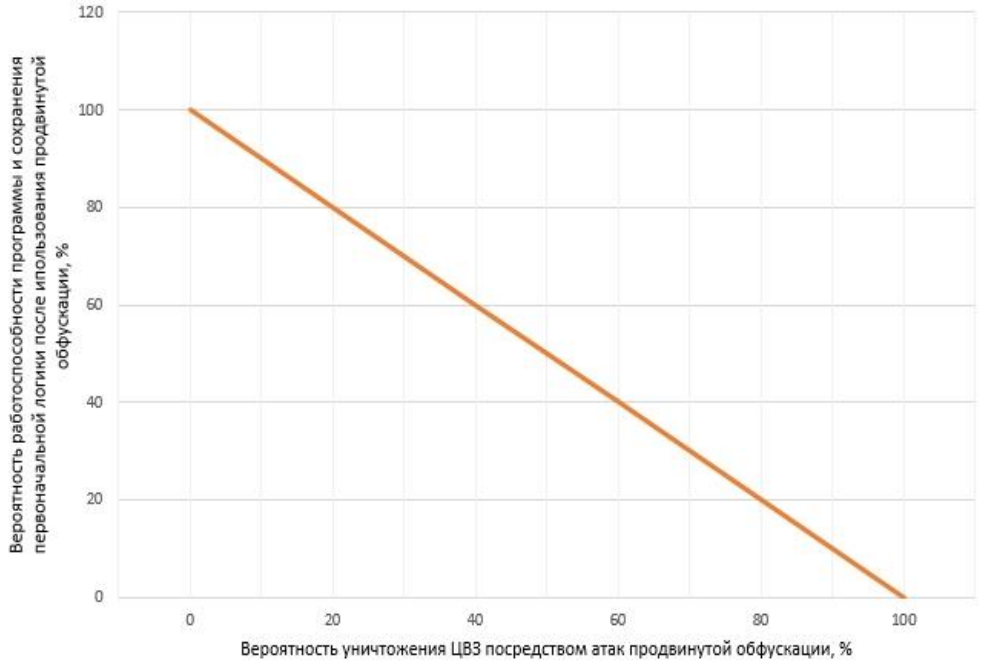


Рисунок 4.6 – Взаимосвязь вероятности работоспособности программы и уничтожения ЦВЗ посредством атак продвинутой обфускацией

Таким образом, исходя из конечной цели злоумышленника, где повреждение или удаление цифрового водяного знака, является одним из промежуточных действий, возможно сделать заключение о том, что добиться одновременного удаления или повреждения ЦВЗ и получения чистого исходного, работоспособного кода не представляется возможным. Действия, где повреждение или удаление цифрового водяного знака является конечной целью, без дальнейшей возможности

редактирования полученного исходного кода программы, а также с высоким риском неработоспособности программы после поведенных действий по обфускации, не рассматриваются в силу отсутствия выгоды злоумышленника и невозможности ее получения из неработоспособного кода.

В различных проектах, информационных системах, java-приложениях используются разные подходы по определению допустимого объема class-файла, деления функционала на блоки и классы. Для того, чтобы получить среднюю оценку возможностей методики по объемам вложения цифрового водяного знака в class-файлы, необходимо провести анализ class-файлов по заранее описанному критерию. Так как требования могут отличаться даже в пределах разных отделов одной компании, было принято решение использовать в качестве маркерных и эталонных class-файлы составляющих ядро и основу логики языка программирования Java.

Таким образом, был произведен анализ 320 class-файлов из 26 пакетов, отвечающих за различный функционал языка программирования Java. Анализ производился по 4 параметрам:

- Количество строк кода в class-файле (без комментариев и документации, но с сохранением шаблонов проектирования кода)
- Количество конструкций if-else
- Количество конструкций while
- Количество конструкций for

На основе полученной информации высчитывалось отношение количества опкодов, доступных к эквивалентным заменам, к общему количеству строк кода в class-файле. Подобный анализ уже проводился в упрощенном виде в блоке 2.7.

Среди проанализированных class-файлов присутствуют классы-фабрики, функционально обоснованные действием создания нового экземпляра какого-либо класса и не имеющие в себе больше никакой логики. Также, в выборке присутствуют class-файлы необходимые для генерации различного рода ошибок,

аварийно-завершающих выполнение java-приложения, не содержащие никакой другой логики. Результаты вычислений представлен в таблице 4.2.

Таблица 4.2 – Процентное соотношение возможного объема вложения в class-файлы информационной системы

Пакет	Наименование класса	Кол-во строк кода	Кол-во if-else	Кол-во while	Кол-во for	Итого возможных замен	Отношение минимально возможных замен к LOC (%)
java.compiler.javax.annotation.processing	AbstractProcessor	96	8	0	1	9	9,375 %
java.compiler.javax.annotation.processing	Completions	40	1	0	0	1	2,5 %
java.base.javax.crypto.spec	ChaCha20ParameterSpec	29	1	0	0	1	3,448 %
java.base.javax.crypto.spec	DESedeKeySpec	39	2	0	0	2	5,128 %
java.base.javax.crypto.spec	DESKeySpec	73	8	0	3	11	15,068 %
java.base.javax.crypto.spec	DHGenParameterSpec	16	0	0	0	0	0 %
java.base.javax.crypto.spec	DHParameterSpec	29	0	0	0	0	0 %
java.base.javax.crypto.spec	DHPrivateKeySpec	33	0	0	0	0	0 %
java.base.javax.crypto.spec	DHPublicKeySpec	33	0	0	0	0	0 %
java.base.javax.crypto.spec	GCMParameterSpec	37	3	0	0	3	8,108 %
java.base.javax.crypto.spec	IvParameterSpec	25	4	0	0	4	16 %
java.base.javax.crypto.spec	OAEPParameterSpec	56	6	0	0	6	10,714 %
java.base.javax.crypto.spec	PBEKeySpec	98	12	0	0	12	12,245 %
java.base.javax.crypto.spec	PBEParameterSpec	36	0	0	0	0	0 %
java.base.javax.crypto.spec	Psource	37	0	0	0	0	0 %
java.base.javax.crypto.spec	RC2ParameterSpec	69	5	0	0	5	7,246 %
java.base.javax.crypto.spec	RC5ParameterSpec	87	6	0	1	7	8,046 %
java.base.javax.crypto.spec	SecretKeySpec	88	10	0	2	12	13,636 %
java.base.javax.crypto	Cipher	1068	129	20	0	149	13,951 %
java.base.javax.crypto	CipherInputStream	160	15	0	2	17	10,625 %
java.base.javax.crypto	CipherOutputStream	81	4	0	0	4	4,938 %
java.base.javax.crypto	CipherSpi	187	14	2	0	16	8,556 %
java.base.javax.crypto	CryptoAllPermission	32	0	0	0	0	0 %
java.base.javax.crypto	CryptoAllPermissionCollection	41	3	0	0	3	7,317 %
java.base.javax.crypto	CryptoPermission	211	26	0	0	26	12,322 %
java.base.javax.crypto	CryptoPermissionCollection	52	4	1	0	5	9,615 %
java.base.javax.crypto	CryptoPermissions	233	21	8	3	32	13,734 %

java.base.javax.crypto	CryptoPolicyParser	495	58	4	3	65	13,131 %
java.base.javax.crypto	EncryptedPrivateKeyInfo	202	19	0	1	20	9,901 %
java.base.javax.crypto	ExemptionMechanism	131	6	0	0	6	4,58 %
java.base.javax.crypto	ExemptionMechanismException	19	0	0	0	0	0 %
java.base.javax.crypto	ExemptionMechanismSpi	29	0	0	0	0	0 %
java.base.javax.crypto	extObjectInputStream	34	2	0	0	2	5,882 %
java.base.javax.crypto	IllegalBlockSizeException	19	0	0	0	0	0 %
java.base.javax.crypto	JarVerifier	487	30	6	12	48	9,856 %
java.base.javax.crypto	JceSecurity	292	20	3	0	23	7,877 %
java.base.javax.crypto	JceSecurityManager	184	20	2	1	23	12,5 %
java.base.javax.crypto	KeyAgreement	272	22	6	0	28	10,294 %
java.base.javax.crypto	KeyAgreementSpi	30	0	0	0	0	0 %
java.base.javax.crypto	KeyGenerator	275	26	6	0	32	11,636 %
java.base.javax.crypto	KeyGeneratorSpi	23	0	0	0	0	0 %
java.base.javax.crypto	Mac	338	33	6	0	39	11,538 %
java.base.javax.crypto	MacSpi	60	3	0	1	4	6,667 %
java.base.javax.crypto	NoSuchPaddingException	19	0	0	0	0	0 %
java.base.javax.crypto	NullCipher	12	0	0	0	0	0 %
java.base.javax.crypto	NullCipherSpi	79	2	0	0	2	2,532 %
java.base.javax.crypto	PermissionsEnumerator	56	5	1	0	6	10,714 %
java.base.javax.crypto	ProviderVerifier	142	11	0	0	11	7,746 %
java.base.javax.crypto	SealedObject	198	13	0	0	13	6,566 %
java.base.javax.crypto	SecretKeyFactory	188	17	0	0	17	9,043 %
java.base.javax.crypto	SecretKeyFactorySpi	21	0	0	0	0	0 %
java.base.javax.crypto	ShortBufferException	19	0	0	0	0	0 %
java.base.javax.net.ssl	CertPathTrustManagerParameters	20	0	0	0	0	0 %
java.base.javax.net.ssl	DefaultSSLServerSocketFactory	47	0	0	0	0	0 %
java.base.javax.net.ssl	DefaultSSLSocketFactory	52	0	0	0	0	0 %
java.base.javax.net.ssl	ExtendedSSLSession	25	0	0	0	0	0 %
java.base.javax.net.ssl	HandshakeCompletedEvent	76	1	0	0	1	1,316 %
java.base.javax.net.ssl	HttpsURLConnection	117	8	0	0	8	6,838 %
java.base.javax.net.ssl	KeyManagerFactory	83	1	0	0	1	1,205 %
java.base.javax.net.ssl	KeyManagerFactorySpi	23	0	0	0	0	0 %

java.base.javax.net.ssl	KeyStoreBuilderParameters	32	1	0	0	1	3,125 %
java.base.javax.net.ssl	SNIHostName	110	7	0	0	7	6,364 %
java.base.javax.net.ssl	SNIMatcher	26	2	0	0	2	7,692 %
java.base.javax.net.ssl	SNIServerName	85	7	0	1	8	9,412 %
java.base.javax.net.ssl	SSLContext	123	3	0	0	3	2,439 %
java.base.javax.net.ssl	SSLContextSpi	51	0	0	0	0	0 %
java.base.javax.net.ssl	SSLEngine	158	8	0	0	8	5,063 %
java.base.javax.net.ssl	SSLEngineResult	76	1	0	0	1	1,316 %
java.base.javax.net.ssl	SSLException	26	0	0	0	0	0 %
java.base.javax.net.ssl	SSLHandshakeException	14	0	0	0	0	0 %
java.base.javax.net.ssl	SSLKeyException	14	0	0	0	0	0 %
java.base.javax.net.ssl	SSLParameters	206	11	2	1	14	6,796 %
java.base.javax.net.ssl	SSLPeerUnverifiedException	14	0	0	0	0	0 %
java.base.javax.net.ssl	SSLPermission	20	0	0	0	0	0 %
java.base.javax.net.ssl	SSLProtocolException	14	0	0	0	0	0 %
java.base.javax.net.ssl	SSLServerSocket	89	6	0	0	6	6,742 %
java.base.javax.net.ssl	SSLServerSocketFactory	71	5	0	0	5	7,042 %
java.base.javax.net.ssl	SSLSessionBindingEvent	26	0	0	0	0	0 %
java.base.javax.net.ssl	SSLSocket	124	6	0	0	6	4,839 %
java.base.javax.net.ssl	SSLSocketFactory	107	7	0	0	7	6,542 %
java.base.javax.net.ssl	StandardConstants	14	0	0	0	0	0 %
java.base.javax.net.ssl	TrustManagerFactory	79	1	0	0	1	1,266 %
java.base.javax.net.ssl	TrustManagerFactorySpi	21	0	0	0	0	0 %
java.base.javax.net.ssl	X509ExtendedKeyManager	21	0	0	0	0	0 %
java.base.javax.net.ssl	X509ExtendedTrustManager	22	0	0	0	0	0 %
java.base.javax.net	ServerSocketFactory	39	1	0	0	1	2,564 %
java.base.javax.net	SocketFactory	45	1	0	0	1	2,222 %
java.base.javax.security.auth.callback	ChoiceCallback	91	3	0	1	4	4,396 %
java.base.javax.security.auth.callback	ConfirmationCallback	152	12	0	2	14	9,211 %
java.base.javax.security.auth.callback	NameCallback	40	2	0	0	2	5 %
java.base.javax.security.auth.callback	PasswordCallback	100	4	0	0	4	4 %
java.base.javax.security.auth.callback	TextInputCallback	100	2	0	0	2	2 %
java.base.javax.security.auth.callback	TextOutputCallback	50	1	0	0	1	2 %

java.base.javax.security.auth.callback	UnsupportedCallbackException	30	0	0	0	0	0 %
java.base.javax.security.auth.login	AppConfigurationEntry	45	1	0	0	1	2,222 %
java.base.javax.security.auth.login	Configuration	235	9	0	0	9	3,83 %
java.base.javax.security.auth.login	ConfigurationSpi	20	0	0	0	0	0 %
java.base.javax.security.auth.login	LoginContext	448	46	0	4	50	11,161 %
java.base.javax.security.auth.x500	X500Principal	150	15	0	0	15	10 %
java.base.javax.security.auth.x500	X500PrivateCredential	47	2	0	0	2	4,255 %
java.base.javax.security.auth	PrivateCredentialPermission	329	33	3	15	51	15,502 %
java.base.javax.security.auth	Subject	896	46	16	3	65	7,254 %
java.base.javax.security.auth	SubjectDomainCombiner	236	28	0	7	35	14,831 %
java.base.com.sun.crypto.provider	AESCipher	273	10	0	0	10	3,663 %
java.base.com.sun.crypto.provider	AESCrypt	480	16	1	36	53	11,042 %
java.base.com.sun.crypto.provider	AESKeyGenerator	51	2	0	0	2	3,922 %
java.base.com.sun.crypto.provider	AESParameters	50	1	0	0	1	2 %
java.base.com.sun.crypto.provider	AESWrapCipher	245	16	0	8	24	9,796 %
java.base.com.sun.crypto.provider	ARCFOURCipher	187	12	1	0	13	6,952 %
java.base.com.sun.crypto.provider	BlockCipherParamsCore	83	6	0	0	6	7,229 %
java.base.com.sun.crypto.provider	BlowfishCipher	91	0	0	0	0	0 %
java.base.com.sun.crypto.provider	BlowfishCrypt	176	4	0	7	11	6,25 %
java.base.com.sun.crypto.provider	BlowfishKeyGenerator	49	2	0	0	2	4,082 %
java.base.com.sun.crypto.provider	BlowfishParameters	50	1	0	0	1	2 %
java.base.com.sun.crypto.provider	ChaCha20Cipher	746	44	2	0	46	6,166 %
java.base.com.sun.crypto.provider	ChaCha20Poly1305Parameters	80	6	0	0	6	7,5 %
java.base.com.sun.crypto.provider	CipherBlockChaining	105	4	0	4	8	7,619 %
java.base.com.sun.crypto.provider	CipherCore	704	99	0	0	99	14,063 %
java.base.com.sun.crypto.provider	CipherFeedback	136	7	0	6	13	9,559 %
java.base.com.sun.crypto.provider	CipherTextStealing	113	8	0	4	12	10,619 %
java.base.com.sun.crypto.provider	CipherWithWrappingSpi	116	1	0	0	1	0,862 %
java.base.com.sun.crypto.provider	ConstructKeys	98	0	0	0	0	0 %
java.base.com.sun.crypto.provider	CounterMode	99	4	0	2	6	6,061 %
java.base.com.sun.crypto.provider	DESCipher	96	1	0	0	1	1,042 %
java.base.com.sun.crypto.provider	DESCrypt	1153	59	0	1	60	5,204 %
java.base.com.sun.crypto.provider	DESedeCipher	96	1	0	0	1	1,042 %

java.base.com.sun.crypto.provider	DESedeCrypt	80	4	0	1	5	6,25 %
java.base.com.sun.crypto.provider	DESedeKey	94	4	0	1	5	5,319 %
java.base.com.sun.crypto.provider	DESedeKeyFactory	66	6	0	0	6	9,091 %
java.base.com.sun.crypto.provider	DESedeKeyGenerator	73	3	0	0	3	4,11 %
java.base.com.sun.crypto.provider	DESedeParameters	50	1	0	0	1	2 %
java.base.com.sun.crypto.provider	DESedeWrapCipher	282	15	0	3	18	6,383 %
java.base.com.sun.crypto.provider	DESKey	92	4	0	0	4	4,348 %
java.base.com.sun.crypto.provider	DESKeyFactory	66	6	0	0	6	9,091 %
java.base.com.sun.crypto.provider	DESKeyGenerator	70	3	1	1	5	7,143 %
java.base.com.sun.crypto.provider	DESParameters	50	1	0	0	1	2 %
java.base.com.sun.crypto.provider	DHKeyAgreement	209	27	0	1	28	13,397 %
java.base.com.sun.crypto.provider	DHKeyFactory	105	14	0	0	14	13,333 %
java.base.com.sun.crypto.provider	DHKeyPairGenerator	107	8	2	0	10	9,346 %
java.base.com.sun.crypto.provider	DHParameterGenerator	94	5	0	0	5	5,319 %
java.base.com.sun.crypto.provider	DHParameters	99	7	0	0	7	7,071 %
java.base.com.sun.crypto.provider	DHPrivateKey	188	13	0	0	13	6,915 %
java.base.com.sun.crypto.provider	DHPublicKey	190	14	0	0	14	7,368 %
java.base.com.sun.crypto.provider	ElectronicCodeBook	73	1	0	2	3	4,11 %
java.base.com.sun.crypto.provider	EncryptedPrivateKeyInfo	65	4	0	0	4	6,154 %
java.base.com.sun.crypto.provider	FeedbackCipher	65	0	0	0	0	0 %
java.base.com.sun.crypto.provider	GaloisCounterMode	416	39	0	7	46	11,058 %
java.base.com.sun.crypto.provider	GCMParameters	91	6	0	0	6	6,593 %
java.base.com.sun.crypto.provider	GCTR	93	8	0	3	11	11,828 %
java.base.com.sun.crypto.provider	GHASH	157	8	0	5	13	8,28 %
java.base.com.sun.crypto.provider	HmacCore	195	13	0	2	15	7,692 %
java.base.com.sun.crypto.provider	HmacMD5	14	0	0	0	0	0 %
java.base.com.sun.crypto.provider	HmacMD5KeyGenerator	44	1	0	0	1	2,273 %
java.base.com.sun.crypto.provider	HmacPKCS12PBECore	143	12	0	1	13	9,091 %
java.base.com.sun.crypto.provider	HmacSHA1	14	0	0	0	0	0 %
java.base.com.sun.crypto.provider	HmacSHA1KeyGenerator	44	1	0	0	1	2,273 %
java.base.com.sun.crypto.provider	ISO10126Padding	52	4	0	0	4	7,692 %
java.base.com.sun.crypto.provider	JceKeyStore	567	45	5	4	54	9,524 %
java.base.com.sun.crypto.provider	KeyGeneratorCore	184	5	0	0	5	2,717 %

java.base.com.sun.crypto.provider	KeyProtector	271	15	0	0	15	5,535 %
java.base.com.sun.crypto.provider	OAEPParameters	206	20	0	1	21	10,194 %
java.base.com.sun.crypto.provider	OutputFeedback	108	5	0	3	8	7,407 %
java.base.com.sun.crypto.provider	PBEKey	117	8	0	3	11	9,402 %
java.base.com.sun.crypto.provider	PBEKeyFactory	194	5	0	1	6	3,093 %
java.base.com.sun.crypto.provider	PBEParameters	88	4	0	0	4	4,545 %
java.base.com.sun.crypto.provider	PBES1Core	261	17	0	5	22	8,429 %
java.base.com.sun.crypto.provider	PBES2Core	387	30	0	1	31	8,01 %
java.base.com.sun.crypto.provider	PBES2Parameters	393	33	0	0	33	8,397 %
java.base.com.sun.crypto.provider	PBEWithMD5AndDESCipher	100	2	0	0	2	2 %
java.base.com.sun.crypto.provider	PBEWithMD5AndTripleDESCipher	100	2	0	0	2	2 %
java.base.com.sun.crypto.provider	PBKDF2Core	97	6	0	0	6	6,186 %
java.base.com.sun.crypto.provider	PBKDF2HmacSHA1Factory	64	6	0	0	6	9,375 %
java.base.com.sun.crypto.provider	PBKDF2KeyImpl	249	15	0	4	19	7,631 %
java.base.com.sun.crypto.provider	PBMAC1Core	202	17	0	1	18	8,911 %
java.base.com.sun.crypto.provider	PCBC	96	2	0	6	8	8,333 %
java.base.com.sun.crypto.provider	PKCS5Padding	60	6	0	1	7	11,667 %
java.base.com.sun.crypto.provider	PKCS12PBECipherCore	746	33	1	5	39	5,228 %
java.base.com.sun.crypto.provider	Poly1305	162	9	1	1	11	6,79 %
java.base.com.sun.crypto.provider	PrivateKeyInfo	40	2	0	0	2	5 %
java.base.com.sun.crypto.provider	RC2Cipher	114	2	0	0	2	1,754 %
java.base.com.sun.crypto.provider	RC2Crypt	195	3	0	9	12	6,154 %
java.base.com.sun.crypto.provider	RC2Parameters	123	13	0	1	14	11,382 %
java.base.com.sun.crypto.provider	RSACipher	389	31	0	0	31	7,969 %
java.base.com.sun.crypto.provider	SealedObjectForKeyProtector	129	8	0	0	8	6,202 %
java.base.com.sun.crypto.provider	SslMacCore	186	8	0	0	8	4,301 %
java.base.com.sun.crypto.provider	SunJCE	262	2	0	0	2	0,763 %
java.base.com.sun.crypto.provider	SymmetricCipher	21	0	0	0	0	0 %
java.base.com.sun.crypto.provider	TlsKeyMaterialGenerator	203	15	0	1	16	7,882 %
java.base.com.sun.crypto.provider	TlsMasterSecretGenerator	150	7	0	1	8	5,333 %
java.base.com.sun.crypto.provider	TlsPrfGenerator	246	11	0	4	15	6,098 %
java.base.com.sun.crypto.provider	TlsRsaPremasterSecretGenerator	61	4	0	0	4	6,557 %
java.base.com.sun.java.util.jar.pack	AdaptiveCoding	298	23	0	8	31	10,403 %

java.base.com.sun.java.util.jar.pack	Attribute	1698	114	5	32	151	8,893 %
java.base.com.sun.java.util.jar.pack	BandStructure	2759	199	1	37	237	8,59 %
java.base.com.sun.java.util.jar.pack	ClassReader	647	40	2	10	52	8,037 %
java.base.com.sun.java.util.jar.pack	ClassWriter	317	20	0	10	30	9,464 %
java.base.com.sun.java.util.jar.pack	Code	398	34	0	12	46	11,558 %
java.base.com.sun.java.util.jar.pack	Coding	775	81	4	21	106	13,677 %
java.base.com.sun.java.util.jar.pack	CodingChooser	1489	191	5	36	232	15,581 %
java.base.com.sun.java.util.jar.pack	ConstantPool	1658	133	5	35	173	10,434 %
java.base.com.sun.java.util.jar.pack	Constants	503	0	0	0	0	0 %
java.base.com.sun.java.util.jar.pack	Driver	749	66	4	14	84	11,215 %
java.base.com.sun.java.util.jar.pack	DriverResource	136	0	0	0	0	0 %
java.base.com.sun.java.util.jar.pack	DriverResource_ja	136	0	0	0	0	0 %
java.base.com.sun.java.util.jar.pack	DriverResource_zh_CN	136	0	0	0	0	0 %
java.base.com.sun.java.util.jar.pack	FixedList	173	0	0	1	1	0,578 %
java.base.com.sun.java.util.jar.pack	Fixups	570	43	0	7	50	8,772 %
java.base.com.sun.java.util.jar.pack	Histogram	818	36	3	36	75	9,169 %
java.base.com.sun.java.util.jar.pack	Instruction	688	50	2	7	59	8,576 %
java.base.com.sun.java.util.jar.pack	NativeUnpack	332	28	4	1	33	9,94 %
java.base.com.sun.java.util.jar.pack	Package	1377	93	0	30	123	8,932 %
java.base.com.sun.java.util.jar.pack	PackageReader	2376	160	7	88	255	10,732 %
java.base.com.sun.java.util.jar.pack	PackageWriter	1743	184	3	68	255	14,63 %
java.base.com.sun.java.util.jar.pack	PackerImpl	611	58	0	7	65	10,638 %
java.base.com.sun.java.util.jar.pack	PopulationCoding	500	54	4	13	71	14,2 %
java.base.com.sun.java.util.jar.pack	PropMap	332	10	1	2	13	3,916 %
java.base.com.sun.java.util.jar.pack	TLGlobals	125	0	0	0	0	0 %
java.base.com.sun.java.util.jar.pack	UnpackerImpl	263	20	0	2	22	8,365 %
java.base.com.sun.java.util.jar.pack	Utils	320	9	0	5	14	4,375 %
java.base.com.sun.net.ssl	HttpsURLConnection	105	8	0	0	8	7,619 %
java.base.com.sun.net.ssl	KeyManagerFactory	88	3	0	0	3	3,409 %
java.base.com.sun.net.ssl	KeyManagerFactorySpiWrapper	51	3	0	1	4	7,843 %
java.base.com.sun.net.ssl	SSLContext	77	2	0	0	2	2,597 %
java.base.com.sun.net.ssl	SSLContextSpiWrapper	84	8	0	2	10	11,905 %
java.base.com.sun.net.ssl	SSLSecurity	142	14	2	1	17	11,972 %

java.base.com.sun.net.ssl	TrustManagerFactory	87	3	0	0	3	3,448 %
java.base.com.sun.net.ssl	TrustManagerFactorySpiWrapper	50	3	0	1	4	8 %
java.base.com.sun.net.ssl	X509KeyManagerJavaxWrapper	75	4	0	2	6	8 %
java.base.com.sun.net.ssl	X509TrustManagerComSunWrapper	39	0	0	0	0	0 %
java.base.com.sun.net.ssl	X509TrustManagerJavaxWrapper	34	2	0	0	2	5,882 %
java.base.com.sun.security.cert.internal.x509	X509V1CertImpl	161	0	0	0	0	0 %
java.base.com.sun.security.ntlm	Client	140	9	0	1	10	7,143 %
java.base.com.sun.security.ntlm	NTLM	350	7	0	1	8	2,286 %
java.base.com.sun.security.ntlm	Server	150	20	0	0	20	13,333 %
java.base.java.io	Bits	123	0	0	0	0	0 %
java.base.java.io	BufferedInputStream	300	28	1	1	30	10 %
java.base.java.io	BufferedOutputStream	70	5	0	0	5	7,143 %
java.base.java.io	BufferedReader	381	45	4	3	52	13,648 %
java.base.java.io	BufferedWriter	122	11	2	0	13	10,656 %
java.base.java.io	ByteArrayInputStream	96	4	0	0	4	4,167 %
java.base.java.io	ByteArrayOutputStream	103	5	0	0	5	4,854 %
java.base.java.io	CharArrayReader	113	10	0	0	10	8,85 %
java.base.java.io	CharArrayWriter	103	7	0	0	7	6,796 %
java.base.java.io	Console	291	35	2	1	38	13,058 %
java.base.java.io	DataInputStream	247	21	5	0	26	10,526 %
java.base.java.io	DataOutputStream	172	8	0	5	13	7,558 %
java.base.java.io	DeleteOnExitHook	50	1	0	1	2	4 %
java.base.java.io	EOFException	40	0	0	0	0	0 %
java.base.java.io	ExpiringCache	97	6	0	2	8	8,247 %
java.base.java.io	File	745	102	1	5	108	14,497 %
java.base.java.io	FileCleanable	48	2	0	0	2	4,167 %
java.base.java.io	FileDescriptor	184	11	0	1	12	6,522 %
java.base.java.io	FileInputStream	190	14	1	0	15	7,895 %
java.base.java.io	FileOutputStream	196	15	1	0	16	8,163 %
java.base.java.io	FilePermission	647	80	4	2	86	13,292 %
java.base.java.io	FilePermissionCollection	616	79	4	3	86	13,961 %
java.nio	Bits	234	11	3	0	14	5,983 %

java.nio	BufferMismatch	212	29	0	7	36	16,981 %
java.nio	ByteBufferAsCharBufferB	250	2	0	0	2	0,8 %
java.nio	CharBufferSpliterator	93	3	0	0	3	3,226 %
java.nio	DirectByteBuffer	1098	12	0	0	12	1,093 %
java.util.stream	DistinctOps	180	12	0	0	12	6,667 %
java.util.stream	DoublePipeline	670	13	2	0	15	2,239 %
java.util.stream	ReduceOps	800	17	0	0	17	2,125 %
java.util.stream	ReferencePipeline	739	16	5	0	21	2,842 %
java.util.stream	SortedOps	709	32	0	15	47	6,629 %
java.util.stream	Streams	888	39	2	0	41	4,617 %
java.util.stream	StreamSpliterators	1550	55	18	6	79	5,097 %
java.util.stream	WhileOps	1394	52	4	0	56	4,017 %
java.util.zip	Adler32	143	5	1	0	6	4,196 %
java.util.zip	CheckedInputStream	98	3	1	0	4	4,082 %
java.util.zip	CRC32	160	9	1	0	10	6,25 %
java.util.zip	CRC32C	290	17	1	14	32	11,034 %
java.util.zip	DeflaterInputStream	189	18	2	0	20	10,582 %
java.util.zip	DeflaterOutputStream	256	12	3	0	15	5,859 %
java.util.zip	GZIPInputStream	294	19	3	0	22	7,483 %
java.util.zip	ZipOutputStream	640	77	3	1	81	12,656 %
java.util.zip	ZipUtils	218	4	0	0	4	1,835 %
java.util	ArrayDeque	687	73	0	28	101	14,702 %
java.util	ArrayList	1160	97	1	24	122	10,517 %
java.util	ArrayPrefixHelpers	623	96	4	24	124	19,904 %
java.util	ArraysParallelSortHelpers	925	72	16	24	112	12,108 %
java.util	BitSet	764	72	11	21	104	13,613 %
java.util	ImmutableCollections	961	56	4	15	75	7,804 %
java.util	JumboEnumSet<E extends Enum<E>>	230	22	1	8	31	13,478 %
java.util	LinkedHashMap<K,V>	378	33	0	7	40	10,582 %
java.util	LinkedList<E>	722	51	3	17	71	9,834 %
java.util	PriorityQueue<E>	536	56	4	14	74	13,806 %
java.util	Spliterators	1129	83	12	0	95	8,415 %
java.util	Stack<E>	46	2	0	0	2	4,348 %

java.util	TimSort<T>	532	58	20	1	79	14,85 %
jdk.internal.event	EventHelper	150	3	0	0	3	2 %
jdk.internal.jimage.decompressor	CompressedResourceHeader	130	1	0	0	1	0,769 %
jdk.internal.jimage.decompressor	CompressIndexes	135	2	0	4	6	4,444 %
jdk.internal.jimage.decompressor	Decompressor	105	5	0	1	6	5,714 %
jdk.internal.jimage.decompressor	ResourceDecompressorRepository	78	1	0	0	1	1,282 %
jdk.internal.jimage.decompressor	SignatureParser	129	5	0	2	7	5,426 %
jdk.internal.event	StringSharingDecompressor	239	4	0	2	6	2,51 %
jdk.internal.event	ZipDecompressor	71	0	1	0	1	1,408 %
jdk.internal.jimage	BasicImageReader	450	31	0	0	31	6,889 %
jdk.internal.jimage	ImageBufferCache	135	7	0	2	9	6,667 %
jdk.internal.jimage	ImageHeader	183	1	0	0	1	0,546 %
jdk.internal.jimage	ImageLocation	332	26	1	3	30	9,036 %
jdk.internal.jimage	ImageReader	831	31	0	5	36	4,332 %
jdk.internal.jimage	ImageReaderFactory	90	0	0	0	0	0 %
jdk.internal.jimage	ImageStream	207	3	0	1	4	1,932 %
jdk.internal.jimage	ImageStringsReader	325	20	7	5	32	9,846 %
jdk.internal.jmod	JmodFile	240	6	0	0	6	2,5 %
jdk.internal.jrtfs	ExplodedImage	304	18	0	4	22	7,237 %
jdk.internal.jrtfs	JrtDirectoryStream	100	5	0	0	5	5 %
jdk.internal.jrtfs	JrtFileAttributes	141	2	0	0	2	1,418 %
jdk.internal.jrtfs	JrtFileAttributeView	198	14	0	2	16	8,081 %
jdk.internal.jrtfs	JrtFileStore	104	0	0	0	0	0 %
jdk.internal.jrtfs	JrtFileSystem	500	25	1	3	29	5,8 %
jdk.internal.jrtfs	JrtFileSystemProvider	355	21	0	0	21	5,915 %
jdk.internal.jrtfs	JrtPath	818	83	9	6	98	11,98 %
jdk.internal.jrtfs	JrtUtils	195	20	2	0	22	11,282 %
jdk.internal.jrtfs	SystemImage	130	5	0	0	5	3,846 %
jdk.internal.loader	FileURLMapper	68	2	0	0	2	2,941 %
jdk.internal.loader	BootLoader	187	10	0	0	10	5,348 %

Необходимо уточнить, что данный анализ проводился на упрощенном наборе опкодов доступных к эквивалентным заменам. Например, в представленном анализе не были учтены опкоды тернарных операторов, циклы функциональности StreamAPI, условия и предикаты фильтров в функциональности StreamAPI.

Средний размер всех class-файлов JDK представляет собой 100 мегабайт. В JDK версии 12 всего 4433 class-файлов. Таким образом, средний размер class-файла реализующего логику языка программирования Java составляет 22,1 Килобайт. Основываясь на эталонном расчете, приведенном в блоке 2.7, возможно предположить, что в среднем, на каждый байт объема class-файла гарантированный объем цифрового водяного знака, который возможно вложить в class-файл, является 32 бита. Статистические результаты, полученные из приведенной в таблице 4.2 выборки class-файлов представлены в таблице 4.3.

Таблица 4.3 – Статистические результаты возможностей методики для представленной выборки class-файлов

Рассчитываемая зависимость	**Результат**
Среднее количество возможных эквивалентных замен в class-файле	21 опкод
Среднее количество возможных эквивалентных замен в class-файле (без учета class-файлов с отсутствием опкодов возможных к эквивалентным заменам)	25 опкодов
Среднее количество возможных эквивалентных замен в class-файле у 50 class-файлов с самыми высокими значениями по количеству строк (15,6% от общего объема)	87 опкодов
Среднее количество возможных эквивалентных замен в class-файле у 50 class-файлов с самыми низкими значениями по количеству строк. Без учета class-файлов с отсутствием опкодов возможных к эквивалентным заменам (15,6% от общего объема)	3 опкода
Среднее количество возможных эквивалентных замен в class-файле у 50 class-файлов с самыми низкими значениями по количеству строк. (15,6% от общего объема)	~ 1 опкод
Среднее количество возможных эквивалентных замен в class-файле у 220 class-файлов не относящихся к верхней и	11 опкодов

нижней границе по количеству строк (68,8% от общего объема)	
Среднее количество возможных эквивалентных замен в class-файле у 168 class-файлов, не относящихся к верхней и нижней границе по количеству строк. Без учета class-файлов с отсутствием опкодов возможных к эквивалентным заменам (52,5% от общего объема)	13 опкодов

Таким образом, исходя из статистического анализа, представленного в таблице 4.3, возможен вывод о том, что class-файлы, не имеющие операционных кодов пригодных для эквивалентных замен при реализации методики необходимо пропускать, так как данные class-файлы не пригодны для вложения цифрового водяного знака. Однако данные class-файлы можно использовать для внедрения в их байт-код фиктивных методов, принимая риски, что данное решение легко обнаруживается и удаляется, но также затрачивает ресурсы злоумышленника.

Также, статистический анализ демонстрирует зависимость количества строк (объема class-файла) и количества опкодов доступных для эквивалентных замен. Однако большинство результатов демонстрирует, что стандартный class-файл в среднем содержит в себе 11-13 операционных кодов доступных к эквивалентным заменам, что, исходя из выводов блока 2.7, позволяет произвести вложение цифрового водяного знака объемом 0,4% от общего объема class-файла, при использовании минимального и безопасного набора опкодов для эквивалентных замен. При использовании расширенного набора опкодов и принятия повышенного риска о вероятности потери работоспособности class-файла, возможно добиться результатов в 4-6% от общего объема class-файла.

График отношения количества доступных к эквивалентным заменам опкодов к количеству class-файлов в созданной выборке представлен на рисунках 4.7-4.8.

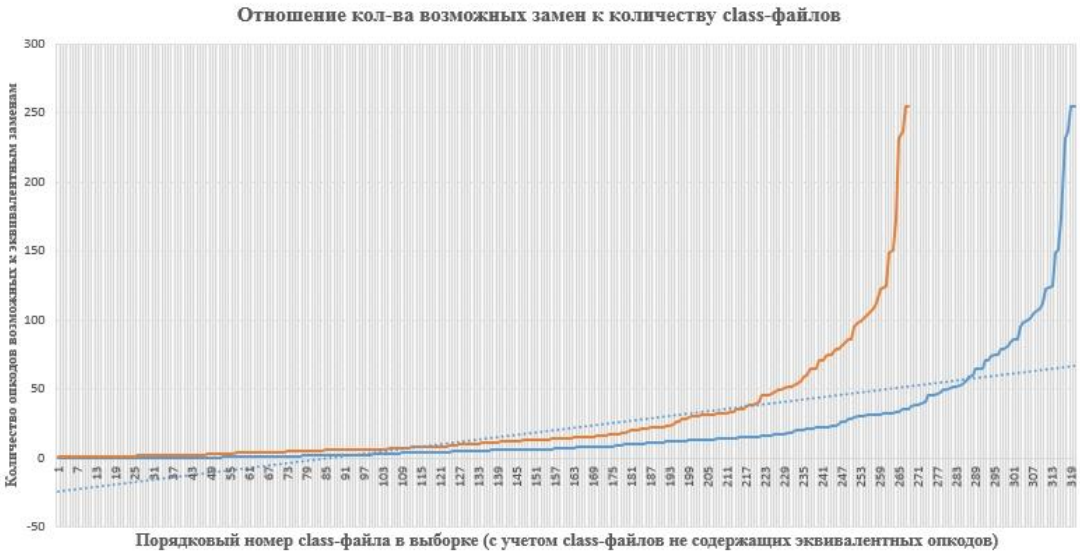


Рисунок 4.7 – Отношение количества возможных эквивалентных замен опкодов к количеству class-файлов

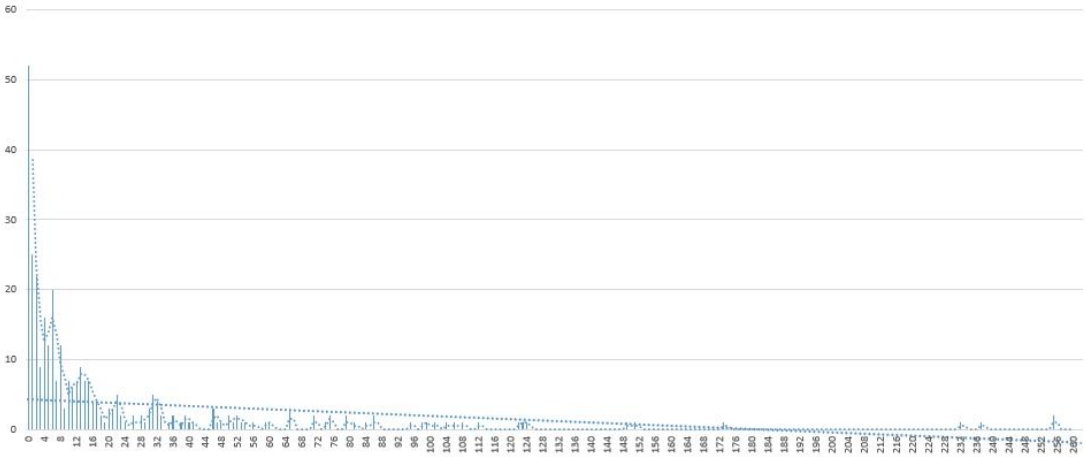


Рисунок 4.8 – Отношение количества возможных к эквивалентным заменам опкодов к количеству class-файлов с таким же кол-вом опкодов возможных к эквивалентным заменам

Рисунки 4.7-4.8 наглядно демонстрирует, что присутствующие в выборке class-файлы, у которых соотношение количества строк кода к количеству возможных эквивалентных замен опкодов выше 10%, являются скорее исключениями.

Отношение количества опкодов возможных для эквивалентных замен к количеству данных class-файлов по интервалам, с учетом class-файлов не

пригодных для вложения цифрового водяного знака посредством эквивалентных замен без создания фиктивного кода, продемонстрировано на рисунке 4.9.

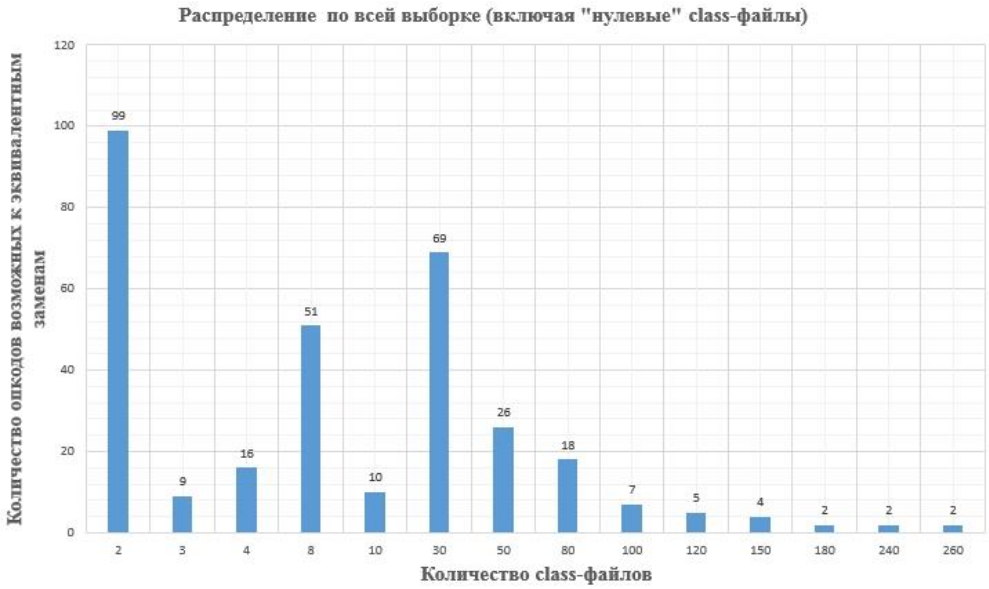


Рисунок 4.9 – Отношение количества опкодов в class-файле к количеству таких class-файлов, по заданным интервалам, с учетом не содержащих пригодных опкодов class-файлов

Отношение количества опкодов возможных для эквивалентных замен к количеству данных class-файлов по интервалам, без учета class-файлов не пригодных для вложения цифрового водяного знака посредством эквивалентных замен без создания фиктивного кода, продемонстрировано на рисунке 4.10.

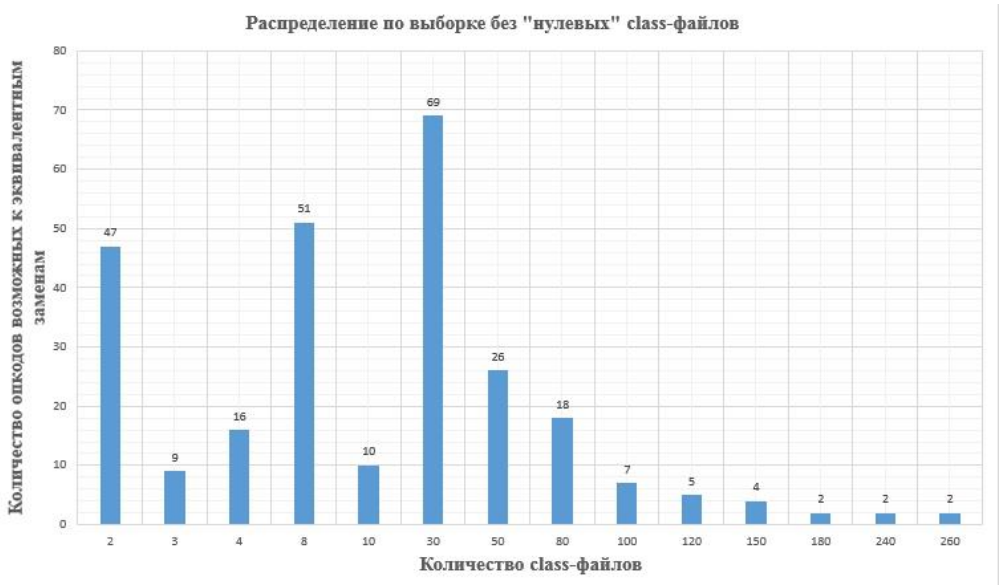


Рисунок 4.10 – Отношение количества опкодов в class-файле к количеству таких class-файлов, по заданным интервалам, без учета не содержащих пригодных опкодов class-файлов

Как видно из рисунков 4.9-4.10 в выборке присутствуют черты нормального распределения, которые при увеличении class-файлов в выборке будут приобретать более четкие очертания.

Исходя из описанного выше, возможно сделать вывод, что существует зависимость объема цифрового водяного знака к объему class-файла, в который производится вложение. Также существует зависимость количества строк кода в class-файле к количеству доступных к эквивалентным заменам операционных кодов. Что в свою очередь, предполагает зависимость количества строк исходного кода в class-файле к объему возможного к вложению цифрового водяного знака. Средний процент опкодов доступных для безопасных эквивалентных замен в class-файле равен 7,5%, при использовании минимального набора опкодов байт-кода для замен, что равняется 9-10 байтам объема цифрового водяного знака.

Таким образом, разработанная методика позволяет производить создание и вложение цифрового водяного знака на уровне java-приложений информационной системы и их class-файлов. Методика обеспечивает возможность контроля целостности информационной системы, распределение цифровых водяных знаков

равномерно по class-файлам всей информационной системы. Также вложение ЦВЗ-регистратора в ключевые class-файлы с ключевой логикой, при атаке обфускацией увеличивает вероятность приведения java-приложения или class-файла в неработоспособное состояние с невозможностью получения оригинального исходного кода, что противоречит целям злоумышленника при атаках обфускацией на информационную систему и ее части.

В данном разделе были рассмотрены 12 обфускаторов, лучших представителей инструментов обфускации, находящихся в открытом доступе, имеющих популярность и чьи алгоритмы используются в качестве основных в других менее известных обфускаторах. При подробном анализе каждого обфускатора и его возможностей, были выявлены различные недостатки, которые существенно затрудняют атаки обфускацией на цифровой водяной знак в class-файлах. Многие обфускаторы устарели, в связи с тем, что виртуальная машина Java развивается быстрыми темпами и разработчики обфускаторов не успевают добавлять новый функционал, исправлять существующий.

Например, обфускатор от компании Zelix Klassmaster – не может использоваться с целью получения чистого исходного кода, потому что сильно увеличивает сложность и нагрузку. Сильно увеличивает потери производительности [131]. Таким образом, высок риск невозможности запуска на исполнение всего java-приложения, поврежденной внутренней логике class-файлов и небольшой процент повреждения цифрового водяного знака.

Таким образом злоумышленник попадает в петлю возможностей обфускаторов. Обфускаторы, которые следуют изначальному предназначению – не делают ничего больше, чем оптимизация кода, удаление не используемых переменных, сжатие размера файлов. Таких обфускаторов большинство – около 7. Использование таких обфускаторов бессмысленно для повреждения или удаления цифрового водяного знака, созданного и вложенного посредством разработанной методики [86, 87].

Инструменты продвинутой обфускации практически гарантировано меняют граф управления потоком java-приложения, что приводит к критическому

изменению логики, уменьшению скорости работы приложения, увеличению времени старта, загрузки и вызова ключевых функций. Такие обфускаторы имеют шансы в значительной степени повредить цифровой водяной знак, однако, они требуют продолжительной настройки, указаний методов и объектов на которых могут использоваться и все равно крайне ухудшают производительность, помимо проблем потери исходного кода. Компрометация class-файлов, атака вложенного в них ЦВЗ посредством обфускации, с дальнейшим использованием полученного исходного кода – бессмысленна, по причине того, что код может быть не рабочим и гарантированно будет не исходным, а сильно усложненным [114].

Также, практически любую обфускацию можно обойти с помощью стандартной библиотеки jhsdb java [49, 54, 80, 170]. Если целью является понимание логики программы, то ни один обфускатор не способен скрыть основные принципы работы программы от высококлассного специалиста с набором инструментов. Существуют методы обратной обфускации, что узнать какие инструменты/методы применялись и определить инструмент обфускации, применяемый в качестве отправной точки для оценки сложности обратного запутывания и восстановления исходного кода [50].

Выводы

Раздел содержит основные научные результаты, полученные в работе и направленные на разрешение актуальной научной задачи исследования.

1) Рассмотрена возможная модель действий злоумышленника информационной системы. Рассмотрен вариант взаимодействия java-приложения, реализующего функцию проверки наличия ЦВЗ в структуре информационной системы и ее составных частях. Произведен анализ обфускаторов находящихся в открытом доступе и сравнение результатов их работы, функций. Результаты этих исследований опубликованы в работах [156].

2) Рассмотрена возможная структура ифнормационной системы. Исследовано воздействие обфускаторов на class-файлы информационной системы

содержащие изменения байт-кода. Рассмотрена возможность вложения дробной части цифрового водяного знака, который впоследствии может быть использован в качестве регистратора изменений в информационной системе. Исследована возможность тиражируемого вложения цифрового водяного знака в class-файлы информационной системы и одновременно части цифрового водяного знака регистратора. Результаты этих исследований опубликованы в работах [38, 43].

3) В качестве научного результата, интегрально объединяющего вышеизложенные результаты, и позволяющего обеспечить достижение цели исследования была разработана методика создания и вложения цифрового водяного знака в class-файлы информационной системы устойчивого к атакам обфускацией направленных на его разрушение, за счет создания и вложения двух видов цифровых водяных знаков в части информационной системы, использование функций негативно влияющих на обфускацию в совокупности с расширенным набором опкодов пригодных для эквивалентных замен. К элементам научной новизны данной методики относится следующее:

а) Впервые инструменты обфускации рассматриваются, как средство атаки на цифровой водяной знак с целью его разрушения, а не как инструменты оптимизации исходного кода;

б) используется связка из двух цифровых водяных знаков: на уровне java-приложений информационной системы и на уровне информационной системы. Данный подход позволяет обеспечивать наибольшую устойчивость цифровых водяных знаков к атакам и позволяет эффективнее определять скомпроментированные участки информационной системы посредством анализа целостности цифрового водяного знака регистратора и углубленного анализа, при наличии сигнала повреждения, отдельных цифровых водяных знаков в class-файлах java-приложения в котором был сигнал повреждения верхнеуровнего ЦВЗ-регистратора;

в) для вложения цифрового водяного знака в информационную систему используется комплексный анализ взаимосвязей java-приложений в

информационной системе, определение class-файлов с критически важной логикой, определение class-файлов с возможностью вложения части ЦВЗ-регистратора.

г) используются конструкции, которые некорректно обрабатываются обфускаторами находящимися в открытом доступе, что приводит к неработоспособности class-файла или java-приложения после атаки обфускацией и невозможностью получения исходного кода или его дальнейшего использования.

Данная методика и ее отдельные элементы опубликованы в работах [33, 38, 39, 43, 156].

Разработанная в главе 4 методика создания и вложения устойчивого к атакам обфускацией цифрового водяного знака регистратора в class-файлы информационной системы использует два вида цифровых водяных знаков: единый цифровой водяной знак регистратор для информационной системы и тиражируемый цифровой водяной знак в class-файлы java-приложений информационной системы. Вложение ЦВЗ-регистратора в class-файлы с ключевой логикой, при атаке обфускацией увеличивает вероятность приведения java-приложения или class-файла в неработоспособное состояние с невозможностью получения оригинального исходного кода до 70-80%.

Преимущества методики:

- Учитывает инструменты обфускации, как инструменты атаки на цифровой водяной знак, а не оптимизации кода
- Цифровой водяной знак устойчив к атакам обфускацией
- Позволяет проводить оценку целостности информационной системы и ее отдельных частей
- Использует два типа цифровых водяных знаков для информационной системы
- Позволяет произвести вложение цифровых водяных знаков таким образом, что применение инструментов продвинутой обфускации приведет к неработоспособности исходного кода
- Возможность автоматизации процесса

За счет проведенных экспериментов продемонстрировано:

- Устойчивость ЦВЗ к атакам обфускацией
- Ухудшение производительности до 300% при использовании инструментов продвинутой обфускации
- Невозможность произвести отладку информационной системы и получить ее исходный код или произвести его редактирование для последующей повторной компиляции после использования инструментов продвинутой обфускации для атаки на ЦВЗ.

Материалы главы 4 были представлены в материалах международных и российских научно-технических конференциях [33, 156] и опубликованы в изданиях, включенных в перечень ВАК России [38, 43], а также в изданиях Scopus [156]. Статья «Исследование атаки обфускацией на байт-код java-приложения с целью разрушения или повреждения цифрового водяного знака» опубликована в изданиях, включенных в перечень ВАК России, без соавторства.

Результатом работы, проведенной в главе 4, является разработанная автором программа для ЭВМ «Анализатор байт-кода java-программы для скрытого вложения цифрового водяного знака посредством автоматического редактирования байт-кода class-файла», номер регистрации 2020617872 от 15.07.2020 г., копия свидетельства о регистрации программы для ЭВМ представлена в приложении А.

Заключение и выводы по работе

Поставленная в диссертационном исследовании цель по увеличению объема цифрового водяного знака и повышению его устойчивости к атакам декомпиляцией и обфускацией в интересах защиты авторских прав на исходный код за счет расширенного набора операционных команд байт-кода для эквивалентных замен без перекомпиляции class-файла, дополнительных методов, затрудняющих декомпиляцию class-файла и java-приложения, использования результатов анализа информационной системы, модели предполагаемого злоумышленника и единого для всей системы цифрового водяного знака выполнена.

Для достижения цели были поставлены и выполнены задачи, получены научные результаты, составляющие итоги исследования:

1. Разработана методика создания и скрытого вложения цифрового водяного знака увеличенного объема в байт-код class-файла основанная на использовании дополнительных наборов операционных кодов байт-кода. Методика позволяет производить скрытое вложение цифрового водяного знака увеличенного объема в байт-код class-файлов на 60-80% по сравнению с известными.

Разработанная методика позволяет:

- Производить увеличенное в объеме вложение в исполняемые class-файлы
- Не нарушать логику работы class-файлов
- Не изменять в большую сторону объем class-файлов
- Производить вложение тиражируемого цифрового водяного знака
- Возможность автоматизировать процесс доведя до реализации программы для ЭВМ.

2. Разработана методика создания и скрытого вложения цифрового водяного знака, устойчивого к атакам декомпиляцией, в байт-код class-файлов java-приложений. Методика основана на использовании результатов анализа class-

файлов java-приложения, выявления class-файлов с ключевой логикой для использования шаблонов и методов проектирования кода, затрудняющих декомпиляцию class-файла, с последующим вложением цифрового водяного знака. Методика в отличие от известных позволяет произвести вложение цифрового водяного знака устойчивого к атакам перекомпиляцией, в class-файлы java-приложения в 80-90% случаев.

Разработанная методика позволяет:

- Учитывать декомпиляцию и повторную компиляцию class-файлов, как инструмент атаки на цифровой водяной знак
- Производить вложение функций, затрудняющих декомпиляцию, как на этапе написания исходного кода, так и во время работы с скомпилированным class-файлом
- Использовать расширенный набор опкодов для эквивалентных замен в байт-коде class-файлов
- Возможность автоматизировать процесс доведя до реализации программы для ЭВМ.

3. Разработана методика создания и скрытого вложения цифрового водяного знака регистратора, устойчивого к атакам обфускацией, в байт-код class-файлов java-приложений информационной системы. Методикой в отличие от известных используется два вида цифровых водяных знаков: цифровой водяной знак регистратор информационной системы и тиражируемый цифровой водяной знак в class-файлы java-приложений информационной системы. Вложение ЦВЗ-регистратора в class-файлы с ключевой логикой, при атаке обфускацией увеличивает вероятность приведения java-приложения или class-файла в неработоспособное состояние с невозможностью получения оригинального исходного кода до 70-80%.

Разработанная методика позволяет:

– Учитывать инструменты обфускации, как инструменты атаки на цифровой водяной знак, а не оптимизации кода

– Проводить оценку целостности информационной системы и ее отдельных частей

– Использовать два типа цифровых водяных знаков для информационной системы, на уровне информационной системы и на уровне java-приложений, составляющих информационную систему

– Произвести вложение цифровых водяных знаков таким образом, что применение инструментов продвинутой обфускации приведет к неработоспособности исходного кода

– Возможность автоматизировать процесс доведя до реализации программы для ЭВМ.

4. Проведена оценка эффективности разработанных методик.

Доказано, что предложенные методики обладают большей эффективностью для решения задач, связанных с созданием и вложением устойчивых к атакам декомпиляцией и обфускацией цифровых водяных знаков в class-файлы java-приложений и информационных систем.

Эффективность разработанных методик достигается:

– анализом операционных кодов class-файлов;

– анализом ключевой логики в class-файлах, их связей внутри java-приложения и связей java-приложений внутри информационной системы;

– использованием недокументированных возможностей языка программирования Java и виртуальной машины Java, эквивалентные операционные коды байт-кода Java;

– возможностью адаптации под конкретные цели владельцев ИС при проведении оценки class-файлов и используемых в них функций, шаблонов проектирования кода.

Все результаты, выносимые на защиту, являются новыми и подтверждаются актами внедрения (Приложение Г, Д).

Основные результаты диссертационного исследования опубликованы в 24 печатных трудах, среди которых:

– 8 статей, опубликованных в изданиях, рекомендованных ВАК России, 4 из них без соавторства.

– 4 статьи, опубликованных в изданиях, входящих в перечень Scopus.

– Программа для ЭВМ «Модификатор байт-кода java-программы для скрытого вложения цифрового водяного знака посредством автоматического редактирования байт-кода class-файла», номер регистрации 2020664301 от 11.11.2020 г. (Приложение А).

– Программа для ЭВМ «Анализатор байт-кода java - программы для скрытого вложения цифрового водяного знака посредством автоматического редактирования байт-кода class-файла», номер регистрации 2020617872 от 15.07.2020 г. (Приложение Б).

– Программа для ЭВМ «Программное обеспечение по реализации стеганографических методов при передаче сообщения», номер регистрации 2022613440 от 14.03.2022 г. (Приложение В).

– 11 статей, опубликованных в изданиях, входящих в перечень РИНЦ РФ.

– Учебно-методическое пособие «Разработка защищенных приложений» (Россия, Санкт – Петербург, СПБГУТ им. проф. М.А. Бонч-Бруевича).

– тезисы докладов.

Сформулированы **рекомендации** по использованию результатов работы для выполнения производственных задач и в научных исследованиях. Разработанная методика по скрытому созданию и вложению увеличенного объема цифрового водяного знака в class-файлы может использоваться для контроля целостности class-файла и защиты интеллектуальной собственности в виде исходного кода. Разработанная методика создания и вложения устойчивого к атакам декомпиляцией цифрового водяного знака в class-файлы java-приложения может использоваться для защиты поставляемых программных продуктов от декомпиляции и дальнейшего использования исходного кода в сторонних

нелегитимных проектах. Разработанная методика создания и вложения устойчивого к атакам обфускацией цифрового водяного знака в class-файлы информационной системы может быть использована для контроля целостности информационной системы, защиты информационной системы от атак обфускацией нацеленных на удаление цифрового водяного знака с последующей кражей модулей информационной системы для использования в фарминговых целях. Все разработанные методики могут использоваться для защиты интеллектуальной собственности на уровнях class-файла, java-приложения и информационной системы.

Дальнейшие научные исследования по теме диссертационного исследования целесообразно продолжить в следующих направлениях:

– Исследование использования опкодов виртуальной машины Java отвечающих за операции работы со строками и коллекциями. Вычисление предельно допустимого количества эквивалентных замен в опкодах class-файла без потери производительности и увеличения объема class-файла;

– Разработка автоматизированных средств полного цикла анализа, модификации и проверки вложения ЦВЗ в class-файлах java-приложений и информационных систем;

– Разработка и повышение качеств существующих методик создания и вложения цифрового водяного знака в байт-код исполняемых файлов

– Исследование атак на ЦВЗ посредством эквивалентных замен опкодов. Расчет эффективности атак такого рода на тиражируемые ЦВЗ и ЦВЗ-регистраторы;

– Исследование устойчивости цифровых водяных знаков, созданных и вложенных посредством существующих методик к комплексным атакам нескольких видов, направленным на его повреждение и уничтожение.

Все результаты, выносимые на защиту, сопоставлены с пунктами 7 и 16 паспорта специальности 2.3.6 – «Методы и системы защиты информации,

информационная безопасность»: «Модели и методы формирования комплексов средств противодействия угрозам хищения (разрушения, модификации) информации и нарушения информационной безопасности для различного вида объектов защиты вне зависимости от области их функционирования» и «Методы, модели и средства выявления не декларированных возможностей программного обеспечения и технических средств. Противодействие скрытым каналам передачи данных. Выявление уязвимостей в системах различного типа», соответственно.

Список сокращений и обозначений

ВАК	–	высшая аттестационная комиссия при Министерстве науки и высшего образования Российской Федерации
ИС	–	информационная система
НИР	–	научно-исследовательская работа
ООП	–	объектно-ориентированное программирование
ОС	–	операционная система
ПО	–	программное обеспечение
РФ	–	Российская Федерация
СГ	–	стеганография
УК РФ	–	Уголовный кодекс Российской Федерации
ЦВЗ	–	цифровой водяной знак
ЭВМ	–	электронная вычислительная машина
AOT	–	Ahead-of-Time
API	–	Application Programming Interface
ARM	–	Advanced RISC Machine
ASCII	–	American Standard Code for Information Interchange
CF	–	Class File
CRC	–	Cyclic Redundancy Check (Циклический избыточный код)
DOM	–	Document Object Model
GUI	–	Graphical User Interface (Графический интерфейс пользователя)
HEX	–	Hexadecimal (обозначение шестнадцатеричной системы счисления)
HTML	–	HyperText Markup Language (Язык гипертекстовой разметки)
IT	–	Information Technology (информационные технологии)
JAR	–	Java Archive (Java-архив)
JDK	–	Java Development Kit (комплект разработчика приложений на языке Java)

JIT – Just-in-Time (динамическая компиляция)

JRE – Java Runtime Edition

JVM – Java Virtual Machine (Виртуальная машина Java)

LIFO – Last In, First Out

LOC – Line of Code (количество строк кода)

OS – Operating System

REST – Representational State Transfer (передача репрезентативного состояния)

SWT – Standard Widget Toolkit

UTF-8 – Unicode Transformation Format, 8-bit (формат преобразования Юникода, 8-бит)

XML – eXtensible Markup Language (расширяемый язык разметки)

Список используемой литературы

1. Блинов И. Н., Романчик В. С. Java. Промышленное программирование. – 2007.
2. Коржик С. О., Анфиногенов С. О. Обеспечение прав собственности на цифровые изображения в российской федерации с использованием технологии водяных знаков //Электросвязь. – 2012. – №. 8. – С. 44а-48.
3. Варновский Н. П. и др. Современное состояние исследований в области обфускации программ: определения стойкости обфускации //Труды Института системного программирования РАН. – 2014. – Т. 26. – №. 3. – С. 167-198.
4. Герлинг Е. Ю. и др. Модели нарушителей информационной безопасности //Известия высших учебных заведений. Технология легкой промышленности. – 2017. – Т. 35. – №. 1. – С. 27-30.
5. Грибунин В.Г. Цифровая стеганография / В.Г. Грибунин, И.Н. Оков, И.В. Туринцев. - М.: Солон-Пресс, 2009. - 265 с.
6. Грибунин В.Г., Агрsuperновский А.В., Балакин А.В., Сапожников С.А. Стеганография, цифровые водяные знаки и стеганоанализ. - М.: Вузовская книга, 2009. - 220 с.
7. Ермаков М. К., Вартанов С. П. Подход к проведению динамического анализа java-программ методом модификации виртуальной машины java //Труды Института системного программирования РАН. – 2015. – Т. 27. – №. 2. – С. 23-38.
8. Комяков Д. С., Елсаков С. М. Обзор методов внедрения статических водяных знаков в исходный код программного обеспечения //Южно-Уральская молодежная школа по математическому моделированию. – 2015. – С. 84-92.
9. Коржик В. И., Зинковская А. В., Пиварелис Л. Л. Точная аутентификация цифровых изображений вформате jpec // Научно-технические ведомости Санкт-Петербургского государственного политехнического университета. Информатика. Телекоммуникации. Управление. – 2009. – №. 6 (91).
10. Коржик В. И., Чесноков Р. В. Обнаружение стегосистем методом" слепого" стегоанализа //Информация и космос. – 2010. – №. 3. – С. 17-24.
11. Коржик В.И., Зенковская А.В., Луссе С.А. Метод точной аутентификации цифровых изображений с использованием технологии водяных знаков // Вопросы защиты информации. - 2008. - №. 1. - С. 8-14
12. Коржик В.И., Флаксман Д.А. Система цифровых водяных знаков с возможностью их извлечения из бумажных копий цифровых документов // Труды учебных заведений связи. 2019. Т. 5. № 3. С. 75–85. DOI:10.31854/1813-324X-2019-5-3-75-85
13. Коржик В.И., Яковлев В.А. Основы криптографии : учеб. пособие. СПб, 2017.

14. Коржик, В. И. Система цифровых водяных знаков с повторным вложением информации по различным алгоритмам / В. И. Коржик, А. И. Кочкарев, Д. А. Флаксман //Телекоммуникации. – 2014. – №. 7. – С. 22-33.
15. Коробейников А. Г., Кутузов И. М., Колесников П. Ю. Анализ методов обфускации //Кибернетика и программирование. – 2012. – №. 1. – С. 31-37.
16. Красов А. В., Верещагин А. С., Цветков А. Ю. Аутентификация программного обеспечения при помощи вложения цифровых водяных знаков в исполняемый код //Телекоммуникации. – 2013. – №. S7. – С. 27-29.
17. Красов А. В., Штеренберг С. И. Разработка методов защиты от копирования ПО на основе цифровых водяных знаков внедряемых в исполняемые и библиотечные файлы //Актуальные проблемы инфотелекоммуникаций в науке и образовании. – 2013. – С. 847-852.
18. Красов А.В., Верещагин А.С., Абатуров В.С. Методы скрытого вложения информации в исполняемые файлы // Известия Санкт-Петербургского государственного электротехнического университета ЛЭТИ. - 2012. - № 8. - С. 51-55.
19. Ларин А.А. Защита коммерчески значимой информации как одна из гарантий свободы предпринимательской деятельности // Теория и практика общественного развития. - № 19. - 2014. - С. 92-94.
20. Романов В. Ю. Моделирование и верификация архитектуры программного обеспечения, разработанного на языке Java //Современные информационные технологии и ИТ-образование. – 2013. – №. 9. – С. 343-348.
21. Сафонов В. О. Введение в Java-технологию //Международный журнал прикладных и фундаментальных исследований. – 2011. – №. 7. – С. 105-107.
22. Семашко А. В., Кулаков А. В. Криптографическая хеш-функция //Информационные системы и технологии ИСТ-2018. – 2018. – С. 534-538.
23. Соловьев В. В., Липский Н. В. Построение блоков обработки исключений при декомпиляции Java-байткода //Системная информатика. – 2013. – №. 2. – С. 1-22.
24. Темчишен А. А., Попков Ю. С. Современная стеганография //проблемы развития современного общества. – 2021. – С. 34-36.
25. Хашковский В. В., Лутай В. Н., Юрченко В. В. Сравнительная оценка времени выполнения программ на различных платформах //Известия Южного федерального университета. Технические науки. – 2009. – Т. 101. – №. 12.
26. Хомяков И.Н., Красов А.В. Возможность скрытого вложения информации в байт-код java//Информационные технологии моделирования и управления, 2014, № 2 (86), с. 185-191. EDN: RZHODN
27. Хорстман К., Корнелл Г. JAVA. Библиотека профессионала. Том II. Расширенные средства программирования. 9-е изд. - Москва: Издательский дом «Вильямс», 2016. - 1008 с

28. Цифровая стеганография и цифровые водяные знаки. Часть 1. Цифровая Стеганография.: Монография / [Коржик В.И. и др.]; под общ. ред. проф. В.И. Коржика. В.И.Коржик, К.А. Небаева, Е.Ю. Герлинг, П.С. Догиль, И.А. Федянин, – СПб: СПбГУТ, 2016. 226.с ISBN: 978-5-89160-125-3
29. Цифровая стеганография и цифровые водяные знаки. Часть 2 Цифровые водяные знаки: Монография / [Коржик В.И. и др.]; под общей редакцией В. И. Коржика. В. И. Коржик, С. О. Анфиногенов, А. И. Кочкарёв, И. А. Федянин, А. Г. Жувикин, Д. А. Флаксман, В. Г. Алексеев, – СПб: СПбГУТ, 2017. – ISBN: -978-5-89160-125-3
30. **Шариков П. И**. Алгоритм вложения цифрового водяного знака в исполняемые файлы необходимый для подтверждения авторства программы при судебной экспертизе //Молодежная научная школа кафедры" Защищенные системы связи". – 2021. – Т. 1. – №. 1. – С. 7-11.
31. **Шариков П. И.** Исследование возможностей методики скрытого вложения цифрового водяного знака в class-файлы на виртуализированных платформах с отличающейся архитектурой / Иванов А. В., Красов А. В., Шариков П. И //Научно-аналитический журнал «Вестник Санкт-Петербургского университета Государственной противопожарной службы МЧС России». – 2018. – №. 2. – С. 79-88.
32. **Шариков П. И.** Методика анализа байт-кода с целью оценки эффективности скрытого вложения информации и ее объемов в java-проект //Молодежная научная школа кафедры" Защищенные системы связи". – 2020. – Т. 1. – №. 1. – С. 30-34.
33. **Шариков П. И.** Методика быстрой обфускации class-файлов java-приложения с минимальным использованием ресурсов //Актуальные проблемы инфотелекоммуникаций в науке и образовании (АПИНО 2019). – 2019. – С. 548-551.
34. **Шариков П. И.** Методика защиты байт-кода java-программы от декомпиляции и хищения исходного кода злоумышленником / Красов А. В., Шариков П. И. //Вестник Санкт-Петербургского государственного университета технологии и дизайна. Серия 1: Естественные и технические науки. – 2017. – №. 1. – С. 47-50.
35. **Шариков П. И.** Методика нахождение величины наиболее выгодного контейнера в форматах исполняемых файлов //Наукоемкие технологии в космических исследованиях Земли. – 2015. – Т. 7. – №. 5. – С. 58-62.
36. **Шариков П. И.** Обеспечение безопасности java программ посредством вложения программного водяного знака / Красов А. В., Шариков П. И. //Актуальные проблемы инфотелекоммуникаций в науке и образовании (АПИНО 2018). – 2018. – С. 517-520.
37. **Шариков П. И.** Построение доверенной вычислительной среды / А. В. Красов, А. М. Гельфанд, В. И. Коржик [и др.]. – Санкт-Петербург :

Индивидуальный предприниматель Петрив Роман Богданович, 2019. – 108 с. – ISBN 978-5-6043143-2-6. – EDN RECXBI.

38. **Шариков П. И.**, Красов А. В. Исследование возможности использования java-агентов для вложения скрытого цифрового водяного знака непосредственно перед запуском java-приложения //Вестник Санкт-Петербургского государственного университета технологии и дизайна. Серия 1: Естественные и технические науки. – 2019. – №. 4. – С. 14-18.
39. **Шариков П. И.**, Красов А. В. Исследование уязвимости сериализации и десериализации данных в Java //Региональная информатика и информационная безопасность. – 2017. – С. 333-336.
40. **Шариков П. И**., Красов А. В. Исследование уязвимости сериализации и десериализации данных в Java //Региональная информатика и информационная безопасность. – 2017. – С. 333-336.
41. **Шариков П. И.**, Красов А. В., Штеренберг С. И. Методика создания и вложения цифрового водяного знака в исполняемые java файлы на основе замен опкодов //T-Comm-Телекоммуникации и Транспорт. – 2017. – Т. 11. – №. 3. – С. 66-70.
42. **Шариков П.И.** Методика защиты байт-кода JAVA-программы от декомпиляции и хищения исходного кода злоумышленником / Красов А. В., Шариков П. И. //Вестник Санкт-Петербургского государственного университета технологии и дизайна. Серия 1: Естественные и технические науки. – 2017. – №. 1. – С. 47-50.
43. **Шариков, П. И.** Исследование атаки обфускацией на байт-код java-приложения с целью разрушения или повреждения цифрового водяного знака / П. И. Шариков // I-methods. – 2022. – Т. 14. – № 1. – EDN GQGKIV.
44. **Шариков, П. И.** Исследование возможности вложения цифрового водяного знака в байт-код путем замены уязвимого байт-кода Java класса / П. И. Шариков, А. В. Красов // Информационная безопасность регионов России (ИБРР-2017) : Материалы конференции, Санкт-Петербург, 01–03 ноября 2017 года. – Санкт-Петербург: Санкт-Петербургское Общество информатики, вычислительной техники, систем связи и управления, 2017. – С. 499-500. – EDN YVFPRS.
45. **Шариков, П. И.** Исследование устойчивости цифрового водяного знака к атакам направленной декомпиляции составных частей приложения Java // Электронный сетевой политематический журнал "Научные труды КубГТУ". – 2022. – № 2. – С. 100-112. – EDN QWHOIV.
46. **Шариков, П. И.** Методика обфускации байт-кода java-приложения с целью его защиты от атак декомпиляцией / П. И. Шариков // Вестник Санкт-Петербургского государственного университета технологии и дизайна. Серия 1: Естественные и технические науки. – 2022. – № 1. – С. 64-72. – DOI 10.46418/2079-8199_2022_1_10. – EDN AUOFNA.

47. **Шариков П. И.** Методика создания и скрытого вложения цифрового водяного знака в байт-код class-файла на основе не декларированных возможностей виртуальной машины java // Современная наука: актуальные проблемы теории и практики. Серия: Естественные и Технические Науки. - 2023. -№07/2. -С. 165-174 DOI 10.37882/2223-2982.2023.7-2.37
48. Шипилов А. и др. Авторские права в цифровую эпоху // Ephoto. - 2002. - №. 3 (13).
49. 周志明. 深入理解 Java 虚拟机: JVM 高级特性与最佳实践. – 机械工业出版社, 2020. (Чжоу Чжимин, Глубокое понимание виртуальной машины Java: расширенные функции и лучшие практики JVM, — Machinery Industry Press, 2020.)
50. 玉田春昭 et al. Java バイトコードを対象とした命令の頻度解析による適用難読化ツールの特定 //コンピュータセキュリティシンポジウム 2019 論文集. – 2019. – Т. 2019. – С. 119-124. (Харуаки Тамада и др. Идентификация применяемых средств обфускации с помощью частотного анализа инструкций, нацеленных на байт-код Java //Материалы симпозиума по компьютерной безопасности 2019. – 2019. – Т. 2019. – С.)
51. AboRizka M., Kouta M., Mohamed H. Practical View of Mobile Agent Obfuscation in Digital Cash System //Journal of the ACS. – 2007. – Т. 1.
52. Albert E. et al. Cost analysis of java bytecode //European symposium on programming. – Springer, Berlin, Heidelberg, 2007. – С. 157-172.
53. Allatori. [Электронный ресурс] – Режим доступа к рес.: https://allatori.com/ (дата обращения: 04.02.2022).
54. Androsch L. C. Analysing Raw Process Dumps as Managed Java and. NET Dumps/Author Lukas Christoph Androsch, BSc. – 2021.
55. Arboit G. A method for watermarking java programs via opaque predicates //The Fifth International Conference on Electronic Commerce Research (ICECR-5). – 2002. – С. 102-110
56. Ashbacher C. Jclasslib bytecode viewer //Mathematics and Computer Education. – 2005. – Т. 39. – №. 1. – С. 84.
57. Bangerter E., Bühlmann S., Kirda E. Efficient and stealthy instruction tracing and its applications in automated malware analysis: Open problems and challenges //International Workshop on Open Problems in Network Security. – Springer, Berlin, Heidelberg, 2011. – С. 55-64.
58. Barak B. et al. On the (im) possibility of obfuscating programs //Annual international cryptology conference. – Springer, Berlin, Heidelberg, 2001. – С. 1-18.
59. Batchelder M., Hendren L. Obfuscating Java: The most pain for the least gain //International Conference on Compiler Construction. – Springer, Berlin, Heidelberg, 2007. – С. 96-110.

60. Bissyandé T. F. et al. Popularity, interoperability, and impact of programming languages in 100,000 open source projects //2013 IEEE 37th annual computer software and applications conference. – IEEE, 2013. – С. 303-312.
61. Brody R. G., Mulig E., Kimball V. Phishing, pharming and identity theft //Academy of Accounting & Financial Studies Journal. – 2007. – Т. 11. – №. 3.
62. Bytecode Viewer. [Электронный ресурс] – Режим доступа к рес.: https://github.com/Konloch/bytecode-viewer
63. Cavaj Java Decompiler for Windows. [Электронный ресурс] – Режим доступа к рес.: https://cavaj-java-decompiler.en.softonic.com/ (дата обращения: 04.12.21).
64. Cavaj Java Decompiler V1.11. [Электронный ресурс] – Режим доступа к рес.: https://sureshotsoftware.com/products/cavaj/ (дата обращения: 04.12.21).
65. CFR. [Электронный ресурс] – Режим доступа к рес.: http://www.benf.org/other/cfr/ (дата обращения: 04.12.21).
66. Chen J. et al. Software watermarking for java program based on method name encoding //International Conference on Advanced Intelligent Systems and Informatics. – Springer, Cham, 2017. – С. 865-874.
67. Chiba S. Foreign language interfaces by code migration //Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences. – 2019. – С. 1-13.
68. Chiba S. Javassist //Proc. of 2nd Int'l Conf. on Generative Programming and Component Engineering (GPCE'03). – Springer-Verlag, 2003.
69. Chiba S. Javassist—a reflection-based programming wizard for Java //Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java. – ACM, 1998. – Т. 174. – С. 21.
70. Chionis I., Chroni M., Nikolopoulos S. D. Evaluating the WaterRpg software watermarking model on Java application programs //Proceedings of the 17th Panhellenic Conference on Informatics. – 2013. – С. 144-151.
71. Choi J. D. et al. Escape analysis for Java //Acm Sigplan Notices. – 1999. – Т. 34. – №. 10. – С. 1-19.
72. Coley. Recaf. [Электронный ресурс] – Режим доступа к рес.: https://www.coley.software/Recaf/ (дата обращения: 04.12.21).
73. Collberg C. et al. Graph theoretic software watermarks: Implementation, analysis, and attacks //International Workshop on Information Hiding. – Springer, Berlin, Heidelberg, 2004. – С. 192-207.
74. Commons Apache BCEL. [Электронный ресурс] – Режим доступа к рес.: https://commons.apache.org/proper/commons-bcel/ (дата обращения: 04.02.2022).
75. Cox. I.J. Digital Watermarking and Steganography, Second Edition: Morgan Kaufmann, 2008. 593 p.
76. Curran D., Hurley N. J., Cinnéide M. Ó. Securing java through software watermarking //Proceedings of the 2nd international conference on Principles and

practice of programming in Java. – Computer Science Press, Inc., 2003. – С. 145-148.
77. D4j - The Free Java Decompiler for Everyone. [Электронный ресурс] – Режим доступа к рес.: http://www.secureteam.net/d4j/ (дата обращения: 04.12.21).
78. Daoud H., Dagenais M. Multilevel analysis of the java virtual machine based on kernel and userspace traces //Journal of Systems and Software. – 2020. – Т. 167. – С. 110589.
79. DJ Java Decompiler. [Электронный ресурс] – Режим доступа к рес.: http://www.neshkov.com/ (дата обращения: 04.12.21).
80. Docs.oracle.com. [Электронный ресурс] – Режим доступа к рес.: https://docs.oracle.com/javase/9/tools/jhsdb.htm#JSWOR-GUID-0345CAEB-71CE-4D71-97FE-AA53A4AB028E (дата обращения: 04.02.2022).
81. Dufour B., Hendren L., Verbrugge C. * J: a tool for dynamic analysis of Java programs //Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. – 2003. – С. 306-307.
82. Evans B. J., Gough J., Newland C. Optimizing Java: Practical Techniques for Improving JVM Application Performance. – " O'Reilly Media, Inc.", 2018.
83. Fernflower. [Электронный ресурс] – Режим доступа к рес.: https://github.com/fesh0r/fernflower (дата обращения: 04.12.21).
84. Free Java™ Bytecode Obfuscator and Shrinker. [Электронный ресурс] – Режим доступа к рес.: https://www.yworks.com/products/yguard (дата обращения: 04.02.2022).
85. Fukushima K., Sakurai K. A software fingerprinting scheme for java using classfiles obfuscation //International Workshop on Information Security Applications. – Springer, Berlin, Heidelberg, 2003. – С. 303-316.
86. Garg S. et al. Candidate indistinguishability obfuscation and functional encryption for all circuits //SIAM Journal on Computing. – 2016. – Т. 45. – №. 3. – С. 882-929.
87. Goldwasser S., Rothblum G. N. On best-possible obfuscation //Theory of Cryptography Conference. – Springer, Berlin, Heidelberg, 2007. – С. 194-213.
88. Gosling J. et al. The Java language specification. – Addison-Wesley Professional, 2000.
89. Guardsquare. [Электронный ресурс] / Режим доступа: https://www.guardsquare.com/proguard (дата обращения: 04.12.21).
90. Hamilton J., Danicic S. An evaluation of current java bytecode decompilers //2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation. – IEEE, 2009. – С. 129-136.
91. Hamilton J., Danicic S. An evaluation of static java bytecode watermarking //Proceedings of the International Conference on Computer Science and Applications (ICCSA'10), The World Congress on Engineering and Computer Science (WCECS'10), San Francisco. – 2010.

92. Hamilton J., Danicic S. An evaluation of the resilience of static java bytecode watermarks against distortive attacks // IAENG International Journal of Computer Science. – 2011. – Т. 38. – №. 1. – С. 1-15.
93. Hamilton J., Danicic S. An evaluation of the resilience of static java bytecode watermarks against distortive attacks //IAENG International Journal of Computer Science. – 2011. – Т. 38. – №. 1. – С. 1-15.
94. Harrand N. et al. Java decompiler diversity and its application to meta-decompilation //Journal of Systems and Software. – 2020. – Т. 168. – С. 110645.
95. Helios. [Электронный ресурс] – Режим доступа к рес.: https://ci.samczsun.com/job/helios-decompiler/ (дата обращения: 04.12.21).
96. Hillert E. Obfuscate java bytecode: an evaluation ofobfuscating transformations using jbco. – 2014.
97. Intellij-community. [Электронный ресурс] – Режим доступа к рес.: https://github.com/JetBrains/intellij-community/tree/master/plugins/java-decompiler/engine (дата обращения: 04.12.21).
98. Ishizaki K. et al. A study of devirtualization techniques for a Java Just-In-Time compiler //Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. – 2000. – С. 294-310.
99. JAD. Wikipedia. [Электронный ресурс] – Режим доступа к рес.: https://en.wikipedia.org/wiki/JAD_(software) (дата обращения: 04.12.21).
100. Java Archive Grinder. [Электронный ресурс] – Режим доступа к рес.: http://jarg.sourceforge.net/ (дата обращения: 04.02.2022).
101. Java Decompiler. [Электронный ресурс] – Режим доступа к рес.: http://java-decompiler.github.io/ (дата обращения: 04.12.21).
102. Javaguard. [Электронный ресурс] – Режим доступа к рес.: http://sourceforge.net/projects/javaguard/ (дата обращения: 04.02.2022).
103. Javassist. [Электронный ресурс] – Режим доступа к рес.: https://www.javassist.org/ (дата обращения: 04.12.21).
104. JBCO: the Java ByteCode Obfuscator. [Электронный ресурс] – Режим доступа к рес.: http://www.sable.mcgill.ca/JBCO/ (дата обращения: 04.02.2022).
105. JBE - Java Bytecode Editor [Электронный ресурс] – Режим доступа к рес.: https://set.ee/jbe/ (дата обращения: 21.11.21).
106. Jboss-javassist. [Электронный ресурс] – Режим доступа к рес.: https://github.com/jboss-javassist/javassist (дата обращения: 04.12.21).
107. JBVD - Java Bytecode Viewer & Decompiler. [Электронный ресурс] – Режим доступа к рес.: https://sourceforge.net/projects/jbdec/ (дата обращения: 04.12.21).
108. Ji J. H., Woo G., Cho H. G. A plagiarism detection technique for Java program using bytecode analysis //2008 third international conference on

convergence and hybrid information technology. – IEEE, 2008. – Т. 1. – С. 1092-1098.

109. JObfuscator. [Электронный ресурс] – Режим доступа к рес.: https://www.pelock.com/products/jobfuscator (дата обращения: 04.02.2022).
110. JODE. [Электронный ресурс] – Режим доступа к рес.: http://jode.sourceforge.net/ (дата обращения: 04.02.2022).
111. Kalinovsky A. Covert Java: Techniques for Decompiling, Patching, and Reverse Engineering. – Pearson Higher Education, 2004.
112. Kanani P. et al. Obfuscation: maze of code //2017 2nd International Conference on Communication Systems, Computing and IT Applications (CSCITA). – IEEE, 2017. – С. 11-16.
113. Karlof C. et al. Dynamic pharming attacks and locked same-origin policies for web browsers //Proceedings of the 14th ACM conference on Computer and communications security. – 2007. – С. 58-71.
114. Karnick M. et al. A qualitative analysis of java obfuscation //proceedings of 10th IASTED international conference on software engineering and applications, Dallas TX, USA. – 2006.
115. Kim S. et al. Predicting method crashes with bytecode operations //Proceedings of the 6th India software engineering conference. – 2013. – С. 3-12.
116. Korzhik, V. Digital watermarking under filterring and addition noise attack condition / Valery Korzhik, Guillermo Morales-Luna, Irina Marakova, Carlos PatiñoRuvalcaba // International Workshop on Mathematical Methods, Models, and Architectures for Computer Network Security. – Springer, Berlin, Heidelberg, 2003. – С. 371-382.
117. Korzhik, V. I. Concatenated Digital Watermarking System Robust to Different Removal Attacks / V. I. Korzhik, G. Morales-Luna, A.Kochkarev, D. Flaksman // Computer Science and Information Systems.2014.Т.11.№4.С.1581-1594.
118. Korzhik, V. I. Fingerprinting System for Still Images Based on the Use of a Holographic Transform Domain / V. I. Korzhik, G. Morales-Luna, A.Kochkarev, I. Shevchuk // in FedCSIS, M. Ganzha, L. A. Maciaszek, and M. Paprzycki, Eds., 2013, pp. 585–590
119. Kostelansky J., Dedera L. Decompiling java bytecode: from common java to newest jdk8 features, from obsolete decompilers to cutting edge technology //Science & Military Journal. – 2018. – Т. 13. – №. 1. – С. 23-30.
120. Krakatau. [Электронный ресурс] – Режим доступа к рес.: https://github.com/Storyyeller/Krakatau (дата обращения: 04.12.21).
121. Krasov A. V., Arshinov A. S., Ushakov I. A. Embedding the hidden information into java byte code based on operands' interchanging //ARPN Journal of Engineering and Applied Sciences. – 2018. – Т. 13. – №. 8. – С. 2746-2752.

122. Kumar K., Kaur P. A Comparative Analysis of Static and Dynamic Java Bytecode Watermarking Algorithms // Software Engineering. – Springer, Singapore, 2019. – С. 319-334.
123. Kumar K., Kehar V., Kaur P. A comparative analysis of static java bytecode software watermarking algorithms //African J Comput ICT. – 2015. – Т. 8. – №. 4. – С. 201-208.
124. Kumar K., Kehar V., Tulsi P. K. An Evaluation of Dynamic Java Bytecode Software Watermarking Algorithms //Algorithms International Journal of Security and Its Applications. – 2016. – Т. 10. – №. 7. – С. 147-156.
125. Lam P. et al. The Soot framework for Java program analysis: a retrospective //Cetus Users and Compiler Infastructure Workshop (CETUS 2011). – 2011. – Т. 15. – №. 35.
126. Leroy X. Java bytecode verification: algorithms and formalizations //Journal of Automated Reasoning. – 2003. – Т. 30. – №. 3. – С. 235-269. (понятным верификатору)
127. Lindholm T., Yellin F. The java virtual machine specification. – 1996.
128. Lindholm T. et al. The Java virtual machine specification. – Pearson Education, 2014.
129. Logozzo F., Fähndrich M. On the relative completeness of bytecode analysis versus source code analysis //International Conference on Compiler Construction. – Springer, Berlin, Heidelberg, 2008. – С. 197-212.
130. Luyten. [Электронный ресурс] – Режим доступа к рес.: https://github.com/deathmarine/Luyten (дата обращения: 04.12.21).
131. Macbride J. et al. A comparative study of java obfuscators //in Proceedings of the IASTED International Conference on Software Engineering and Applications (SEA 2005. – 2005.
132. Mariano Ceccato et al. A large study on the effect of code obfuscation on the quality of java code //Empirical Software Engineering. – 2015. – Т. 20. – С. 1486-1524.
133. Miecznikowski J. New algorithms for a Java decompiler and their implementation in Soot. – 2003.
134. Miecznikowski J., Hendren L. Decompiling Java bytecode: Problems, traps and pitfalls //International Conference on Compiler Construction. – Springer, Berlin, Heidelberg, 2002. – С. 111-127.
135. Monden A. et al. A practical method for watermarking java programs //Proceedings 24th Annual International Computer Software and Applications Conference. COMPSAC2000. – IEEE, 2000. – С. 191-197.
136. Monga V., Evans B.L. Perceptual Image Hashing Via Feature Points: Performance Evaluation and Tradeoffs // IEEE Transactions on Image Processing. 2006. Vol. 15. Iss. 11. PP. 3452–3465. DOI:10.1109/TIP.2006.881948

137. Naeem N. A., Hendren L. Programmer-friendly decompiled Java //14th IEEE International Conference on Program Comprehension (ICPC'06). – IEEE, 2006. – С. 327-336.
138. Naharuddin A., Wibawa A. D., Sumpeno S. A high capacity and imperceptible text steganography using binary digit mapping on ASCII characters //2018 International Seminar on Intelligent Technology and Its Applications (ISITIA). – IEEE, 2018. – С. 287-292.
139. Nolan G., Zukowski J. Decompiling Java. – Apress, 2004.
140. Oaks S. Java Performance: The Definitive Guide: Getting the Most Out of Your Code. – " O'Reilly Media, Inc.", 2014.
141. Park H. et al. An Evaluation of the Proguard, Obfuscation Tool for Android //Proceedings of the Korea Information Processing Society Conference. – Korea Information Processing Society, 2012. – С. 730-733.
142. Pfitzmann B. Information Hiding Terminology, in Information Hiding, Springer Lecture Notes in Computer Science, v.1174, 1996, 347-350.
143. Piao Y., Jung J., Yi J. H. Structural and functional analyses of ProGuard obfuscation tool //The Journal of Korean Institute of Communications and Information Sciences. – 2013. – Т. 38. – №. 8. – С. 654-662.
144. PreEmptive S. DashO-The premier Java obfuscator and efficiency enhancing tool // http://www.agtech.co.jp/products/preemptive/dasho/index.html
145. Procyon. [Электронный ресурс] – Режим доступа к рес.: https://github.com/mstrobel/procyon/ (дата обращения: 04.12.21).
146. Prweb. [Электронный ресурс] – Режим доступа к рес.: https://www.prweb.com/releases/2009/04/prweb2337824.htm (дата обращения: 04.02.2022).
147. Rainer R. K., Prince B. Introduction to information systems. – John Wiley & Sons, 2021.
148. Recaf. [Электронный ресурс] – Режим доступа к рес.: https://github.com/Col-E/Recaf (дата обращения: 04.12.21).
149. Rose E. Lightweight bytecode verification //Journal of Automated Reasoning. – 2003. – Т. 31. – №. 3. – С. 303-334.
150. Saccente N. et al. Project achilles: A prototype tool for static method-level vulnerability detection of Java source code using a recurrent neural network //2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW). – IEEE, 2019. – С. 114-121.
151. Sakabe Y., Soshi M., Miyaji A. Java obfuscation with a theoretical basis for building secure mobile agents //IFIP International Conference on Communications and Multimedia Security. – Springer, Berlin, Heidelberg, 2003. – С. 89-103.
152. Sari M. M. et al. Measuring total mercury due to small-scale gold mining activities to determine community vulnerability in Cihonje, Central Java, Indonesia //Water Science and Technology. – 2016. – Т. 73. – №. 2. – С. 437-444.

153. **Sharikov P. I.** Research of the Possibility of Hidden Embedding of a Digital Watermark Using Practical Methods of Channel Steganography / P. I. Sharikov, A. V. Krasov, A. M. Gelfand, N. A. Kosov //Intelligent Distributed Computing XIII. – Springer International Publishing, 2020. – С. 203-209.

154. **Sharikov P. I.**, Krasov A. V., Volkogonov V. N. A study of the correctness of the execution of a class file with an embedded digital watermark in different environments //IOP Conference Series: Materials Science and Engineering. – IOP Publishing, 2020. – Т. 862. – №. 5. – С. 052052.

155. **Sharikov, P.** A Technique for Analyzing Bytecode in a Java Project for the Purpose of an Automated Assessment of the Possibility and Effectiveness of the Hidden Investment of Information and its Volumes in a Java Project / A. Krasov, P. Sharikov // 12th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops, ICUMT 2020. – Brno: Institute of Electrical and Electronics Engineers, 2020. – P. 258-263. – DOI 10.1109/ICUMT51630.2020.9222419. – EDN BOWZHK.

156. **Sharikov, Pavel**. A Technique for Detecting the Substitution of a Java-Module of an Information System Prone to Pharming with Using a Hidden Embedding of a Digital Watermark Resistant to Decompilation / Pavel, S., Andrey, K., Artem, G., & Ernest, B. //2021 13th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT). – IEEE, 2021. – С. 219-223.

157. Shi J. C. C. W. Q., Lv G. Implementation of bytecode-based software watermarking for java programs.

158. Silva J. E. An overview of cryptographic hash functions and their uses //GIAC. – 2003. – Т. 6.

159. Stackoverflow, maximum size of a Java .class file. [Электронный ресурс] – Режим доступа к рес.: https://stackoverflow.com/questions/42294998/what-is-the-maximum-size-of-a-java-class-file (дата обращения: 18.10.21).

160. Stackoverflow, recommended size of a class. [Электронный ресурс] – Режим доступа к рес.: https://stackoverflow.com/questions/16351566/what-is-the-recommended-size-of-a-class (дата обращения: 18.10.21).

161. Stair R., Reynolds G. Fundamentals of information systems. – Cengage Learning, 2017.

162. Stärk R. F., Schmid J., Börger E. Java and the Java virtual machine: definition, verification, validation. – Springer Science & Business Media, 2012.

163. Stern J. P., Hachez G., Koeune F., Quisquater J. J. Robust object watermarking: Application to code // Information Hiding, 1999. P. 368-378.

164. Stringer Java Obfuscator. [Электронный ресурс] – Режим доступа к рес.: https://jfxstore.com/stringer/ (дата обращения: 04.02.2022).

165. Sun W. et al. Code Search based on Context-aware Code Translation //arXiv preprint arXiv:2202.08029. – 2022.

166. The Java® Virtual Machine Specification. Java SE 19 Edition [Электронный ресурс] – Режим доступа к рес.: https://docs.oracle.com/javase/specs/jvms/se19/html/ (дата обращения: 21.11.21).
167. TIOBE Index| TIOBE–The Software Quality Company [Электронный ресурс]. Режим доступа к рес.: https://www. tiobe. com/tiobe-index/ (дата обращения: 05.08.2021)
168. Venners B. The java virtual machine //Java and the Java virtual machine: definition, verification, validation. – 1998.
169. Wermke D. et al. A large scale investigation of obfuscation use in google play //Proceedings of the 34th Annual Computer Security Applications Conference. – 2018. – С. 222-235.
170. Yang H. Java Tools Tutorials-Herong's Tutorial Examples. – HerongYang. com, 2020.
171. Zelix Klassmaster. [Электронный ресурс] – Режим доступа к рес.: http://www.zelix.com/klassmaster/ (дата обращения: 04.02.2022).
172. Красов А. В., Шариков П. И. Метод использования самомодифицирующегося кода для защиты приложения с кодовым зашумлением //Телекоммуникационные и вычислительные системы. – 2016. – С. 118-121.

Приложение А

Сведения о регистрации программы для ЭВМ «Модификатор байт-кода java-программы для скрытого вложения цифрового водяного знака посредством автоматического редактирования байт-кода class-файла»

РОССИЙСКАЯ ФЕДЕРАЦИЯ

RU2020664301

ФЕДЕРАЛЬНАЯ СЛУЖБА
ПО ИНТЕЛЛЕКТУАЛЬНОЙ СОБСТВЕННОСТИ

ГОСУДАРСТВЕННАЯ РЕГИСТРАЦИЯ ПРОГРАММЫ ДЛЯ ЭВМ

Номер регистрации (свидетельства):
2020664301
Дата регистрации: 11.11.2020
Номер и дата поступления заявки:
2020663386 30.10.2020
Дата публикации и номер бюллетеня:
11.11.2020 Бюл. № 11

Автор(ы):
Красов Андрей Владимирович (RU),
Пешков Андрей Иванович (RU),
Шариков Павел Иванович (RU),
Жиркова Галина Петровна (RU)
Правообладатель(и):
Федеральное государственное бюджетное образовательное учреждение высшего образования «Санкт-Петербургский государственный университет телекоммуникаций им. проф. М.А.Бонч-Бруевича» (RU)

Название программы для ЭВМ:
Модификатор байт-кода java-программы для скрытого вложения цифрового водяного знака посредством автоматического редактирования байт-кода class-файла

Реферат:
Программа предназначена для скрытого вложения цифрового водяного знака в байт-код java-программы посредством автоматической модификации операционных команд class-файла. Её областью применения является коммерческое программное обеспечение, государственные программные разработки в области информационной безопасности. Обладает функциональными возможностями: фундаментальный анализ байт-кода class-файла; замена и модификация операционных команд в байт-коде class-файла, благодаря встроенному словарю опкодов виртуальной машины; вложение цифрового водяного знака в байт-код без перекомпиляции java-программы.

Язык программирования: Java

Объем программы для ЭВМ: 20 КБ

Приложение Б

Сведения о регистрации программы для ЭВМ «Анализатор байт-кода java - программы для скрытого вложения цифрового водяного знака посредством автоматического редактирования байт-кода class-файла»

РОССИЙСКАЯ ФЕДЕРАЦИЯ

RU2020617872

ФЕДЕРАЛЬНАЯ СЛУЖБА
ПО ИНТЕЛЛЕКТУАЛЬНОЙ СОБСТВЕННОСТИ

ГОСУДАРСТВЕННАЯ РЕГИСТРАЦИЯ ПРОГРАММЫ ДЛЯ ЭВМ

Номер регистрации (свидетельства):
2020617872
Дата регистрации: 15.07.2020
Номер и дата поступления заявки:
2020616770 29.06.2020
Дата публикации и номер бюллетеня:
15.07.2020 Бюл. № 7
Контактные реквизиты:
нет

Автор(ы):
Красов Андрей Владимирович (RU),
Пешков Андрей Иванович (RU),
Шариков Павел Иванович (RU)
Правообладатель(и):
Федеральное государственное бюджетное образовательное учреждение высшего образования «Санкт-Петербургский государственный университет телекоммуникаций им. проф. М.А. Бонч-Бруевича» (СПбГУТ) (RU)

Название программы для ЭВМ:
«Анализатор байт-кода java - программы для скрытого вложения цифрового водяного знака посредством автоматического редактирования байт-кода class-файла»

Реферат:
Программа предназначена для скрытого вложения цифрового водяного знака в байт-код java-программы посредством автоматического анализа и редактирования class-файла. Программа может применяться для коммерческого программного обеспечения, государственных программных разработок в области информационной безопасности. Программа обеспечивает выполнение следующих функций: чтение class-файла; анализ операторов условных переходов; замена операторов условных переходов на эквивалентные с заданной последовательностью бит; вложение цифрового водяного знака в байт-код class-файла; использование менее объемных операционных кодов без перекомпиляции java-программы.

Язык программирования: Java

Объем программы для ЭВМ: 30 Кб

Приложение В

Сведения о регистрации программы для ЭВМ «Программное обеспечение по реализации стеганографических методов при передаче сообщения»

РОССИЙСКАЯ ФЕДЕРАЦИЯ

RU2022613440

ФЕДЕРАЛЬНАЯ СЛУЖБА
ПО ИНТЕЛЛЕКТУАЛЬНОЙ СОБСТВЕННОСТИ

ГОСУДАРСТВЕННАЯ РЕГИСТРАЦИЯ ПРОГРАММЫ ДЛЯ ЭВМ

Номер регистрации (свидетельства):
2022613440
Дата регистрации: 14.03.2022
Номер и дата поступления заявки:
2022612582 24.02.2022
Дата публикации и номер бюллетеня:
14.03.2022 Бюл. № 3
Контактные реквизиты:
нет

Автор(ы):
Красов Андрей Владимирович (RU),
Гельфанд Артем Максимович (RU),
Шариков Павел Иванович (RU),
Катасонов Александр Игоревич (RU)
Правообладатель(и):
Федеральное государственное бюджетное образовательное учреждение высшего образования «Санкт-Петербургский государственный университет телекоммуникаций им. проф. М.А. Бонч-Бруевича» (СПбГУТ) (RU)

Название программы для ЭВМ:
Программное обеспечение по реализации стеганографических методов при передаче сообщения

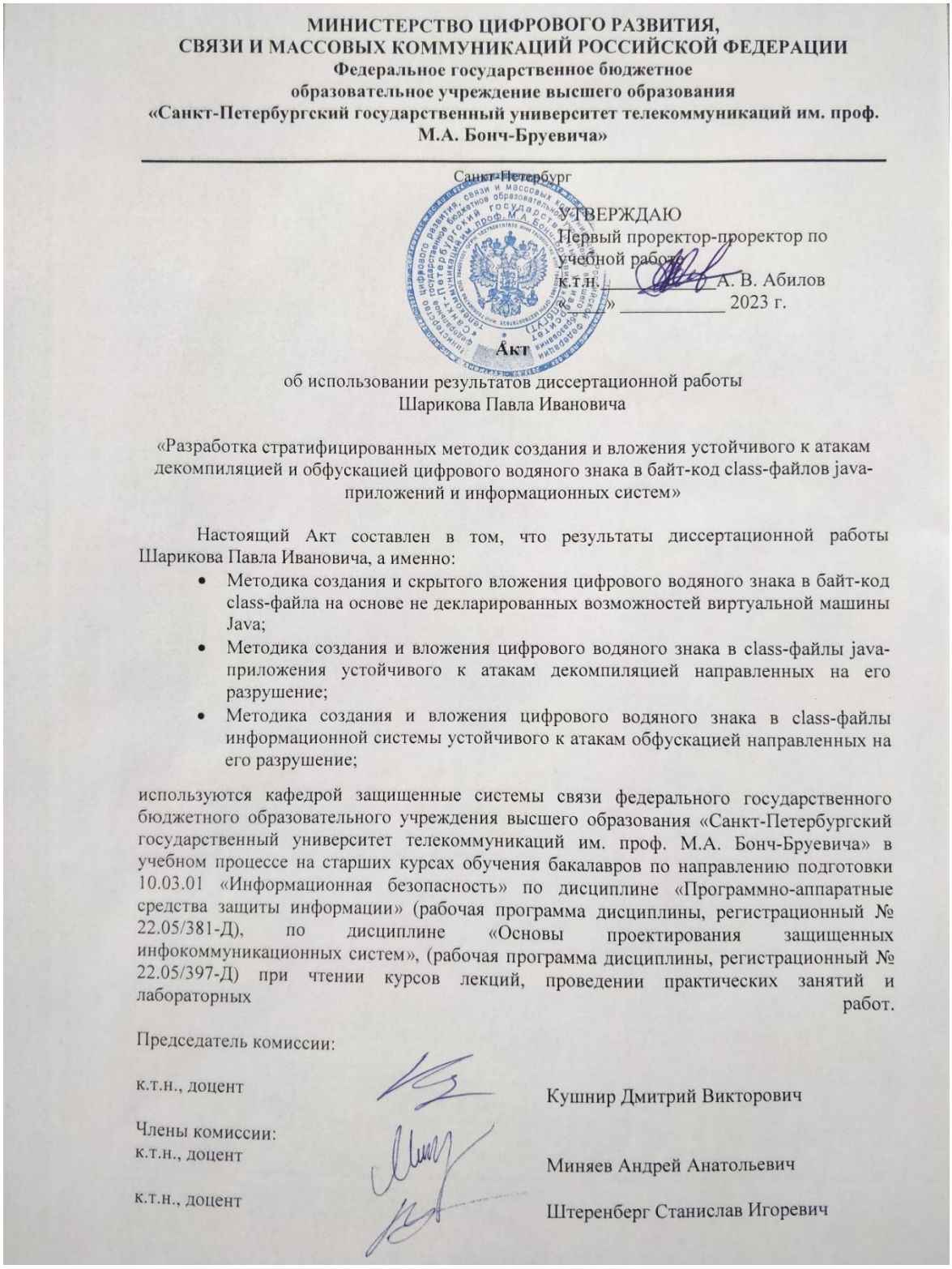
Реферат:
Программа предназначена для реализации стеганографических методов. Область применения: стеговложения. Функциональные возможности программы: в программе производится алгоритмический перебор пикселей изображения, необходимых для внедрения информации. В соответствии с методом наименее значащего бита внедряемая информация первоначально конвертируется в битовое представление, после чего производится её побитовое вложение в крайние значения цветовых компонентов изображения.

Язык программирования: C++

Объем программы для ЭВМ: 9 216 Байт

Приложение Г

Копия Акта использования результатов исследования в «Санкт-Петербургском государственном университете телекоммуникаций им. проф. М.А. Бонч-Бруевича»



МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ,
СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Санкт-Петербургский государственный университет телекоммуникаций им. проф. М.А. Бонч-Бруевича»

Санкт-Петербург

УТВЕРЖДАЮ
Первый проректор-проректор по учебной работе
к.т.н. _____ А. В. Абилов
«___» _____ 2023 г.

Акт

об использовании результатов диссертационной работы
Шарикова Павла Ивановича

«Разработка стратифицированных методик создания и вложения устойчивого к атакам декомпиляцией и обфускацией цифрового водяного знака в байт-код class-файлов java-приложений и информационных систем»

Настоящий Акт составлен в том, что результаты диссертационной работы Шарикова Павла Ивановича, а именно:

- Методика создания и скрытого вложения цифрового водяного знака в байт-код class-файла на основе не декларированных возможностей виртуальной машины Java;
- Методика создания и вложения цифрового водяного знака в class-файлы java-приложения устойчивого к атакам декомпиляцией направленных на его разрушение;
- Методика создания и вложения цифрового водяного знака в class-файлы информационной системы устойчивого к атакам обфускацией направленных на его разрушение;

используются кафедрой защищенные системы связи федерального государственного бюджетного образовательного учреждения высшего образования «Санкт-Петербургский государственный университет телекоммуникаций им. проф. М.А. Бонч-Бруевича» в учебном процессе на старших курсах обучения бакалавров по направлению подготовки 10.03.01 «Информационная безопасность» по дисциплине «Программно-аппаратные средства защиты информации» (рабочая программа дисциплины, регистрационный № 22.05/381-Д), по дисциплине «Основы проектирования защищенных инфокоммуникационных систем», (рабочая программа дисциплины, регистрационный № 22.05/397-Д) при чтении курсов лекций, проведении практических занятий и лабораторных работ.

Председатель комиссии:
к.т.н., доцент — Кушнир Дмитрий Викторович

Члены комиссии:
к.т.н., доцент — Миняев Андрей Анатольевич

к.т.н., доцент — Штеренберг Станислав Игоревич

Приложение Д

Копия Акта использования результатов исследования в ООО «ДИЛЕР»

www.raisecom.su
Телефон:
(812)970-23-53
sales@raisecom.su
ООО " ДИЛЕР " ИНН 7840448665 КПП 784201001
191119, Санкт-Петербург г, вн.тер.г. муниципальный округ Лиговка-Ямская, ул Роменская, д. 10, литера К, помещ. 5
р/с 40702810537080000493 в Филиале ОПЕРУ ПАО БАНК ВТБ в Санкт-Петербурге
г.Санкт-Петербург
БИК 044030704 к/с 30101810200000000704

АКТ
О внедрении научных результатов,
полученных Шариковым Павлом Ивановичем

Комиссия в составе:

- Шуйский Андрей Николаевич
- Малинин Никита Игоревич

составила настоящий акт о том, что научные результаты, полученные Шариковым Павлом Ивановичем, а именно:

1. Методика создания и скрытого вложения цифрового водяного знака в байт-код class-файла на основе не декларированных возможностей виртуальной машины Java.
2. Методика создания и вложения цифрового водяного знака в class-файлы java-приложения устойчивого к атакам декомпиляцией направленных на его разрушение.
3. Методика создания и вложения цифрового водяного знака в class-файлы информационной системы устойчивого к атакам обфускацией направленных на его разрушение.

использованы ООО Дилер в мероприятиях по выработке перспективных подходов для вложения цифрового водяного знака в исполняемые файлы программного обеспечения, предназначенного для тестирования телекоммуникационного оборудования.

Комиссия отмечает практическую значимость и новизну полученных в работе результатов.

Генеральный директор ООО " ДИЛЕР" _____ Дейкун Д.А.