А. В. Вишняков, И. А. Кобрин, А. Н. Федотов, Поиск ошибок в бинарном коде методами динамической символьной интерпретации,
Труды ИСП РАН, 2022, том 34, выпуск 2, 25–42

DOI: 10.15514/ISPRAS-2022-34(2)-3

Поиск ошибок в бинарном коде методами динамической символьной интерпретации

[1]*А.В. Вишняков, ORCID: 0000-0003-1819-220X <vishnya@ispras.ru>*
[1,2]*И.А. Кобрин, ORCID: 0000-0002-6035-0577 <kobrineli@ispras.ru>*
[1]*А.Н. Федотов, ORCID: 0000-0002-8838-471X <fedotoff@ispras.ru>*

[1]*Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25*
[2]*Московский государственный университет имени М.В. Ломоносова, 119991, Россия, Москва, Ленинские горы, д. 1.*

Аннотация. Современное программное обеспечение стремительно развивается, принося новые ошибки, и все больше компаний следуют безопасному циклу разработки ПО. Одними из самых популярных средств для поддержки безопасного цикла разработки являются фаззинг и символьная интерпретация программ, позволяющие автоматически тестировать программу и искать в ней ошибки. Гибридный фаззинг – наиболее эффективный подход, который заключается в применении комбинации этих двух техник, при котором две техники работают совместно. Другим способом искать программные ошибки является символьная интерпретация с использованием предикатов безопасности – условий на входные данные, при выполнении которых будет проявлена ошибка. В этой работе мы предлагаем метод автоматизированного поиска ошибок с помощью динамической символьной интерпретации, совмещающий гибридный фаззинг с проверкой предикатов безопасности. Гибридный фаззинг требуется для получения большого количества различных входных данных, а ошибки работы с памятью и неопределенного поведения в программах ищут предикаты безопасности, которые позволяют находить ошибки деления на нуль, выхода за границы массива, целочисленного переполнения и другие. Результаты работы предикатов безопасности верифицируются с помощью санитайзеров, чтобы отбросить ложно положительные срабатывания. В результате практического применения предложенного метода к программам с открытым исходным кодом было найдено 11 различных новых ошибок в 5 разных проектах.

Ключевые слова: динамическая символьная интерпретация; DSE; фаззинг; предикат безопасности; автоматическое обнаружение ошибок; безопасный цикл разработки; SDL; бинарный код; санитайзер; ошибка; CWE

Для цитирования: Вишняков А.В., Кобрин И.А., Федотов А.Н. Поиск ошибок в бинарном коде методами динамической символьной интерпретации. Труды ИСП РАН, том 34, вып. 2, 2022 г., стр. 25-42. DOI: 10.15514/ISPRAS–2022–34(2)–3

Благодарности. Работа поддержана грантом РФФИ № 20-07-00921 А**.**

Error detection in binary code with dynamic symbolic execution

[1]*A.V. Vishnyakov, ORCID: 0000-0003-1819-220X <vishnya@ispras.ru>*
[1,2]*E.A. Kobrin, ORCID: 0000-0002-6035-0577 <kobrineli@ispras.ru>*
[1]*A.N. Fedotov, ORCID: 0000-0002-8838-471X <fedotoff@ispras.ru>*

[1]*Ivannikov Institute for System Programming of the RAS, 25, Alexander Solzhenitsyn Str., Moscow, 109004, Russia*
[2]*Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russian Federation*

Abstract. Modern software is rapidly developing, revealing new program errors. More and more companies follow security development lifecycle (SDL). Fuzzing and symbolic execution are among the most popular options for supporting SDL. They allow to automatically test programs and find errors. Hybrid fuzzing is one of the most effective ways to test programs, which combines these two techniques. Checking security predicates during symbolic execution is an advanced technique, which focuses on solving extra constraints for input data to find an error and generate an input file to reproduce it. In this paper we propose a method for automatically detecting errors with the help of dynamic symbolic execution, combining hybrid fuzzing and checking security predicates. Firstly, we run hybrid fuzzing, which is required to increase number of corpora seeds. Then we minimize corpora. Thus, it would give the same coverage as the original corpora. After that we check security predicates on minimized corpora. Thus, security predicates allow to find errors like division by zero, out of bounds access, integer overflow, and more. Security predicates results are later verified with sanitizers to filter false positive results. As a result of applying the proposed method to different open source programs, we found 11 new different errors in 5 projects.

Keywords: dynamic symbolic execution; DSE; fuzzing; security predicate; automatic error detection; security development lifecycle; SDL; binary code; sanitizer; bug; error; CWE

For citation: Vishnyakov A.V., Kobrin E.A., Fedotov A.N. Error detection in binary code with dynamic symbolic execution. Trudy ISP RAN/Proc. ISP RAS, vol. 34, issue 2, 2022, pp. 25-42 (in Russian). DOI: 10.15514/ISPRAS–2022–34(2)–3

Acknowledgements. This work was supported by RFBR grant 20-07-00921 A.

1. Введение

Современное программное обеспечение стремительно развивается. Новый код неизбежно приносит с собой новые ошибки и уязвимости [1]. Всё больше и больше индустриальных компаний следуют безопасному циклу разработки [2-4] для улучшения качества программного обеспечения и защиты его от злонамеренных атак. Главной целью безопасного цикла разработки является поиск ошибок в разрабатываемом программном обеспечении, включающих в себя как простые ошибки, приводящие к неправильной работе программы в несущественных местах, так и те, которые приводят к аварийному завершению работы программы или открывают доступ для различных злонамеренных атак. В результате применения безопасного цикла разработки программное обеспечение становится безопаснее, надежнее и качественней.

Одной из наиболее распространенных технологий для поддержки безопасного цикла разработки остается фаззинг с обратной связью по покрытию [5-6]. Гибридный фаззинг извлекает выгоду из динамической символьной интерпретации [7-17], открывая сложные состояния программы, трудно досягаемые с помощью простого фаззинга.

Фаззинг представляет собой метод автоматического тестирования программ, при котором программе на вход подается множество наборов входных данных, после чего анализируется реакция программы и генерируются новые входные данные. Целью фаззинга является поиск входных данных, которые приведут выполнение программы к аварийному завершению, переполнению по памяти или зависанию.

Символьная интерпретация представляет собой метод автоматического тестирования, при котором происходит интерпретация программы, где конкретным значениям переменных, зависящих от входных данных, сопоставляются символьные переменные, принимающие произвольные значения. Анализируемый путь выполнения программы может быть описан предикатом пути – системой уравнений и неравенств, зависящих от символьных переменных, решение которой обеспечивает прохождение потока управления по тому же пути. Предикатом безопасности мы называем дополнительные условия на предикат пути, которые позволяют проявить ошибочную ситуацию в программе.

В данной работе мы описываем разработанный метод поиска ошибок работы с памятью и неопределенного поведения в программном обеспечении, использующий гибридный фаззинг программ и предикаты безопасности, и представляем результаты практического применения этого метода на программах с открытым исходным кодом.

2. Обзор

2.1 KLEE

KLEE – это инструмент для символьной интерпретации, способный автоматически генерировать тесты для достижения высокого покрытия и поиска ошибок на разнообразном наборе сложных программ [18]. Во время символьной интерпретации KLEE для потенциально опасных операций генерирует условные переходы, которые проверяют, существует ли какой-либо набор входных данных, который может привести к проявлению ошибки. Например, инструкция деления генерирует условный переход, проверяющий, может ли делитель быть равен нулю. Такие условные переходы будут работать так же, как и обычные условные переходы: при выполнимости предиката пути, содержащего такой условный переход, будет сгенерирован входной файл для воспроизведения ошибки и будет порожден новый процесс работы KLEE, который продолжит анализ пути с условием, обратным условию для проявления ошибки. Таким образом, даже если проверка успешна (то есть обнаружена ошибка), KLEE продолжает работу по пути, на котором ошибка не проявляется, добавляя отрицание условия проверки в качестве ограничения на путь.

2.2 Mayhem

Mayhem – это система для автоматического обнаружения программных ошибок в бинарных файлах [19]. Каждая ошибка, найденная с помощью Mayhem, сопровождается работающей эксплуатацией этой ошибки. Mayhem генерирует эксплойты для каждой возможной перезаписи счетчика инструкций, обычно вызываемой переполнением буфера. Когда Mayhem находит символьный адрес инструкции, он пытается сгенерировать эксплуатацию перехода по регистру. В таких случаях счетчик инструкций должен указывать на инструкции по типу `jmp eax`, а регистр `eax` должен указывать на место в памяти, где может быть размещен какой-то произвольный код. Такие условия конструируются в виде формулы, которая передается математическому решателю. Если решение существует, то ошибка может быть проэксплуатирована. Если решения не существует, то генерируется более простая формула, где счетчик инструкций должен указывать конкретно на место в памяти, где может быть размещен произвольный код.

Для форматных строк Mayhem проверяет, содержат ли аргументы для форматной строки или сама форматная строка символьные байты. Если так, то Mayhem пытается разместить в аргументе данные, которые приведут к перезаписи адреса возврата из форматирующей функции.

2.3 Google Sanitizers

Google Sanitizers – это набор инструментов с открытым исходным кодом для динамического анализа кода [20], позволяющие обнаруживать различные ошибочные ситуации во время выполнения программы.

Реализация санитайзеров основана на инструментации программного обеспечения на этапе его компиляции. Например, при использовании AdressSanitizer различные манипуляции с памятью в процессе выполнения программы одновременно отображаются определенным способом в отдельной структуре данных, называемой теневой памятью, позволяющей следить за корректностью выполнения различных операций с памятью. При использовании UndefinedBehaviorSanitizer в код программы на этапе компиляции встраиваются различные проверки на целочисленные переполнения и другие ошибки неопределенного поведения.

Существует несколько санитайзеров: AddressSanitizer, LeakSanitizer, ThreadSanitizer, UndefinedBehaviorSanitizer и MemorySanitizer.

AddressSanitizer позволяет обнаруживать ошибки работы с памятью во время ее выполнения. Он способен обнаружить переполнение буфера на стеке, куче, а также переполнение глобальных буферов. Также он позволяет обнаруживать использования указателей после освобождения памяти, на которую они ссылаются, и обнаруживает двойные и некорректные освобождения памяти, а также другие ошибки работы с памятью.

LeakSanitizer позволяет обнаруживать утечки памяти. Для использования этого санитайзера отдельно не требуется инструментации программы, однако также он встроен в AddressSanitizer, что позволяет использовать эти два санитайзера совместно.

ThreadSanitizer позволяет обнаруживать ошибки гонки данных, которые происходят, когда разные потоки выполнения обращаются к одним и тем же областям памяти без синхронизации и где хотя бы одно из обращений являются записью в эту область памяти.

UndefinedBehaviorSanitizer позволяет обнаруживать ошибки неопределенного поведения, являющиеся результатом выполнения операций с неспецифицированными семантиками, таких как деление на нуль, разыменование нулевого указателя, использование неинициализированной нестатической переменной и т.п.

MemorySanitizer позволяет обнаруживать ошибки неинициализированного чтения из памяти. Этот санитайзер находит случаи, когда производится чтение из памяти, выделенной на стеке или в куче, до того, как в эту память было что-либо записано.

2.4 SAVIOR

SAVIOR – это фреймворк для гибридного тестирования программного обеспечения, нацеленный на поиск ошибок [21]. Работа SAVIOR основывается на двух механизмах: механизме распределении приоритетов, основанном на ошибках, и механизме верификации с учетом ошибок.

Суть механизма распределения приоритетов заключается в том, чтобы производить символьную интерпретацию не на всех наборах входных данных, а отдавать приоритет таким входным данным, которые с большей вероятностью приведут к обнаружению программных ошибок. До тестирования программного обеспечения SAVIOR анализирует исходный код и статически помечает потенциально опасные места в программе. Также SAVIOR высчитывает набор базовых блоков, достижимых от каждого условного перехода. Во время символьной интерпретации SAVIOR считает приоритетными те наборы входных данных, которые смогут привести к посещению наибольшего числа важных условных переходов. Таким образом, SAVIOR ускоряет обнаружение новых ошибок в программах.

Суть механизма верификации заключается в следующем. Получая наборы входных данных в результате фаззинг-тестирования, SAVIOR выполняет программу на этих входных данных и для каждого ранее помеченного потенциально опасного места верифицирует

соответствующий ему предикат, проверяя его с учетом пути выполнения. Если предикат удовлетворим, то ошибка подтверждена. Это позволяет SAVIOR генерировать доказательство обнаружения ошибки в виде конкретных входных данных или ее несуществования на конкретном пути выполнения программы.

Пометка потенциально опасных мест производится с помощью Undefined Behavior Sanitizer. На основе инструментации от санитайзера SAVIOR инструментирует потенциально опасные места в программе предикатами, удобными для математического решателя, которые будут проверяться на этапе верификации. Таким образом, SAVIOR поддерживает предикаты для ошибок выхода за границы массива, битовых сдвигов на слишком большое значение, знакового и беззнакового целочисленного переполнения. С помощью статического анализа SAVIOR учитывает знаковость операндов и их размеры при составлении предикатов для дальнейшей проверки на этапе верификации.

2.5 ParmeSan

ParmeSan – инструмент для фаззинга с обратной связью по санитайзерам, который оптимизирован специально для покрытия ошибок [22]. Основной идеей является использование инструментации, производимой санитайзерами, для обеспечения эффективного механизма поиска интересных базовых блоков для обратной связи с фаззером. В отличие от фаззеров с обратной связью по покрытию, которые вслепую покрывают все базовые блоки программы, ParmeSan направлен на исследование путей выполнения программы, имеющих наибольший шанс проявления ошибок за кратчайшее время.

Первым делом ParmeSan исследует интересные цели из данного санитайзера, которым была проинструментирована программа, убирая неинтересные проверки. Затем динамически конструируется граф потока управления, чтобы направить фаззинг на конкретные цели. Сам процесс фаззинга основан на двух этапах: фаззинга для конструирования графа потока управления и фаззинга целевых точек программы, нацеленного на поиск ошибок в программе. Для ускорения работы второго этапа ParmeSan использует доступную из анализа потока данных информацию. Одной из основных идей, заложенных в работу ParmeSan, является то, как система направляет фаззинг по путям выполнения, ведущим к интересным блокам. ParmeSan в процессе работы строит граф потока управления. ParmeSan выполняет программу на исходных наборах входных данных и смотрит, до каких базовых блоков дошло выполнение программы. Основываясь на построенном графе потока управления, ParmeSan с помощью введенной авторами метрики вычисляет дистанцию от пройденных базовых блоков до целевых блоков, и на основе полученных значений метрики и графа потока управления направляет фаззинг так, чтобы как можно скорее дойти до целевых блоков, содержащих потенциальную ошибку. Для этого фаззер, представленный ParmeSan, при выборе следующего набора входных данных для мутации выбирает тот набор, при выполнении которого были посещены базовые блоки, имеющие наименьшую дистанцию с целевыми блоками, и повышает приоритет для инвертирования тех условных переходов, которые наиболее близко расположены к целевым базовым блокам.

Санитайзеры инструментируют программу двумя разными способами. Некоторые инструментации просто обновляют внутренние структуры данных (например, теневую память), другие инструментации используются, когда санитайзеры обнаруживают ошибку, используя условие условного перехода, которое взаимодействует с внутренними структурами данных или непосредственным состоянием программы. Цель ParmeSan – направить фаззинг в те точки программы, где санитайзер обновляет свои внутренние структуры данных, и решать предикаты, вставленные санитайзерами, выполнимость которых означает обнаружение ошибки, с помощью мутаций на основе анализа потока данных.

Таким образом, процесс работы ParmeSan состоит из трех фаз. Первая фаза – быстрое исследование путей исполнения с опорой на покрытие и этап трассировки, чтобы получить

граф потока управления. В течение первой фазы ParmeSan собирает трассы и пытается построить граф потока управления настолько точно, насколько это возможно. Вторая фаза – направленное исследование, чтобы достичь целевых базовых блоков. Третья фаза – генерация входных данных, проявляющих ошибку, – начинается, когда целевой блок был достигнут.

2.6 IntScope

IntScope – инструмент для обнаружения ошибок целочисленного переполнения при помощи символьной интерпретации программного обеспечения [23]. IntScope предлагает следующий подход для обнаружения ошибок целочисленного переполнения. При помощи символьной интерпретации анализируются инструкции, реализующие арифметические операции. Далее найденные места с потенциальной ошибкой целочисленного переполнения проверяются лениво, то есть арифметическая инструкция проверяется не сразу, а лишь когда помеченное символьное значение используется в чувствительных точках, таких как выделение памяти функциями `malloc`, `calloc`. IntScope предупреждает о потенциальной ошибке целочисленного переполнения, только если помеченное символьное значение, используемое в чувствительных местах, может переполниться.

Недостоверный источник – арифметическая инструкция в программе, где потенциально может произойти ошибка целочисленного переполнения, и один из операндов получен из помеченных данных. Помеченные данные могут быть получены из недоверенных источников входных данных, таких как сеть, файлы или опции командной строки. Функциями, с помощью которых могут быть получены помеченные данные, считаются `read`, `fread`, `recv`.

Сток ошибки – место в программе, в котором использование переполненного значения может привести к дальнейшим программным уязвимостям. В IntScope выделяются следующие стоки ошибок:

- выделение памяти (использование переполненного значения в аргументах таких функций, как `malloc`);
- доступ к памяти (переполненное значение использовано в качестве индекса или смещения при разыменовании адреса);
- условный переход (использование переполненного значения в условном переходе, который может привести к пропуску проверок на безопасность).

При достижении потенциального стока ошибки, использующего уже найденный источник ошибки, происходит соответствующая проверка. Если сток – аргумент функции выделения памяти, то значение проверяется как беззнаковое. Также информацию о знаковости символьного значения IntScope получает с помощью инструкций условных переходов, таких как `JG` (знаковый), `JA` (беззнаковый) и подобных.

2.7 Обнаружение уязвимостей целочисленного переполнения в двоичном коде программного обеспечения

В данной работе авторы предлагают свой подход для поиска ошибок целочисленного переполнения в бинарных файлах [24]. Подход основан на символьной интерпретации и двойном отображении памяти. Для этого строится усеченный граф потока управления, основанный на машинном коде. Слои графа проверяются на выполнение условий для обнаружения уязвимости.

Главная идея подхода – использовать символьную интерпретацию кода для конструирования условий, проявляющих уязвимость, на входные переменные. После эти условия нужно будет решить.

Представленный подход можно разделить на следующую последовательность фаз.

- **Построение графа потока управления**. Входной файл используется для построения графа потока управления тестируемой программы, в результате которого должны быть обнаружены вершины ввода и вывода. Вершина ввода отображает части кода, где происходит ввод данных, а вершина вывода отображает части кода, где происходит вызов функций выделения памяти.
- **Отсеивание неиспользованного пути в графе потока управления**. Все вершины, которые не были использованы в любом из путей от вершины ввода до вершины вывода должны быть удалены.
- **Построение символьного входа**. Входные данные и неинициаллизированные ячейки памяти полагаются символьными. Каждая ячейка изначально отображается как пара, состоящая из символьного числа и символьного адреса.
- **Символьная эмуляция**. Каждый путь обрабатывается независимо. Обработка включает в себя символьную эмуляцию машинных инструкций. Она затрагивает обе части отображения ячеек (число и адрес). Символьные ячейки могут потерять один из этих параметров, если в процессе интерпретации становится понятно, является ли символьная переменная адресом или числом. Например, если переменная была разыменована, то она является адресом, тогда параметр числа отбрасывается.
- **Построение системы условий**. Система условий описывает доступность вершины вывода от вершины ввода. Ограничения добавляются в систему по мере исследования путей выполнения. Условия для проверки возможности ошибки используются в качестве последнего условия в системе.
- **Проверка существования решения системы**. Созданные условия проверяются с помощью автоматической системы доказательства теорем. Если система имеет решение, то программа уязвима, а сгенерированное решение является доказательством найденной ошибки.

Символьная эмуляция программы применяется к каждому пути в графе, приводящему к назначенной вершине вывода. Уравнения и неравенства на символьные переменные, из которых составляется система, которую позднее будет решать математический решатель, могут встречаться между вершинами графа и являться условиями операторов ветвления или условиями на переполнение в инструкциях, реализующих арифметические операции, таких как `add`, `sub`, `mul`, `imul`, `shl` и т.д.

Составленная система уравнений и неравенств на символьные переменные проверяется решателем последовательно по достижению вершины вывода, то есть вызова некоторой опасной функции, использование переполненного значения в которой может привести к серьезным последствиям. Если система решаема, то это говорит об уязвимости вызываемой функции.

3. Символьные предикаты безопасности

Предикаты безопасности – это дополнительные условия на предикат пути, которые позволяют обнаружить ошибку в программе. Для построения предиката безопасности во время символьной интерпретации программы анализируются определенные ее инструкции и места, которые считаются опасными. Если такое место находится, то составляется предикат безопасности, представляющий собой уравнения и неравенства над символьными переменными и константными значениями. Затем предикат безопасности и предикат пути совместно образуют систему уравнений и неравенств, которая передается в качестве входных данных SMT-решателю [25]. Решение такой системы уравнений представляет собой набор входных данных, которые обеспечивают как прохождение потока управления по исследуемому пути, так и проявление конкретной программной ошибки. Таким образом,

запуск программы на этих входных данных воспроизводит найденные ошибки. Предикаты безопасности были разработаны в рамках инструмента Sydr [26].

Одним из разработанных предикатов безопасности является предикат безопасности для поиска ошибок целочисленного деления на нуль. Во время символьной интерпретации бинарного файла каждая символьная инструкция деления, такая как `div` или `idiv`, проверяется предикатом безопасности для поиска ошибки деления на нуль. Если делитель инструкции является символьным, то составляется уравнение на равенство делителя нулю. Затем получившееся уравнение конъюнктируется с предикатом пути и получается соответствующий предикат безопасности, который далее проверяется по вышеописанной схеме.

3.1 Выход за границы массива

Выход за границы массива – одна из самых опасных и распространенных программных ошибок [27]. Для обнаружения таких ошибок мы строим предикат безопасности для каждого разыменования символьного адреса.

Для того, чтобы построить предикат безопасности для ошибки выхода за границы массива, сначала нужно определить границы массива [*lower_bound*; *uppper_bound*). Определив границы массива, мы строим предикат безопасности, который возвращает истину, если символьный адрес выходит за границы массива. Далее получившиеся неравенства конкатенируются с предикатом пути и итоговый предикат проверяется на выполнимость. Если предикат выполним, то генерируются входные данные, приводящие к выходу за границу массива.

В случае, если Sydr не может обнаружить обе границы массива, составляется предикат безопасности для выхода за нижнюю границу массива, которую можно получить эвристическими методами. Например, в адресном выражении `[rdx + rax]`, где `rax` является конкретным адресом базы, а `rdx` является символьным индексом, Sydr полагает конкретную часть регистра `rax` за нижнюю границу массива.

Для определения символьных границ адресов мы поддерживаем теневую кучу и теневой стек. Sydr оборачивает все функции для работы с динамической памятью, такие как `malloc`, `calloc`, `realloc`, `free` и т.д., и при вызове таких функций обновляет теневую кучу, которая содержит все границы выделенных в памяти буферов. Для каждой встреченной инструкции `call` Sydr кладет адрес, по которому располагается адрес возврата, на теневой стек; а для каждой инструкции возврата `ret` Sydr снимает элементы с теневого стека в соответствии с текущим значением указателя стека.

Соответственно, когда происходит разыменование символьного адреса, Sydr обнаруживает границы соответствующего буфера следующим образом. Если текущее конкретное значение адреса находится в теневой куче, то обе границы можно получить из нее. Если адрес указывает на стек, то ближайший адрес на теневом стеке, больший текущего адреса, будет считаться верхней границей массива. Нижняя граница вычисляется эвристически из конкретной части формулы символьного адреса. Главной идеей такого способа является суммирование конкретных частей символьной формулы.

Также мы оборачиваем функции копирования, такие как `memcpy`, `memmove`, `memset` и т.д., чтобы обнаруживать переполнения буфера, к которым может привести небезопасное использование таких функций. Если аргумент размера копирования является символьным, Sydr пытается сделать его таким, чтобы выйти за верхнюю границу.

Перед решением предиката безопасности мы составляем конъюнкцию с дополнительными условиями на предикат безопасности, которые позволяют обнаружить ошибку, которая вероятнее приведет к аварийному завершению программы, перезаписав адрес возврата при обращении к памяти или обращаясь к отрицательному адресу. Если такой более сильный

предикат не удовлетворим, Sydr возвращается к решению изначального предиката безопасности. Если при этом и адрес, и значение, которое будет записано по адресу, являются символьными, Sydr предупреждает о возможности ошибки write-what-where для такого выхода за границы.

3.2 Целочисленное переполнение

Ошибка целочисленного переполнения является одной из самых типичных программных ошибок [27], однако она достаточно часто встречается в бинарном коде. Sydr будет работать слишком долго, если он будет проверять каждое место в программе, где может возникнуть целочисленное переполнение. Более того, в некоторых ситуациях, таких как вычисление значения хэш-функции, целочисленное переполнение является корректным. Поэтому мы выделяем только критические части программы и для них проверяем предикаты безопасности. В отличие от других предикатов безопасности, мы отделяем сток ошибки от ее источника. Источник – это инструкция, где может произойти целочисленное переполнение (например, различные арифметические инструкции, такие как сложение или умножение). Сток – это место, в коде, где предшествующее целочисленное переполнение может привести к критической ошибке.

Мы решаем предикаты безопасности для поиска ошибок целочисленного переполнения для стоков ошибки, которые используют потенциально переполненное значение. Например, стоками могу быть условные переходы (изменение потока управления в зависимости от переполненного значения), адреса для обращения к памяти и аргументы функций. Особенно критично может быть использование переполненного значения в аргументах таких функций, как `malloc`, `memcpy` и т.п. [23, 24] Мы оборачиваем некоторые функции стандартной библиотеки и полагаем все их символьные аргументы потенциальными стоками. Для остальных функций мы проверяем первые три аргумента в соответствии со стандартным соглашением о вызовах.

При анализе конкретной инструкции мы проверяем, является ли она потенциальным источником, то есть является ли она арифметической и является ли хотя бы один из ее операндов символьным. Если так, то мы строим предикаты безопасности для беззнакового (флаг `CF`) и знакового (`OF`) целочисленного переполнения. Для большинства арифметических инструкций предикат безопасности будет истинным, когда соответствующий флаг выставлен в единицу. Исключением являются инструкции битовых сдвигов, для которых мы составляем предикат безопасности отдельным способом так, чтобы он соответствовал целочисленному переполнению в таких случаях.

Затем мы проверяем, задействовано ли потенциально переполненное значение, то есть источник ошибки, в вычислении стока. Для этого мы проверяем, является ли абстрактное синтаксическое дерево источника ребенком абстрактного синтаксического дерева стока. Если так, то вычисление стока содержит потенциально переполненное значение.

Далее мы узнаем знаковость арифметической операции с помощью обратного слайсинга [12]. Мы проверяем инструкции условных переходов, начиная с последнего в текущем предикате пути, пока какая-либо такая инструкция не сможет нам дать информацию о знаковости какого-либо из операндов (например, инструкция JL говорит о том, что вычисления знаковые) [23]. Также мы можем узнать знаковость вычислений с помощью реализованных нами семантик функций вида `strto*l` [26].

Кроме того, мы составляем дополнительные условия для функций выделения памяти, таких как `malloc`, `calloc`, и функций копирования, таких как `memcpy`, так как переполнение в таких функциях может привести к более серьезным последствиям. Для функций выделения памяти мы составляем дополнительное условие, чтобы значение, получившееся в результате переполнения, оказалось меньше чем значение, на котором происходило конкретное выполнение, но при этом не равнялось нулю, что позволяет переполнить размер выделяемой

памяти и при этом ее успешно выделить (если переполненное число будет больше количества памяти на машине, то `malloc` вернет нуль). Для функций копирования мы хотим получить переполненное значение больше чем то, что было при конкретном выполнении программы. Если предикат с дополнительными условиями не удовлетворим, мы проверяем изначальный предикат безопасности.

Также мы обрабатываем случай, когда арифметическая операция производится с типами, размер которых меньше чем `sizeof(int)`. В таком случае в силу расширения целочисленных типов данных при вычислениях оригинальная инструкция не может привести к переполнению, поэтому мы самостоятельно строим новые предикаты безопасности для целочисленного переполнения с корректным размером операндов.

4. Метод автоматизированного поиска ошибок символьными предикатами безопасности

Для применения символьных предикатов безопасности был разработан следующий метод автоматизированного поиска ошибок. Сначала производится гибридный фаззинг выбранного проекта, при котором совместно работают фаззер и инструмент динамической символьной интерпретации Sydr [12]. После этого производится минимизация корпуса входных данных, полученных в результате гибридного фаззинга. Затем идет этап проверки предикатов безопасности на всех файлах, полученных после минимизации корпуса. Результаты предикатов безопасности верифицируются с помощью исполняемого файла, собранного с санитайзерами. Верифицированные результаты оцениваются человеком на критичность и корректность. Также возможна автоматизированная дедупликация, кластеризация и оценка критичности аварийных завершений с помощью Casr [28], если таковые были найдены на каких-либо наборах входных данных.

Для фаззинга мы использовали libFuzzer [5]. Для подготовки к этапу гибридного фаззинга требуется собрать два исполняемых файла для каждой фаззинг-цели проекта: первый должен быть собран с санитайзерами и libFuzzer'ом, второй – без санитайзеров, но с отладочной информацией (он будет использоваться Sydr'ом). После подготовки начинается этап гибридного фаззинга, в результате которого получается корпус входных данных, полученных от фаззера и Sydr'а.

Во время работы предикатов безопасности анализируется лишь конкретный путь исполнения, поэтому для проверки как можно большего количества инструкций программы необходимо проверить как можно больше входных файлов. Однако в получившемся в результате работы этапа гибридного фаззинга корпусе может содержаться большое количество файлов, похожих друг на друга и покрывающих один и тот же код в целевом проекте. Поэтому перед началом этапа проверки предикатов безопасности необходимо минимизировать корпус, чтобы оставить в нем наиболее эффективное подмножество файлов, дающее такое же покрытие кода программы, но при этом содержащее намного меньше файлов, чем изначальный корпус.

После минимизации корпуса начинается этап проверки предикатов безопасности. Проверка предикатов безопасности запускается для каждого файла из минимизированного корпуса. Новые файлы, генерируемые предикатами безопасности, верифицируются с помощью исполняемого файла, собранного с санитайзерами. Если при запуске этого исполняемого файла санитайзеры выдают предупреждение для строчки программы, аналогичной той, для которой был сгенерирован проверяемый набор входных данных, то результат считается верифицированным и сохраняется отдельно. Также, если запуск собранного с санитайзерами исполняемого файла на проверяемом наборе приводит к срабатыванию санитайзеров, отсутствовавшему при запуске этого исполняемого файла на оригинальном файле (т.е. файле, на котором проводилась символьная интерпретация и проверка предикатов безопасности),

результат также считается верифицированным. Отдельно проверяется, приводит ли проверяемый набор входных данных к аварийному завершению.

После этапа проверки предикатов безопасности стоит вручную проверить, какие именно были найдены ошибки. Зачастую найденные ошибки действительно приводят к срабатыванию санитайзеров, но с точки зрения логики работы программы могут ошибками не являться. Также, если в результате проверки предикатов безопасности были найдены наборы данных, приводящие к аварийному завершению, стоит провести их автоматический анализ и кластеризацию с помощью инструмента Casr [28], который позволяет получать подробный отчет об аварийных завершениях.

5. Результаты применения метода к проектам с открытым исходным кодом

Разработанный метод автоматизированного поиска ошибок был применен к ряду проектов с открытым исходным кодом. В результате применения разработанного метода в них были найдены новые ошибки.

5.1 FreeImage

FreeImage – это библиотека с открытым исходным кодом [29], которая поддерживает работу с изображениями различных форматов, таких как PNG, BMP, JPEG, TIFF и т.д. С помощью разработанного метода в коде FreeImage было найдено 5 ошибок целочисленного переполнения на этапе проверки предикатов безопасности [30].

```
unsigned off_head, off_setup, off_image, i, temp;
...
fseek(ifp, off_setup + 792, SEEK_SET);
```

Листинг 1. Беззнаковое целочисленное переполнение в fseek (BMP)
Listing 1. Unsigned integer overflow in fseek (BMP)

На листинге 1 приводится фрагмент кода, относящийся к обработке изображений формата BMP, в котором была найдена ошибка беззнакового целочисленного переполнения при вызове функции `fseek`, смещающей текущую позицию в файле. Так как в коде отсутствуют проверки введенных значений, выражение `off_setup + 792` может привести к переполнению, что в свою очередь может привести к неправильной позиции в файле, измененной с помощью функции `fseek`. Источником ошибки, определенным проверкой предикатов безопасности, является инструкция сложения, реализующая вычисление выражения `off_setup + 792`, а стоком ошибки является аргумент смещения в функции `fseek`.

```
fseek(ifp, offset + length - 4, SEEK_SET);
```

Листинг 2. Знаковое целочисленное переполнение в fseek (TIFF)
Listing 2. Signed integer overflow in fseek (TIFF)

Аналогичная ошибка представлена на листинге 2, где также была найдена ошибка целочисленного переполнения при вызове функции `fseek`, но уже в части кода, отвечающей за обработку изображений формата TIFF. Найденная ошибка целочисленного переполнения происходит в аргументе смещения функции `fseek` при вычислении выражения `offset + length - 4`.

На листинге 3 представлен фрагмент кода FreeImage, где в аргументе функции `parse_tiff` происходит неявное преобразование типа `long` к типу `int`, где переменная `thumb_offset` имеет тип `long`. Так как предикат безопасности для поиска ошибок целочисленного переполнения определяет размер операндов по размеру типа стока, являющегося аргументом

функции, то предикат безопасности посчитал эту ошибку как целочисленное переполнение, которое по факту является неявным преобразованием типа, где значение, получающееся из выражения `thumb_offset + 12` типа `long`, не может быть помещено в аргумент функции типа `int` меньшего размера.

```
int parse_tiff(int base);
...
parse_tiff(thumb_offset + 12);
```

Листинг 3. Неявное преобразование в parse_tiff
Listing 3. Implicit conversation in parse_tiff

```
if (*len * tagtype_dataunit_bytes[(*type <= LIBRAW_EXIFTAG_TYPE_IFD8)
      ? *type : 0] > 4)
    fseek(ifp, get4() + base, SEEK_SET);
```

Листинг 4. Беззнаковое переполнение в условном переходе
Listing 4. Unsigned integer overflow in branch condition

На листинге 4 приводится фрагмент кода, где в условии оператора ветвления была найдена ошибка беззнакового целочисленного переполнения при умножении `*len` на `tagtype_dataunit_bytes[(*type <= LIBRAW_EXIFTAG_TYPE_IFD8) ? *type : 0]`. Инструкцией источника является умножение, а стоком – инструкция условного перехода. Были подобраны такие входные данные, что при целочисленном переполнении меняется поток управления, что приводит к ошибочной работе программы.

На листинге 5 приводится фрагмент кода, где вычисляется некоторая ширина. При ее вычислении может произойти ошибка беззнакового целочисленного переполнения. Значение позже используется в условных переходах и многих других местах, и найденная ошибка может привести к неправильной работе программы.

```
width = raw_width - left_margin - (get4() & 7);
```

Листинг 5. Беззнаковое целочисленное переполнение при вычислении ширины
Listing 5. Unsigned integer overflow in width computation

5.2 xlnt

xlnt – это библиотека с открытым исходным кодом для языка C++ для управления электронными таблицами и их чтения/записи из/в файлы формата XLSX [31]. С помощью разработанного метода в xlnt была найдены две ошибки целочисленного переполнения [32, 33] и ошибка выхода за границы массива [34].

```
in_->seekg(static_cast<std::ptrdiff_t>(sector_data_start() +
     sector_size() * static_cast<std::size_t>(id)));
std::vector<byte> sector(sector_size(), 0);
```

Листинг 6. Беззнаковое целочисленное переполнение в xlnt
Listing 6. Unsigned integer overflow in xlnt

На листинге 6 приведен фрагмент кода, где были найдены обе ошибки беззнакового целочисленного переполнения в аргументе функции `seekg` в выражении `sector_data_start() + sector_size() * static_cast <std::size_t>(id)` при умножении и сложении соответственно. Источником ошибки является вычисление приведенного выражения, а стоком – аргумент вызываемой функции, являющийся результатом вычислений. Сама ошибка не приводит к серьезным последствиям, однако

подобранные для воспроизведения переполнения значения могут привести к следующим исходам.

Был найден набор входных данных, при котором значения `sector_size()` и `static_cast<std::size_t>(id)` достаточно велики, чтобы при перемножении привести к переполнению. Так как значение `sector_size()` очень велико, то вызов конструктора `std::vector<byte> sector(sector_size(),0)` приводит к аварийному завершению работы программы из-за невозможности выделить слишком большое количество памяти. Также был найден набор входных данных, при котором значение `id` равняется -1, и при приведении этого значения к типу `size_t` получается очень большое значение. Это также приводит к переполнению в указанном выражении. Однако, так как значение `sector_size()` не было подобрано большим, то аварийного завершения программы при вызове конструктора на следующей строке не происходит. Но далее в том же файле исходного кода в функции `read_directory` происходит обращение на чтение из памяти: `entries_[static_cast<std::size_t>(current_entry_id)]`. При обращении в память в силу подобранных для переполнения значений происходит выход за границы массива и аварийное завершение.

```
sector_chain
compound_document::follow_chain(sector_id start,
                                const sector_chain &table)
{
    auto chain = sector_chain();
    auto current = start;
    while (current >= 0)
    {
        chain.push_back(current);
        current = table[static_cast<std::size_t>(current)];
    }
    return chain;
}
```

Листинг 7. Выход за границы массива в xlnt
Listing 7. Out of bounds access in xlnt

На листинге 7 приведен фрагмент кода, где была найдена ошибка выхода за границы буфера. Ошибка происходит при обращении по адресу на строке `current = table[static_cast<std::size_t>(current)]`. Значение `current` было подобрано равным 2147483647, что приводит к обращению в память за границами буфера. Ошибка была найдена независимо как фаззером, так и предикатами безопасности.

```
int sign = 0;
uint32_t i = 0;
uint32_t seconds = 0;
for (*endptr = nptr; **endptr; (*endptr)++) {
   switch (**endptr) {
   ...
   case '9':
   i *= 10;
   ...
   }
}
```

Листинг 8. Беззнаковое целочисленное переполнение в unbound
Listing 8. Unsigned integer overflow in unbound

5.3 unbound

unbound – это проверяющий, рекурсивный, кэширующий распознаватель DNS [35]. В коде проекта unbound при помощи разработанного метода была найдена ошибка беззнакового целочисленного переполнения. Фрагмент кода, в котором была найдена ошибка, представлен на листинге 8.

Переполнение происходит в функции `sldns_str2period`, которая обрабатывает строку и переводит ее в некоторое значение времени. Переполнение происходит в переменной `i` внутри цикла обработки строки. Переполнение может привести к неправильной работе со строкой и вследствие к неправильному результату работы функции. Ошибка была исправлена автором проекта [36].

5.4 hdp

HDF – это библиотека, предоставляющая набор утилит для работы с файлами формата hdf, который используется для хранения научных данных [37]. Одной из таких утилит является hdp, которая обеспечивает быстрое получение основной информации из hdf-файлов. В ходе применения разработанного метода автоматизированного поиска ошибок в hdp была найдена ошибка целочисленного деления на нуль, приводящая к аварийной остановке работы утилиты [38].

```
int32 buf_size;
/* we are bounded above by VDATA_BUFFER_MAX */
buf_size = MIN(total_bytes, VDATA_BUFFER_MAX);
/* make sure there is at least room for one record in our buffer */
chunk = buf_size / hsize + 1;
```

Листинг 9. Деление на нуль в hdp
Listing 9. Division by zero in hdp

На листинге 9 приведен фрагмент кода, в котором происходит деление на нуль. Код приведен из функции `VSread`, которая считывает данные в некоторый буфер. Приведенный фрагмент кода вычисляет, какое количество элементов можно считать за раз. Так как в коде отсутствует проверка на равенство делителя нулю, на этапе проверки предикатов безопасности подбирается значение `hsize`, равное 0, что приводит к ошибке деления на нуль.

5.5 miniz

miniz – это высокопроизводительная библиотека сжатия данных без потерь в одном исходном файле, реализующая стандарты формата сжатия данных zlib и Deflate [39]. Также miniz входит в зависимости PyTorch – популярного фреймворка для машинного обучения [40]. В результате применения разработанного метода тестирования в miniz была найдена ошибка целочисленного переполнения [41].

```
if (cdir_size < pZip->m_total_files * MZ_ZIP_CENTRAL_DIR_HEADER_SIZE)
   return mz_zip_set_error(pZip, MZ_ZIP_INVALID_HEADER_OR_CORRUPTED);
```

Листинг 10. Целочисленное переполнение в miniz
Listing 10. Integer overflow in miniz

На листинге 10 приведен фрагмент кода, в котором может произойти целочисленное переполнение. Приведенное условие проверяет заголовок файла на корректность, и в случае, если заголовок некорректен, возвращается ошибка. В результате применения разработанного метода удалось найти входные данные, которые приводят к ошибке целочисленного переполнения. Однако для условных переходов мы накладываем дополнительные условия на

предикат безопасности, чтобы при проявлении ошибки был изменен поток управления. Так как при конкретном исполнении проверка проходила успешно, то были подобраны такие входные данные, при которых происходит ошибка целочисленного переполнения и меняется поток управления, в результате чего происходил возврат из функции с сообщением об ошибке. Для такого случая мы поменяли дополнительные условия на предикат безопасности так, чтобы при проявлении ошибки поток управления сохранялся. В результате этого эксперимента мы получили такие входные данные, при которых заголовок файла является некорректным, но благодаря целочисленному переполнению булево выражение в условии принимает значение истины, и проверка проходит успешно.

6. Заключение

В данной работе был предложен метод поиска ошибок в бинарном коде методами динамической символьной интерпретации, заключающийся в фаззинге для генерации корпуса входных данных и проверке предикатов безопасности на сгенерированных данных. С помощью разработанного метода можно находить ошибки целочисленного переполнения, выхода за границы массива, деления на нуль и другие. В результате практического применения разработанного метода к программам с открытым исходным кодом были найдены 11 различных новых ошибок в 5 разных проектах.

Список литературы / References

[1] CWE – common weakness enumeration. URL: https://cwe.mitre.org/.

[2] M. Howard and S. Lipner. The Trustworthy Computing Security Development Lifecycle. Microsof, 2006. URL:http://msdn.microsoft.com/en-us/library/ms995349.aspx.

[3] ISO/IEC 15408-3:2008. Information technology – Security techniques – Evaluation criteria for IT security – Part 3: Security assurance components.

[4] ГОСТ Р 56939-2016: Защита информации. Разработка безопасного программного обеспечения. Общие требования. Национальный стандарт РФ, 2016. / GOST R 56939-2016: Information Protection. Secure Software Development. General Requirements. National Standard of Russian Federation, 2016 (in Russian).

[5] K. Serebryany. Continuous fuzzing with libFuzzer and AddressSanitizer. In Proc. of the 2016 IEEE Cybersecurity Development (SecDev), 2016, p. 157.

[6] A. Fioraldi, D. Maier et al. AFL++: combining incremental steps of fuzzing research. In Proc. of the 14th USENIX Workshop on Offensive Technologies (WOOT 20), 2020, 12 p.

[7] J.C. King. Symbolic execution and program testing. Communications of the ACM, vol. 19, issue 7, 1976, pp. 385-394.

[8] J. Salwan. Triton Under the Hood. 2015, URL: https://shell-storm.org/blog/Triton-under-the-hood/#8.

[9] N. Stephens, J. Grosen et al. Driller: augmenting fuzzing through selective symbolic execution. In Proc. of the Network and Distributed System Security Symposium, volume 16 of number 2016, 16 p.

[10] I. Yun, S. Lee et al. QSYM: a practical concolic execution engine tailored for hybrid fuzzing. In Proc. of the 27th USENIX Security Symposium, 2018, pp. 745–761.

[11] R. Baldoni, E. Coppa et al. A survey of symbolic execution techniques. ACM Computing Surveys, vol. 51, issue 3, 2018, Article no. 50, 39 p.

[12] A. Vishnyakov, A. Fedotov et al. SYDR: cutting edge dynamic symbolic execution. In Proc. of the 2020 Ivannikov ISPRAS Open Conference (ISPRAS), 2020, pp. 46-54.

[13] S. Poeplau and A. Francillon. Symbolic execution with SymCC: don't interpret, compile! In Proc. of the 29th USENIX Security Symposium, 2020, pp. 181-198.

[14] S. Poeplau and A. Francillon. SymQEMU: compilation-based symbolic execution for binaries. In Proc. of the 2021 Network and Distributed System Security Symposium, 2021, 18 p.

[15] L. Borzacchiello, E. Coppa, and C. Demetrescu. FUZZOLIC: mixing fuzzing and concolic execution. Computers & Security, vol. 108, 2021, article no. 102368.

[16] D. Kuts. Towards symbolic pointers reasoning in dynamic symbolic execution. In Proc. of the 2021 Ivannikov Memorial Workshop (IVMEM), 2021, pp. 42-49.

[17] J. Chen, J. Wang et al. JIGSAW: Efficient and Scalable Path Constraints Fuzzing. In Proc. of the 2022 IEEE Symposium on Security and Privacy (SP), 2022, pp. 1531-1531.

[18] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Proc. of the 8th USENIX Conference on Operating Systems Design and Implementation, 2008, pp. 209-224.

[19] S.K. Cha, T. Avgerinos et al. Unleashing Mayhem on binary code. In Proc. of the 2012 IEEE Symposium on Security and Privacy, 2012, pp. 380-394.

[20] Google sanitizers. URL: https://github.com/google/sanitizers.

[21] Y. Chen, P. Li et al. SAVIOR: towards bug-driven hybrid testing. In Proc. of the 2020 IEEE Symposium on Security and Privacy, 2020, pp. 1580-596.

[22] S. Österlund, K. Razavi et al. ParmeSan: sanitizer-guided greybox fuzzing. In Proc. of the 29th USENIX Security Symposium, 2020, pp. 2289-2306.

[23] T. Wang, T. Wei et al. IntScope: automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In Proc. of the Network and Distributed System Security Symposium, 2009, 14 p.

[24] R. Demidov, A. Pechenkin, and P. Zegzhda. Integer overflow vulnerabilities detection in software binary code. In Proc. of the 10th International Conference on Security of Information and Networks, 2017, pp. 101-106.

[25] A. Niemetz and M. Preiner. arXiv: 2006.01621, 2020, 2 p.

[26] A. Vishnyakov, V. Logunova et al. Symbolic security predicates: hunt program weaknesses. In Proc. of the 2021 Ivannikov ISPRAS Open Conference (ISPRAS), 2021, pp. 76-85.

[27] 2021 CWE top 25 most dangerous software weaknesses. URL: https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html.

[28] G. Savidov and A. Fedotov. Casr-Cluster: crash clustering for Linux applications. In Proc. of the 2021 Ivannikov ISPRAS Open Conference (ISPRAS), 2021, pp. 47-51.

[29] FreeImage project. URL: https://freeimage.sourceforge.io/.

[30] FreeImage: Integer overflow errors in differnet places. URL: https://sourceforge.net/p/freeimage/bugs/347/.

[31] xlnt project. URL: https://github.com/tfussell/xlnt.

[32] xlnt: Integer overflow in compound_document::read_sector. URL: https://github.com/tfussell/xlnt/issues/616.

[33] xlnt: Integer Overflow in compound_document::read_sector leads to Heap Buffer Overflow. URL: https://github.com/tfussell/xlnt/issues/626.

[34] xlnt: Segmentation fault in xlnt::detail::compound_document::follow_chain(). URL: https://github.com/tfussell/xlnt/issues/595.

[35] Unbound project. URL: https://github.com/NLnetLabs/unbound.

[36] Unbound: Integer Overflow in sldns_str2period function. URL: https://github.com/NLnetLabs/unbound/issues/637.

[37] HDF4 library. URL: https://support.hdfgroup.org/products/hdf4/whatishdf.html.

[38] Hdp from hdf4-tools division by zero. URL: https://bugs.launchpad.net/ubuntu/+source/libhdf4/+bug/1915417.

[39] Miniz library. URL: https://github.com/richgel999/miniz.

[40] PyTorch project. URL: https://github.com/pytorch/pytorch.

[41] Miniz: Fix integer overflow in header corruption check. URL: https://github.com/richgel999/miniz/pull/238.

Информация об авторах / Information about authors

Алексей Вадимович ВИШНЯКОВ – младший научный сотрудник отдела компиляторных технологий, закончил бакалавриат и магистратуру ВМК МГУ в 2020 году. Сфера научных интересов: компьютерная безопасность, жизненный цикл безопасной разработки (SDL), анализ бинарного кода, символьная интерпретация, фаззинг, автоматическое обнаружение ошибок и компиляторы.

Alexey Vadimovich VISHNYAKOV works for the Compiler Technology Department, obtained BSc degree and M.D. in the Faculty of Computational Mathematics and Cybernetics at Lomonosov Moscow State University. Research interests: computer security, security development lifecycle (SDL), binary analysis, symbolic execution, fuzzing, automatic error detection, and compilers.

Илай Александрович КОБРИН работает в отделе компиляторных технологий в ИСП РАН, получает степень бакалавра ВМК МГУ. Сфера научных интересов: компьютерная

безопасность, анализ бинарного кода, символьная интерпретация, фаззинг, операционные системы.

Eli Aleksandrovich KOBRIN works for the Compiler Technology Department at ISP RAS. He is a BSc student in the Faculty of Computational Mathematics and Cybernetics at Lomonosov Moscow State University. Research interests: computer security, binary analysis, symbolic execution, fuzzing, operating systems.

Андрей Николаевич ФЕДОТОВ – старший научный сотрудник отдела компиляторных технологий, закончил НИЯУ МИФИ в 2013 году, кандидат технических наук с 2017 года. Сфера научных интересов: информационная безопасность, символьная интерпретация, оценка критичности ошибок, обратная инженерия, поиск ошибок, языки программирования, динамический анализ.

Andrey Nikolaevich FEDOTOV works for the Compiler Technology Department, graduated from National Research Nuclear University MEPHI (Moscow Engineering Physics Institute) in 2013. Obtained PhD degree in 2017. Research interests: information security, symbolic execution, error severity estimation, reverse engineering, error search, programming languages, dynamic analysis.