Научная статья
DOI 10.66424/2071-8217-2026-2-6
УДК 004.056

ДЕОБФУСКАЦИЯ ВРЕДОНОСНОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ С ИСПОЛЬЗОВАНИЕМ ПРОМЕЖУТОЧНОГО ПРЕДСТАВЛЕНИЯ LLVM

Н. А. Милютин, Т. Д. Овасапян*, Д. В. Иванов

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

otd@ibks.spbstu.ru

ДЛЯ ЦИТИРОВАНИЯ

Милютин Н. А., Овасапян Т. Д., Иванов Д. В. Деобфускация вредоносного программного обеспечения с использованием промежуточного представления LLVM // Проблемы информационной безопасности. Компьютерные системы. 2026. № 2. С. 70–81. DOI: 10.66424/2071-8217-2026-2-6

ПОСТУПИЛА 05.03.2026
ПРИНЯТА 05.05.2026
ОПУБЛИКОВАНА 15.06.2026

© Милютин Н. А., Овасапян Т. Д., Иванов Д. В.

Издатель: Санкт-Петербургский политехнический университет Петра Великого

АННОТАЦИЯ

Рассматривается задача автоматизации деобфускации вредоносного программного обеспечения. Предложен метод, основанный на промежуточном представлении LLVM, объединяющий динамическую распаковку с трассировкой, гибридное (trace-assisted) восстановление графа потока управления и итеративную девиртуализацию. Разработан программный прототип, реализующий предложенный метод. Проведена экспериментальная оценка, подтвердившая применимость метода к снятию обфускации классов: упаковка, искажение потока управления, обфускация инструкций и виртуализация кода.

КЛЮЧЕВЫЕ СЛОВА

Обфускация, деобфускация, LLVM IR, девиртуализация, распаковка, восстановление графа потока управления

Original article
DOI 10.66424/2071-8217-2026-2-6

DEOBFUSCATION OF MALICIOUS SOFTWARE USING LLVM INTERMEDIATE REPRESENTATION

N. A. Milyutin, T. D. Ovasapyan*, D. V. Ivanov

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

otd@ibks.spbstu.ru

Статьи распространяются в соответствии с лицензией CC BY-NC 4.0

FOR CITATION
Milyutin N. A., Ovasapyan T. D., Ivanov D. V. Deobfuscation of malicious software using LLVM intermediate representation. *Problems of information security. Computer systems*. 2026. No. 2, pp. 70–81. DOI: 10.66424/2071-8217-2026-2-6 (In Russian)

RECEIVED 05.03.2026
ACCEPTED 05.05.2026
PUBLICATION 15.06.2026

ABSTRACT

The problem of automating deobfuscation of malicious software is considered. A method based on the LLVM intermediate representation is proposed that combines dynamic unpacking with tracing, hybrid (trace-assisted) restoration of the control flow graph and iterative devirtualization. A software prototype has been developed that implements the proposed method. An experimental evaluation was carried out, confirming the applicability of the approach to removing class obfuscation: packaging, control flow distortion, instruction obfuscation, and code virtualization.

KEYWORDS

Obfuscation, deobfuscation, LLVM IR, devirtualization, unpacking, control flow graph recovery

1. ВВЕДЕНИЕ

Одним из способов сокрытия логики работы программ является обфускация кода – процесс преднамеренного усложнения структуры исходного кода или исполняемого файла для затруднения анализа. Первоначально обфускация применялась для защиты интеллектуальной собственности, но в текущих реалиях широко используется во вредоносном программном обеспечении [1].

Разработка методов и инструментов, позволяющих автоматически проводить деобфускацию – восстановление исходной или приближенной к исходной структуре программ, – приобретает особую значимость, поскольку позволяет ускорить реверс-инжиниринг при анализе потенциально вредоносных объектов.

В мировой и отечественной практике представлено множество инструментов для статического и динамического анализа программ, однако автоматизация процесса деобфускации остается недостаточно решенной задачей. Существующие инструменты обычно требуют глубоких экспертных знаний, ручной настройки и не обеспечивают высокой степени восстановления кода. Более того, из-за разнообразия техник обфускации (изменение графа потока управления, вставка ложных операций, шифрование строк, виртуализация кода), универсального решения, охватывающего широкий класс случаев, до сих пор не существует.

Целью работы является автоматизация деобфускации вредоносного программного обеспечения за счет его анализа на уровне промежуточного представления LLVM.

2. СОВРЕМЕННЫЕ РЕШЕНИЯ

Классификация техник обфускации. Техники обфускации, применяемые во вредоносном ПО, классифицируются по четырем основным категориям [2]. Обфускация данных направлена на предотвращение извлечения критически важной информации методами статического анализа. Данные преобразуются в нечитаемую форму посредством шифрования строк, констант, разбиения данных с поздней инициализацией, использования вычислимых констант. Восстановление исходных значений происходит только в момент использования данных при работе программы.

Искажение графа потока управления направлено на увеличение цикломатической сложности программы, вследствие чего декомпиляторы не могут корректно восстановить структуру кода. Ключевой техникой является control-flow flattening [3] – преобразование иерархической структуры кода в «плоскую» структуру, где базовые блоки помещаются внутрь цикла с оператором switch, а порядок исполнения определяется переменной состояния.

Статьи распространяются в соответствии с лицензией CC BY-NC 4.0

К данному классу также относятся: инъекция ложных путей (opaque predicates), встраивание функций (inlining) и вынесение фрагментов в отдельные функции (outlining).

Упаковка и маскировка входной точки направлена на сокрытие исполняемого файла до момента исполнения [4]. Файл разделяется на полезную нагрузку (сжатую или зашифрованную) и распаковщик. Современные образцы ВПО применяют многоуровневую упаковку (multi-layer packing [5]), при которой каждый слой выполняет подготовку следующего слоя. Задача автоматической распаковки сводится к корректному воспроизведению всей цепочки зависимых исполнений.

Преобразование логики на уровне инструкций включает подстановку функционально эквивалентных инструкций, вставку мусорного кода и самомодифицирующийся код. Наиболее мощной техникой является виртуализация кода [6–8] при которой исходный машинный код трансформируется в байт-код пользовательской архитектуры, исполняемый встроенной виртуальной машиной (интерпретатором). Виртуальная машина содержит набор виртуальных команд, транслятор и интерпретатор, реализующий цикл выборки-декодирования-исполнения.

Методы распаковки. Ключевым этапом анализа упакованного ВПО является детектирование момента восстановления оригинальной точки входа. Выделяются следующие методы:

• Write-then-Execute (WxE) – эвристика, отслеживающая пары событий «запись в память – исполнение». Реализуется на двух уровнях гранулярности: побайтный мониторинг (Renovo [9], Ether [10]) и постраничный (OmniUnpack [11], GeMU [12]). Метод инвариантен к алгоритмам упаковки.

• Эвристики на основе энтропии: уменьшение энтропии участка памяти может служить индикатором завершения распаковки, однако возможны ложные срабатывания.

• Эвристики на основе API-паттернов: обнаружение последовательностей вызовов VirtualAlloc, WriteProcessMemory и подобных. Ненадежны, поскольку ВПО может скрывать импорт библиотек.

Среди рассмотренных решений наиболее полным является распаковщик GeMU, обеспечивающий полное послойное извлечение (Layer-by-Layer) и поддержку многопроцессности.

Методы восстановления графа потока управления. Граф потока управления (CFG) задает структуру программы: базовые блоки и переходы между ними. Обфускация, искажающая CFG (control-flow flattening, инъекция ложных путей), разрушает эту структуру, вследствие чего декомпиляторы выдают нечитаемый «спагетти-код», а чисто статический анализ не может однозначно определить цели косвенных переходов. Восстановление CFG – ключевой этап деобфускации, без которого невозможен последующий анализ семантики кода. Представим современные методы, решающие данную задачу:

• Статический анализ с абстрактной интерпретацией (Jakstab) [13] – метод итеративного дизассемблирования, использующий Value Set Analysis для разрешения косвенных переходов. Ограничен архитектурой x86 и подвержен проблеме потери точности (over-approximation).

• Посимвольное выполнение (Symbolic Execution) – метод, используемый в средствах KLEE, Angr, Triton, оперирующий символьными переменными с использованием SMT-решателя (Z3). Обеспечивает высокую точность, однако подвержен проблеме экспоненциального роста путей (path explosion).

• Итеративный лифтинг LLVM (SATURN) [14] – метод, объединяющий поднятие бинарного кода в LLVM IR и использование оптимизаций компилятора. Алгоритм выполняет трансляцию инструкций в LLVM IR, применяет стандартные оптимизации и SMT-оптимизатор Souper [15], разрешает переходы с помощью SMT-решателя и итеративно расширяет фронт обнаруженных адресов. Преимуществом является использование готовой инфраструктуры LLVM для автоматического устранения обфускации на этапе построения графа. В качестве недостатка можно отметить ограниченность статического подхода в обработке самомодифицирующегося кода.

Статьи распространяются в соответствии с лицензией CC BY-NC 4.0

Методы деобфускации виртуализированного кода. Виртуализация кода относится к числу наиболее устойчивых техник обфускации: нативный код подменяется байт-кодом авторской архитектуры, исполняемым встроенным интерпретатором. В результате скрываются и поток управления, и поток данных. Аналитику приходится сначала реконструировать саму виртуальную машину, чтобы понять ее систему команд и структуру диспетчера. Девиртуализация сводится к двум задачам – анализу структуры виртуальной машины (ВМ) и переводу ее байт-кода обратно в нативный код или промежуточное представление. Рассмотрим основные методы:

- программный синтез (Syntia): обфусцированный блок рассматривается как черный ящик, для которого синтезируется эквивалентная программа (инвариантен к сложности обфускации, но ограничен масштабируемостью);
- статический паттерн-матчинг: поиск известных сигнатур ВМ (быстр, но неустойчив к полиморфизму обфускатора);
- символьное выполнение с анализом трассы [16, 17]: запись трассы, taint-анализ, посимвольное исполнение обработчиков ВМ (позволяет точно понять семантику обработчиков, но зависит от длины трассы).

3. СРАВНИТЕЛЬНЫЙ АНАЛИЗ МЕТОДОВ И ИНСТРУМЕНТОВ

Результаты сравнительного анализа распаковщиков представлены в табл. 1. GeMU в силу своих возможностей выбран в качестве распаковщика для дальнейшей деобфускации. В табл. 2 представлено сравнение методов восстановления графа потока управления.

Наиболее перспективным для автоматизации деобфускации является метод, используемый SATURN: промежуточное представление LLVM IR унифицирует анализ для разных архитектур и позволяет применять оптимизации компиляторов.

Таблица 1 | Сравнение распаковщиков

Table 1 | Comparison of unpackers

Инструмент	Метод обнаружения	Извлечение слоев	Поддержка многопроцессности	Основные ограничения
GeMU	WxE	Полное (Layer-by-Layer)	Да	Требует ресурсов эмуляции (QEMU)
Renovo/Ether	WxE	Частичное	Да	Устаревшие реализации, высокие накладные расходы
OmniUnpack	WxE + внешние эвристики	Нет	Ограничена	Зависимость от внешних детекторов

Таблица 2 | Сравнение методов к восстановлению графа потока управления

Table 2 | Comparison of approaches to control flow graph reconstruction

Метод	Инструменты	Точность	Скорость
Абстрактная интерпретация	Jakstab	Средняя (false-positives)	Средняя
Посимвольное исполнение	KLEE, Triton, Angr	Высокая	Низкая (path explosion)
Итеративный лифтинг LLVM	SATURN	Высокая (SMT + оптимизации)	Средняя

Статьи распространяются в соответствии с лицензией CC BY-NC 4.0

Однако чисто статический подход не способен обработать код, формируемый во время выполнения.

На основании проведенного анализа определен комбинированный метод:

• использование LLVM-лифтинга для перевода кода в унифицированное представление;

• расширение статического анализа динамической трассой для преодоления ограничений чисто статических методов;

• применение SMT-решателей для очистки локальных обфускаций (MBA).

Ни один из рассмотренных инструментов не обеспечивает одновременно распаковку, восстановление CFG, деобфускацию инструкций и девиртуализацию в рамках единого конвейера.

4. ОПИСАНИЕ ПРЕДЛАГАЕМОГО МЕТОДА

Предлагаемый метод автоматизации деобфускации основывается на промежуточном представлении LLVM и состоит из трех последовательных этапов.

Этап 1: распаковка и выборочная трассировка. На начальном этапе проводится динамическая распаковка кода с целью получения чистого бинарного образа для дальнейшего лифтинга. Используется метод на основе инструмента GeMU [12], реализующий принцип Write-then-Execute.

Работа этапа:

1. Эмуляция среды исполнения: образец запускается в изолированной эмулируемой среде (QEMU), что скрывает факт анализа от средств антиотладки.

2. Мониторинг страниц памяти: при записи на страницу она помечается как кандидат (dirty); при исполнении инструкций из такой страницы срабатывает триггер детектора.

3. Итеративное извлечение слоев: при каждом обнаружении нового слоя создается дамп процесса. После этого страница сбрасывает статус кандидата и эмуляция продолжается.

Дополнительно фиксируется селективная трасса исполнения, включающая:

• поток управления: адрес инструкции, адреса переходов, исходы условных ветвлений;

• контекст вычислений: значения регистров на границах базовых блоков;

• события памяти: чтения и записи, влияющие на вычисление целей переходов;

• карту адресного пространства: регионы base/size/prot с обновлениями при VirtualAlloc/VirtualProtect.

Необходимость трассы обусловлена тем, что значимая часть логики обфусцированной программы реализуется через косвенные переходы, вычисляемые во время выполнения, а также через динамически вычисляемые константы, которые не выводятся из статического контекста.

Результатом этапа является набор дампов памяти, содержащих распакованные фрагменты кода, и синхронизированная трасса исполнения.

Этап 2: лифтинг и восстановление графа потока управления. На данном этапе реализуется основной алгоритм деобфускации, объединяющий восстановление CFG и очистку кода от обфусцированных инструкций в едином цикле анализа. Это принципиально важно, поскольку точное определение адресов переходов часто невозможно без предварительного упрощения арифметики.

С учетом наличия трассы этап расширяется до гибридного (trace-assisted) восстановления CFG: статический анализ используется как базовый механизм, а трасса применяется как источник фактических целей при неоднозначности.

Алгоритм работы:

1. Динамическое расширение фронта: анализ начинается с известной точки входа; адреса новых блоков извлекаются из очереди и поднимаются в LLVM IR.

2. Оптимизации: к каждому поднятому блоку применяются стандартные оптимизации LLVM (Constant Propagation, DCE) и оптимизатор Souper (SMT-решатель Z3, символьное выполнение KLEE). При наличии записей трассы добавляются

Статьи распространяются в соответствии с лицензией CC BY-NC 4.0

ограничения (assumptions), фиксирующие конкретные значения регистров.

3. Отсечение невыполнимых ветвей: SMT-решатель проверяет выполнимость условий; при доказанной тривиальности ветвления граф упрощается.

4. Trace-assisted разрешение переходов: для прямых переходов цели определяются статически; для косвенных – при неоднозначности используется трасса для получения фактической цели с добавлением ограничения.

Цикл продолжается до исчерпания очереди блоков. Результатом является восстановленный модуль LLVM IR и CFG, корректный для наблюдаемого исполнения.

Этап 3: девиртуализация. Девиртуализация трактуется как процедура нормализации и редукции восстановленного IR, направленная на устранение артефактов виртуальной машины.

Для описания этапа необходимо ввести следующие определения: псевдопамять – абстрактный объект памяти в IR, от базового указателя которого вычисляются адреса обращений; псевдостек – часть псевдопамяти, рассматриваемая как стек по правилу классификации адресов.

Трасса на данном этапе используется для фиксации значений, определяющих выбор обработчика, конкретизации чтений из псевдопамяти, направленной развертки цикла диспетчеризации.

Алгоритм конвейера:

1. Инициализация: регистрация анализа через метод llvm:: PassBuilder [18].

2. Специализированная нормализация (trace-assisted):

• подстановка значений для выражений адресов и условий;

• constant propagation для чтений по детерминированным адресам;

• нормализация чтений и устранение избыточных преобразований типов;

• отделение стековой области от общей псевдопамяти;

• канонизация GEP/inttoptr и снижение ложных зависимостей.

3. Финальная оптимизация: применяется стандартный оптимизационный набор LLVM уровня O2.

5. РАЗРАБОТКА ПРОТОТИПА ПРОГРАММНОГО СРЕДСТВА АВТОМАТИЗАЦИИ ДЕОБФУСКАЦИИ

Разработанный прототип состоит из пяти компонентов (см. рисунок):

1. Распаковщик – модифицированный проект GeMU, обеспечивающий извлечение распакованных слоев (WxE) и сбор селективной трассы исполнения.

2. Дизассемблер – IDA Pro с IDAPython-скриптом для извлечения листинга по заданному адресу.

3. Модуль трансляции – модифицированный mcsema_lift, выполняющий лифтинг в LLVM IR с одновременным сопоставлением событий трассы.

4. Модуль восстановления CFG – реализует worklist-обход и построение графа.

5. Модуль девиртуализации – выполняет специализированную нормализацию и развертку цикла диспетчера.

Распаковщик построен как модифицированный GeMU с двумя целями: устойчивое извлечение полезной нагрузки при многоуровневой распаковке и формирование селективной трассы.

Архитектура включает: среду исполнения (QEMU), монитор записи в память, монитор исполнения, менеджер слоев/дампов и модуль трассировки (Trace Recorder – добавлен в рамках модификации).

Трассировка фиксирует три типа событий:

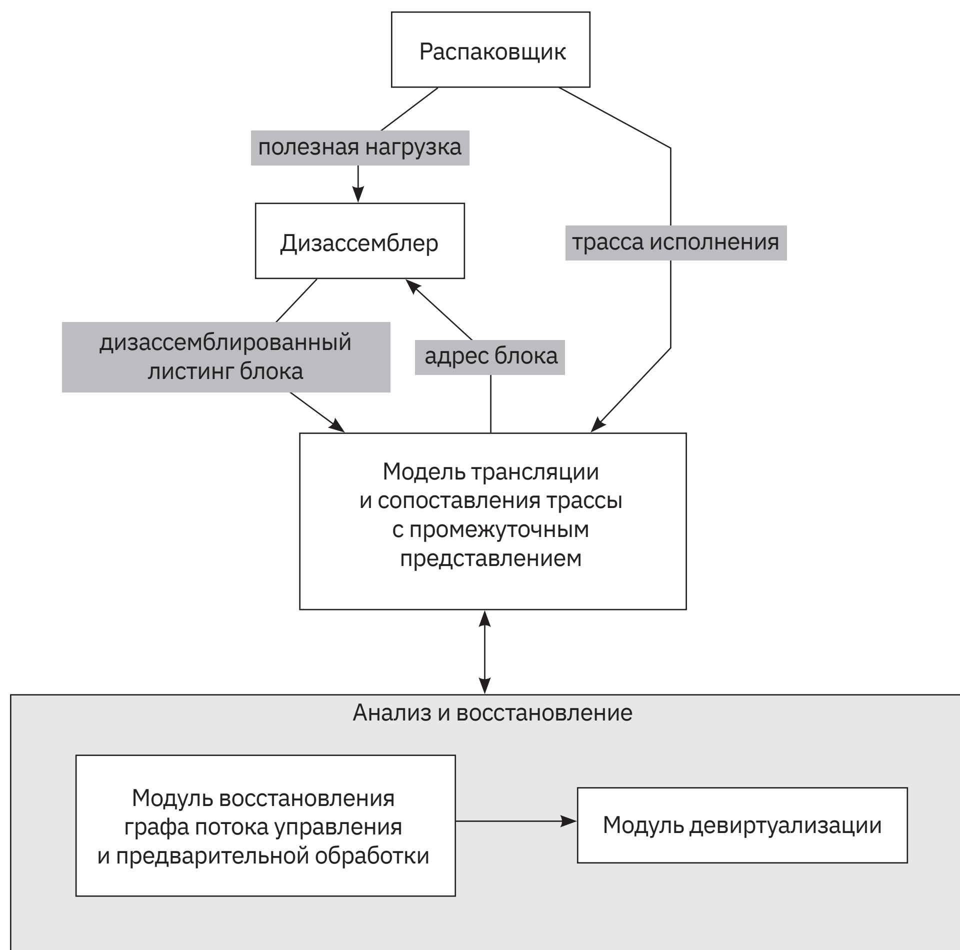
1. Control-flow события: rip_before, kind (Jcc/JMP_indirect/CALL_indirect/RET), target_rip, taken.

2. Memory-access события: rip_before, kind (load/store), ea, size, value;

3. Карта памяти: регионы base/size/prot с обновлениями.

Модуль трансляции. Переход от бинарного слоя к LLVM IR организован как связка IDA → IDAPython – скрипт → модифицированный mcsema_lift [19]. Модуль трансляции одновременно выполняет лифтинг и сопоставляет события трассы с адресами инструкций. Для ускорения запросов к трассе вводится индекс TraceIndex, содержащий: observedTargets,

Статьи распространяются в соответствии с лицензией CC BY-NC 4.0



Схематическое представление прототипа

Schematic representation of the prototype

observedTaken, fallthroughRIP (control-flow), observedMemOps (memory-access) и memoryMaps (адресные пространства).

Инъекция динамических сведений разделяется на два механизма: Trace-assisted assumptions для условного перехода – фиксация наблюдаемого исхода через llvm.assume; Trace-assisted load concretization при наличии наблюдаемых memory-операций – подстановка константного значения вместо неопределенного чтения.

Для косвенных переходов наблюдаемые цели используются для развертки управления в явные ребра CFG.

Модуль девиртуализации. Модуль решает задачу развертки цикла диспетчера. После восстановления CFG косвенные переходы заменяются на switch или каскад условных переходов, но интерпретаторная модель сохраняется.

Анализ трассы и восстановление последовательности обработчиков: определяется базовый блок диспетчера (наибольшее количество control-flow событий), затем линейным проходом по трассе строится последовательность пар (virtual instruction pointer, handler).

Инъекция виртуального указателя: в тело цикла добавляется llvm.assume с фиксацией значения VIP (virtual instruction pointer), что делает диспетчеризацию детерминированной. Последующие оптимизации (Loop Unrolling, Constant Propagation, DCE) автоматически разворачивают и упрощают цикл.

Реализованы специализированные проходы оптимизации:

1. Constant propagation для чтений по константным адресам: замена чтений на константы при выполнении условий политики работы с памятью. Пример:

Статьи распространяются в соответствии с лицензией CC BY-NC 4.0

```
; До: чтение по детерминированному адресу управляет ветвлением
%p = getelementptr i8, ptr %memory, i64 5368725620
%flag = load i32, ptr %p
%is_zero = icmp eq i32 %flag, 0
br i1 %is_zero, label %fast, label %slow

; После constant propagation + SimplifyCFG + DCE:
%a = add i32 %x, 1
ret i32 %a
```

2. Сборка значений из сегментов: при чтении диапазона байтов строится разбиение на ref-сегменты (из предыдущих store) и mem-сегменты, итоговое значение собирается через операции извлечения байтов и битовых сдвигов.

3. Упрощение адресации: замена адресации от общей базы (GEP + offset) на прямое представление (inttoptr), что снижает число ложных зависимостей по данным. Пример:

```
; До:
%p = getelementptr i8, ptr %memory, i64 5368725620
; После:
%p = inttoptr i64 5368725620 to ptr
```

Тестирование. Для оценки разработан ряд тестовых наборов:

• dataset A (юнит-тесты): арифметика (работа с памятью), ветвления по флагам, обращение к памяти, косвенные переходы – 18 тестов;

• dataset B (виртуализация/контроль потока): бинарные файлы, собранные с Tigress Virtualize [20] и коммерческими ВМ-защитами (Themida/VMProtect);

• dataset C (упакованные семплы): упакованные PE файлы с overlay, формируемым модифицированным GeMU.

Для Dataset B верификация выполнялась сравнением с оригинальным ПО либо проверкой эквивалентности по инвариантам на фиксированном наборе входов. Для Dataset C фиксировались структурные метрики: стабилизация CFG, исчезновение VM-диспетчерного цикла, уменьшение доли псевдопамяти, рост константности адресов.

Результаты подтверждают, что предложенный конвейер обеспечивает практическую применимость для широкого класса задач деобфускации: лифтинг в LLVM IR

Таблица 3 | Сравнение реализованного прототипа с фреймворком Saturn

Table 3 | Comparison of the implemented software with the Saturn framework

Класс техники	SATURN	Разработанный прототип
Константные/арифметические маски (MBA)	+	+
Opaque predicates	+	+
Dead code insertion	+	+
Bogus control flow	+	+
Integer encoding/flattening по данным	+	+
Упакованные семплы (overlay)	–	+
Девиртуализация (устранение ВМ-диспетчеризации)	–	+

Статьи распространяются в соответствии с лицензией CC BY-NC 4.0

Таблица 4 | Результаты тестирования

Table 4 | Test results

Датасет	**Программы**	**Проверка результата**	**Снятие обфускации**
A	18 юнит-тестов (bench_add, test_branch_*, test_memory и др.)	Юнит-тест	+ (все тесты пройдены)
B	tigress_virtualize_min.exe, themida_vm_sample.exe	Сравнение потоков управления	+ (для выбранной трассы)
C	upx_packed_sample.exe, packed_vm_sample.exe	Структурные метрики	Частично

с восстановлением CFG и итеративная нормализация обеспечивают заметное снижение структурного и инструкционного шума.

6. ЗАКЛЮЧЕНИЕ

В ходе работы рассмотрены и систематизированы основные классы обфускации, существенные для анализа бинарного кода: упаковка (включая многоуровневую распаковку), обфускация графа потока управления, обфускация данных и инструкций, а также виртуализация кода. Проведен сравнительный анализ существующих методов и инструментов деобфускации.

На основе анализа предложен метод автоматизации деобфускации, основанный на промежуточном представлении LLVM и объединяющий три этапа:

• динамическая распаковка с извлечением слоев по инварианту WxE и формирование селективной трассы исполнения;

• лифтинг с гибридным (trace-assisted) восстановлением графа потока управления, где наблюдаемые цели переходов и операции с памятью используются как ограничения для стабилизации анализа;

• девиртуализация как итеративная редукция LLVM IR стандартными и специализированными проходами оптимизаций, дополненная SMT-ориентированным упрощением выражений.

Разработан программный прототип, реализующий предложенный метод. В сравнении с ближайшим аналогом (SATURN) прототип дополнительно обеспечивает: поддержку упакованных семплов за счет overlay и девиртуализацию с устранением ВМ-диспетчеризации.

Тестирование показало, что прототип обеспечивает автоматизацию ключевых этапов восстановления семантики обфусцированного кода: восстановление графа потока управления, упрощение базовых блоков и управляющих условий, девиртуализацию, корректную работу с распакованными слоями. Ограничения метода определяются качеством трассы и точностью классификации адресных диапазонов, влияющими на полноту конкретизации памяти и степень редукции IR.

КОНФЛИКТ ИНТЕРЕСОВ / CONFLICT OF INTERESTS

Авторы заявляют об отсутствии конфликта интересов / The authors declare no conflict of interests.

Статьи распространяются в соответствии с лицензией CC BY-NC 4.0

СПИСОК ИСТОЧНИКОВ

1. Отчет лаборатории Касперского за второй квартал 2024. URL: https://securelist.ru/it-threat-evolution-q2-2024-pc-statistics/110425/ (дата обращения: 21.02.2026).
2. **Collberg C. S., Thomborson C.** Watermarking, tamper-proofing, and obfuscation tools for software protection // IEEE Transactions on Software Engineering. 2002. Vol. 28 (8). P. 735–746. DOI: 10.1109/TSE.2002.1027797.
3. **Laszlo T., Kiss A.** Obfuscating C++ programs via control flow flattening // Annales Universitatis Scientarum Budapestinensis. 2009. Vol. 30. № 1. P. 3–19.
4. **Jenke T., Liessem S., Padilla E., Bruckschen L.** A Measurement Study on Interprocess Code Propagation of Malicious Software // ICDF2C 2023. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. 2023. Vol 571. P. 264–282.
5. **Behera C., Bhaskari D. L.** Different Obfuscation Techniques for Code Protection // Procedia Computer Science. 2015. Vol. 70. P. 757–763. DOI: 10.1016/j.procs.2015.10.114.
6. **Kochberger P., Schrittwieser S., Schweighofer S. et al.** SoK: Automatic Deobfuscation of Virtualization-protected Applications // ARES 2021: The 16th International Conference on Availability, Reliability and Security. 2021. P. 1–15. DOI: 10.1145/3465481.3465772.
7. **Xiao X., Wang Y., Hu Yi., Gu D.** xvmp: An llvm-based code virtualization obfuscator // IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). 2023. P. 738–742. DOI: 10.1109/SANER56733.2023.00082.
8. **Junod P., Rinaldini J., Wehrli J., Michielin J.** Obfuscator LLVM Software Protection for the Masses // IEEE/ACM 1st International Workshop on Software Protection (SPRO). 2015. DOI: 10.1109/SPRO.2015.10.
9. **Kang M. G., Poosankam P., Yin H.** Renovo: A hidden code extractor for packed executables // Proceedings of the 2007 ACM workshop on Recurring Malcode. 2007. P. 46–53. DOI: 10.1145/1314389.1314399
10. **Dinaburg A., Royal P., Sharif M. I., Lee W.** Ether: malware analysis via hardware virtualization extensions // Proceedings of the 2008 ACM Conference on Computer and Communications Security (CCS 2008), 27–31 October 2008, Alexandria, Virginia, USA. 2008. P. 51–62. DOI: 10.1145/1455770.1455779
11. **Martignoni L., Christodorescu M., Jha S.** OmniUnpack: Fast, Generic, and Safe Unpacking of Malware // ACSAC. 2007. P. 431–441. DOI: 10.1109/ACSAC.2007.15
12. GeMU, the generic malware unpacker based on QEMU. URL: https://github.com/fkie-cad/GeMU (дата обращения: 21.02.2026).
13. **Kinder J.** Static analysis of x86 executables. Darmstadt: Technische Universität, 2010. P. 156–178.
14. **Garba P., Favaro M.** Software Deobfuscation Framework Based on LLVM // Proceedings of the 3rd ACM Workshop on Software Protection. 2019. P. 27–38.
15. Souper (superoptimizer for LLVM IR). URL: https://github.com/google/souper (дата обращения: 21.02.2026).
16. **Coogan K., Lu G., Debray S.** Deobfuscation of Virtualization-Obfuscated Software: A Semantics-Based Approach // Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS 2011), 17–21 October 2011, Chicago, Illinois, USA. 2011. P. 275–284. DOI: 10.1145/2046707.2046739.
17. **Ovasapyan T. D., Knyazev P. V., Moskvin D. A.** Application of taint analysis to study the safety of software of the Internet of Things devices based on the ARM architecture // Automatic Control and Computer Sciences. 2020. Vol. 54. № 8. P. 834–840
18. McSema. URL: https://github.com/lifting-bits/mcsema (дата обращения: 21.02.2026).
19. LLVM's Analysis and Transform Passes. URL: https://github.com/llvm/llvm-project/blob/main/llvm/docs/Passes.rst (дата обращения: 21.02.2026).
20. Tigress C obfuscator. URL: https://tigress.wtf/ (дата обращения: 21.02.2026).

REFERENCES

1. Kaspersky Lab report for the second quarter of 2024. URL: https://securelist.ru/it-threat-evolution-q2-2024-pc-statistics/110425/ (accessed: 21.02.2026). (In Russian)

Статьи распространяются в соответствии с лицензией CC BY-NC 4.0

2. **Collberg C. S., Thomborson C.** Watermarking, tamper-proofing, and obfuscation tools for software protection. *IEEE Transactions on Software Engineering*. 2002. Vol. 28 (8), pp. 735–746. DOI: 10.1109/TSE.2002.1027797.
3. **Laszlo T., Kiss A.** Obfuscating C++ programs via control flow flattening. *Annales Universitatis Scientarum Budapestinensis*. 2009. Vol. 30. No. 1, pp. 3–19.
4. **Jenke T., Liessem S., Padilla E., Bruckschen L.** A Measurement Study on Interprocess Code Propagation of Malicious Software. ICDF2C 2023. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. 2023. Vol 571, pp. 264–282.
5. **Behera C., Bhaskari D. L.** Different Obfuscation Techniques for Code Protection. Procedia Computer Science. 2015. Vol. 70, pp. 757–763. DOI: 10.1016/j.procs.2015.10.114.
6. **Kochberger P., Schrittwieser S., Schweighofer S. et al.** SoK: Automatic Deobfuscation of Virtualization-protected Applications. ARES 2021: The 16th International Conference on Availability, Reliability and Security. 2021, pp. 1–15. DOI: 10.1145/3465481.3465772.
7. **Xiao X., Wang Y., Hu Yi., Gu D.** xvmp: An llvm-based code virtualization obfuscator. IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). 2023, pp. 738–742. DOI: 10.1109/SANER56733.2023.00082.
8. **Junod P., Rinaldini J., Wehrli J., Michielin J.** Obfuscator LLVM Software Protection for the Masses. IEEE/ACM 1st International Workshop on Software Protection (SPRO). 2015. DOI: 10.1109/SPRO.2015.10.
9. **Kang M. G., Poosankam P., Yin H.** Renovo: A hidden code extractor for packed executables. Proceedings of the 2007 ACM workshop on Recurring Malcode. 2007, pp. 46–53. DOI: 10.1145/1314389.1314399
10. **Dinaburg A., Royal P., Sharif M. I., Lee W.** Ether: malware analysis via hardware virtualization extensions. Proceedings of the 2008 ACM Conference on Computer and Communications Security (CCS 2008), 27–31 October 2008, Alexandria, Virginia, USA. 2008, pp. 51–62. DOI: 10.1145/1455770.1455779
11. **Martignoni L., Christodorescu M., Jha S.** OmniUnpack: Fast, Generic, and Safe Unpacking of Malware. ACSAC. 2007, pp. 431–441. DOI: 10.1109/ACSAC.2007.15
12. GeMU, the generic malware unpacker based on QEMU. URL: https://github.com/fkie-cad/GeMU (accessed: 21.02.2026).
13. **Kinder J.** Static analysis of x86 executables. Darmstadt: Technische Universität, 2010, pp. 156–178.
14. **Garba P., Favaro M.** Software Deobfuscation Framework Based on LLVM. Proceedings of the 3rd ACM Workshop on Software Protection. 2019, pp. 27–38.
15. Souper (superoptimizer for LLVM IR). URL: https://github.com/google/souper (accessed: 21.02.2026).
16. **Coogan K., Lu G., Debray S.** Deobfuscation of Virtualization-Obfuscated Software: A Semantics-Based Approach. Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS 2011), 17–21 October 2011, Chicago, Illinois, USA. 2011, pp. 275–284. DOI: 10.1145/2046707.2046739.
17. **Ovasapyan T. D., Knyazev P. V., Moskvin D. A.** Application of taint analysis to study the safety of software of the Internet of Things devices based on the ARM architecture. *Automatic Control and Computer Sciences*. 2020. Vol. 54. No. 8, pp. 834–840
18. McSema. URL: https://github.com/lifting-bits/mcsema (accessed: 21.02.2026).
19. LLVM's Analysis and Transform Passes. URL: https://github.com/llvm/llvm-project/blob/main/llvm/docs/Passes.rst (accessed: 21.02.2026).
20. Tigress C obfuscator. URL: https://tigress.wtf/ (accessed: 21.02.2026).

СВЕДЕНИЯ ОБ АВТОРАХ / INFORMATION ABOUT AUTHORS

МИЛЮТИН Никита Алексеевич – студент, Санкт-Петербургский политехнический университет Петра Великого, Россия, 195251, Санкт-Петербург, ул. Политехническая, д. 29
E-mail: milyutin.na@edu.spbstu.ru
ORCID: 0009-0002-7398-5069

MILYUTIN Nikita A. – Student, Peter the Great St. Petersburg Polytechnic University, Russia, 195251, St. Petersburg, Polytechnicheskaya str., 29

Статьи распространяются в соответствии с лицензией CC BY-NC 4.0

ОВАСАПЯН Тигран Джаникович – канд. техн. наук, доцент, Санкт-Петербургский политехнический университет Петра Великого, Россия, 195251, Санкт-Петербург, ул. Политехническая, д. 29
E-mail: otd@ibks.spbstu.ru
ORCID: 0000-0002-2009-5460

OVASAPYAN Tigran D. – Candidate of Engineering Sciences, Associate Professor, Peter the Great St. Petersburg Polytechnic University, Russia, 195251, St. Petersburg, Polytechnicheskaya str., 29

ИВАНОВ Денис Вадимович – канд. техн. наук, доцент, Санкт-Петербургский политехнический университет Петра Великого, Россия, 195251, Санкт-Петербург, ул. Политехническая, д. 29
E-mail: ivanov@ibks.spbstu.ru
ORCID: 0000-0001-8206-2915

IVANOV Denis V. – Candidate of Engineering Sciences, Associate Professor, Peter the Great St. Petersburg Polytechnic University, Russia, 195251, St. Petersburg, Polytechnicheskaya str., 29

Статьи распространяются в соответствии с лицензией CC BY-NC 4.0