НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»
ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ
Кафедра інформаційної безпеки

До захисту допущено
В.о. завідувача кафедри
_____Микола ГРАЙВОРОНСЬКИЙ
(підпис)
“_____”_____2021 р.

Дипломна робота

на здобуття ступеня бакалавра
за освітньо-професійною програмою «Системи, технології та математичні
методи кібербезпеки»
спеціальності: 125 «Кібербезпека»

на тему: «Аналіз бінарних файлів створених на основі GraalVM Native Image»

Виконала: студентка 4-го курсу, групи ФБ-72
(шифр групи)

Жолоб Тетяна Сергіївна
(прізвище, ім’я, по батькові) (підпис)

Керівник к.т.н., доцент Родіонов А. М.
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант _____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище, ініціали) (підпис)

Рецензент _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповіднихпосилань.
Студент _____
(підпис)

Київ - 2021 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»
ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ

Кафедра інформаційної безпеки

Рівень вищої освіти – перший (бакалаврський)
Спеціальність – 125 «Кібербезпека»
Освітньо-професійна програма «Системи, технології та математичні методи кібербезпеки»

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ Микола ГРАЙВОРОНСЬКИЙ
(підпис)
«__»_____2019 р.

ЗАВДАННЯ
на дипломну роботу здобувачу вищої освіти

Жолоб Тетяна Сергіївна
(прізвище, ім'я, по батькові)

1. Тема роботи

"Аналіз бінарних файлів створених на основі GraalVM Native Image", науковий керівник роботи Родіонов Андрій Миколайович, к.т.н., доцент.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «_____» 2021 р. № _____

2. Термін подання студентом роботи 07 червня 2021 р.

3. Вихідні дані до роботи
   1. Мова програмування Java.
   2. GraalVM Native Image

3. Зміст роботи
   1. Огляд традиційного способу створення бінарного виконуваного файлу
   2. Дослідження створення Native Image, його особливості
   3. Дослідження методів аналізу бінарних виконуваних файлів
   4. Створення застосунків за допомогою GraalVM Native Image
   5. Аналіз створених застосунків

4. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо)

5. Дата видачі завдання 10.10.20.

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів дипломної роботи	Примітка
1	Формулювання теми дипломної роботи, визначення мети та постановка задач.	10.10.20-31.10.20	виконано
2	Узгодження теми з дипломним керівником.	20.11.20-27.11.20	виконано
3	Визначення структури дипломної роботи.	26.01.20-08.02.21	виконано
4	Робота над першим розділом: аналіз існуючих способів обробки коду, в форму зрозумілу комп'ютеру, а також дослідження нового методу створення виконуваних файлів на основі Java.	01.03.21-31.01.21	виконано
5	Робота над другим розділом: дослідження різних методів аналізу виконуваних файлів.	12.04.21-1.05.21	виконано
6	Проведення аналізу бінарних виконуваних файлів	2.05.21-10.05.21	виконано
7	Аналіз отриманих результатів	11.05.21-21.05.21	виконано
8	Оформлення текстової таграфічної документації	21.05.21-01.06.21	виконано
9	Попередній розгляд дипломної роботи на кафедрі	01.06.21	виконано

Студент _____ Тетяна ЖОЛОБ
(підпис) (ініціали, прізвище)

Науковий керівник роботи_____ Андрій РОДІОНОВ
(підпис) (ініціали, прізвище)

РЕФЕРАТ

Робота складається з 3 розділів, містить 8 ілюстрацій, 20 літературних посилань, обсяг роботи - 47 сторінок.

Завданням роботи є створення бінарних виконуваних файлів за допомогою GraalVM Native Image на основі Java, дослідження їх побудови та роботи, а також аналіз статичних блоків ініціалізації та бінарної структури виконуваного файлу на вміст потенційно небезпечної інформації.

Мета цієї дипломної роботи полягає в дослідженні способів запуску програм, а також дослідженні побудови власних зображень за допомогою GraalVM та їх аналіз.

Об’єктом дослідження є технологія GraalVM створення нативних зображень та самі виконувані файли створені за допомогою цієї технології.

Предметом дослідження є бінарні виконувані файли створені за допомогою GraalVM Native Image.

Актуальність роботи зумовлюється тим, що з кожним днем інформаційний світ вимагає швидшу роботу від програм, але водночас економії ресурсів затрачених на їх виконання, саме цім вимогам і відповідають додатки, створені GraalVM Native Image.

Методами дослідження є аналіз інформаційних джерел, новітніх публікацій за темою дослідження, нечітке моделювання, системний підхід та експериментальні дослідження із використанням сучасного програмного забезпечення.

Наукова новизна зумовлюється тим, що отриманий в роботі аналіз виконуваних файлів створених за допомогою GraalVm допоможе зрозуміти надійність цього засобу для подальшого впровадження його в різні сфери інформаційних технологій.

Ключові слова: GraalVM Native Image, Java, інтерпретація, компіляція, бінарний аналіз, Ida Pro.

ABSTRACT

The work consists of 3 sections, contains 8 illustrations, 20 literary references, the volume of work - 47 pages.

The task is to create binary executable files using GraalVM Native Image based on Java, study their construction and operation, as well as analysis of static blocks of initialization and binary structure of the executable file for the content of potentially dangerous information.

The purpose of this thesis is to study ways to run programs, as well as to study the construction of your own images using GraalVM and their analysis.

The object of research is GraalVM technology for creating native images and the executable files themselves are created using this technology.

The subject of the study are binary executable files created using GraalVM Native Image.

The urgency of the work is due to the fact that every day the information world requires faster work from programs, but at the same time saving resources spent on their implementation, it is these requirements and meet the applications created by GraalVM Native Image.

Research methods are the analysis of information sources, the latest publications on the research topic, fuzzy modeling, systems approach and experimental research using modern software.

The scientific novelty is due to the fact that the analysis of executable files created with GraalVm obtained in the work will help to understand the reliability of this tool for its further implementation in various fields of information technology.

The practical application is that the results of the work can be used to further improve this system so that it can be used in the future.

Keywords: GraalVM Native Image, Java, interpretation, compilation, binary analyse, Ida Pro

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

API - Application Programming Interface

AOT - Ahead-of-Time

EOM - електронно-обчислювальна машина

IR - Intermediate Representation

IDA - Interactive DisAssembler

JVM - Java Virtual Machine

JDK-Java Development Kit

PE - Portable Executable

LLVM - Low Level Virtual Machine

RISC - reduced instruction set computer

VM - Virtual Machine

SSA - Singular spectrum analysis

ВСТУП

В даний час ми не можемо уявити наше життя без технологій, з кожним роком вони розвиваються все більше і потребують все більшої швидкості виконання різних задач, так як масштаби цих задач стають все більші і більші, але водночас постає задача в зменшенні ресурсів потрібних на виконання цих задач. Так все більш популярні стають хмарні обчислення, так як вони мають велику кількість переваг для роботи з даними. Хмарні обчислення необхідні організаціям не залежно від типу, розміру та напрямку. Хмару можна використовувати для різних цілей, включаючи резервне копіювання даних, аварійне відновлення, розробку та тестування ПО, аналіз більших даних, для системи електронної майже, віртуальних робочих столів, а також Інтернет-додатки, орієнтовані на клієнтів.

Так як хмарні функції потребують частий запуск невеликих автономних програм, великий обсяг пам’яті значно впливають на витрати на хмарний хостинг, а повільний запуск може порушити домовленості про рівень обслуговування. Для таких програм Java GraalVM Native Image пропонує швидкий запуск та стабільну роботу.

GraalVM Native Image використовує нове ітераційне застосування точкового аналізу та моментального знімка купи, за яким слідує попередня компіляція з оптимізуючим компілятором. Код ініціалізації може працювати під час побудови, тобто виконувані файли можуть бути адаптовані до конкретної конфігурації програми. Саме виконання починається з попередньо заповненої купи, використовуючи спільне використання пам'яті копіювання на запис. Такий підхід покращує продуктивність запуску до двох порядків порівняно з Java HotSpot VM, зберігаючи при цьому пікову продуктивність.

Так як цей спосіб створення застосунків є новий, а під час розробки будь-якого програмного забезпечення не можливо повністю прорахувати всі нюанси, велика імовірність того, що в цій технології також є присутні вади, які в свою чергу можуть призвести до серйозних наслідків. Дефекти системи

можуть нанести непоправних втрат компаніям, які їх використовують і нанести удар на безпеку системи. Тому постає здача в попередньому аналізі цієї системи, виявленню цих багів і виправленні.

Завданням роботи є створення бінарних виконуваних файлів за допомогою GraalVM Native Image на основі Java, дослідження їх побудови та роботи, а також аналіз статичних блоків ініціалізації та бінарної структури виконуваного файлу на вміст потенційно небезпечної інформації.

Мета цієї дипломної роботи полягає в дослідженні способів запуску програм, а також дослідженні побудови власних зображень за допомогою GraalVM та їх аналіз.

Об'єктом дослідження є технологія GraalVM створення нативних зображень та самі виконувані файли створені за допомогою цієї технології.

Предметом дослідження є бінарні виконувані файли створені за допомогою GraalVM Native Image.

Актуальність роботи зумовлюється тим, що з кожним днем інформаційний світ вимагає швидшу роботу від програм, але водночас економії ресурсів затрачених на їх виконання, саме цім вимогам і відповідають додатки, створені GraalVM Native Image.

Методами дослідження є аналіз інформаційних джерел, новітніх публікацій за темою дослідження, нечітке моделювання, системний підхід та експериментальні дослідження із використанням сучасного програмного забезпечення.

Наукова новизна зумовлюється тим, що отриманий в роботі аналіз виконуваних файлів створених за допомогою GraalVm допоможе зрозуміти надійність цього засобу для подальшого впровадження його в різні сфери інформаційних технологій.

Практичне застосування полягає в тому, що результати роботи можуть бути використані подальшому покращенні цієї системи, щоб в майбутньому можна було її використовувати.

1 ОГЛЯД РІШЕНЬ СТВОРЕННЯ БІНАРНИХ ВИКОНУВАНИХ ФАЙЛІВ

1.1 Огляд історії створення трансляторів

У наш час комп’ютерні технології стали невід’ємною частиною нашого життя, вони використовуються всюди і дуже стрімко розвиваються. Оскільки комп’ютери це двійкові машини, які розпізнають нулі і одиички, то і керувати ними потрібно на їх машинній мові, тобто прописувати команди в виді двійкових цифр – бітів, які можна представити як в механічному так і в електричному вигляді. Кожну команду машинної мови напряму виконував електронний пристрій. Дані команди записували в цифровому вигляді, в двійковій чи шістнадцятковій системі числення. Зрозуміти таку програму було надто важко, крім того, навіть не велика програма складалась з великої кількості рядків коду. Ситуація також ускладнювалась тим, що кожна обчислювальна машина розуміє лише свій машинний код.

Людям, на відміну від машин, більш зрозумілі слова, ніж набори цифр. Прагнення людини оперувати словами, а не цифрами призвело до появи асемблерів. Це мови, в яких замість чисельного позначення команд і областей пам'яті використовуються словесно-буквені.

При переході від машинних кодів до мов асемблера отримали відразу кілька корисних ефектів. По-перше, за рахунок зручності для людини підвищилася продуктивність праці, скоротився терміни розробки. По-друге, підвищилась надійність і якість створюваних програм за рахунок меншої кількості можливостей внесення помилок в програму.

При цьому з'являється проблема: машина не в змозі розуміти слова. З’явилась потреба в якомусь перекладачеві на її рідну машинну мову. Тому, починаючи з часів асемблерів, під кожну мову програмування створюються транслятори. Це спеціальні програми, що здатні побудувати програму в

машинному коді по представленому на її вхід тексту з мнемонічними позначеннями команд.

У той же час при вирішенні задачі програміст все одно змушений мислити в термінах машини. Іншими словами, асемблер, як і машинний код, залишається мовою так званого низького рівня. Слід також зазначити ще одну важливу обставину. Програми на асемблері не були багатоплатформними - при зміні використовуваної ЕОМ (наприклад, покупці в іншого виробника, більш потужної і надійної, з іншим набором машинних команд) програми, розроблені з використанням команд попередньої машини, ставали практично марними і все треба було переписувати заново.[6]

Тому ці недоліки і спонукали створити язики програмування високого рівня. В них характерні для машини поняття, такі як комірки пам'яті або простіші операції знаходження суми чисел у комірках пам'яті, замінювались абстрактними змінними і досить складними виразами, схожими на ті, що використовуються в математичних формулах. Більше того, програми на мовах високого рівня стали частково асимільованими до інших пристроїв. Для того щоб програму можна було використовувати на різних ЕОМ, досить було наявності для неї транслятора, який перекладає код з відповідної мови високого рівня в машинний код. На даний момент є дві різновидності трансляторів: компілятор і інтерпретатор.

1.2 Порівняння компілятора і інтерпретатора

Ми вже не можемо уявити світ програмування без цих двох понять. Так а в чому ж полягає їх суттєва відмінність? Різниця між ними полягає в наступному. Компілятор приймає на вхід всю програму на мові високого

рівня цілком і в результаті процесу трансляції будує так званий об'єктний модуль, що містить машинний код, зрозумілий процесору цільової ЕОМ. Інтерпретатор перекладає програму на мові високого рівня через підрядник, при цьому для кожного рядка вихідної програми створює деяке внутрішнє уявлення на спеціальній проміжній мові, яке направляється спеціальній машині часу виконання, або віртуальній машині (Virtual Machine). Ця машина негайно виконує отримане розпорядження.

І у компіляторів, і у інтерпретаторів є переваги і недоліки. До переваг компілятора відноситься в першу чергу те, що отриманий машинний код може безпосередньо виконуватися на даній ЕОМ, при цьому наявності компілятора в момент виконання не потрібно. Виконання ж відбувається на максимально можливій швидкості. Однак в разі необхідності внесення в програму зміни, навіть мінімальної, буде потрібно провести перекомпіляцію вихідного тексту повністю, навіть якщо він налічує десятки тисяч рядків!

Інтерпретатор перекладає програму порядково, тому він зручний для налагодження програми, коли постійно відбуваються уточнення і зміни і потрібно швидко побачити, як поведе себе відкоригована програма. До недоліків інтерпретатора відноситься те, що, оскільки програма виконується інтерпретатором, а не безпосередньо процесором, по-перше, швидкість виконання програми істотно нижче, ніж при компіляції (зазвичай в 10-20 разів), а по-друге, потрібна наявність машини часу виконання в пам'яті.

У той же час скомпільована програма в машинному коді аналогічно програмі на асемблері - може виконуватися лише на архітектурно сумісній ЕОМ. А для інтерпретованої мови (наприклад, Java) вистачить установки на даний комп'ютер віртуальну машини виконання (у разі Java - JVM (Java Virtual Machine)), після чого на ній теоретично можуть бути виконані будь-які програми на даній інтерпретованій мові (гасло Java - Write Once Run Everywhere - «написавши раз, запускаєш скрізь!»).[6]

1.3 Процес компіляції

На перший погляд компілятор представляє собою моноліт, який обробляє вихідний код програми, а потім видає виконуваний файл, або просто виконуваний байт-код, який потім може бути успішно виконаний віртуальною машиною. Але насправді компілятор виконує набагато складніші операції, які складаються з декількох частин, які повинні злагоджено працювати між собою в певному порядку.

Більшість компіляторів мають наступну архітектуру Рисунок 1.1:

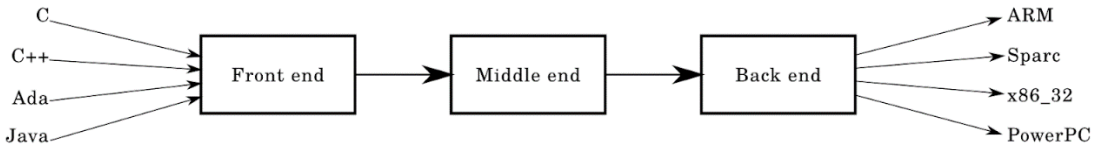

Рисунок 1.1 – Архітектура компілятора

В свою чергу кожен з етапів містить свої частини. До фронтенд відносять лексичний аналізатор або сканер та синтаксичний аналізатор або парсер.
До середнього етапу відносять семантичний аналізатор. До бекенд – один або кілька генераторів коду і один або кілька оптимізаторів. Також до компілятору часто відносять додаткові інструменти, потрібні для створення виконуваного файлу - складальник і компонувальник. [7]

1.3.1 Лексичний аналіз

Перший етап розбору вихідного тексту програми, що здійснюється компілятором, - це лексичний аналіз. Лексичний аналізатор зчитує послідовно всі слова (токени, лексеми) в тексті програми, перетворюючи їх в конструкції, які потім вже використовуються для подальшого розбору тексту. Робиться це для того, щоб розрізнити в подальшому різні ідентифікатори і безпосередні

конструкції самої мови (такі, як зарезервовані слова). Зазвичай аналізатор являються найпростішим компонентом компілятора.[4]

Слідом за лексичним аналізом може бути препроцесор. Основне завдання препроцесора - це заміна одних лексем іншими, які були заздалегідь визначені в тексті програми. Використовується препроцесор також для умовної компіляції (тобто коли шматок коду повинен бути відкомпільований тільки при виконанні певних умов - для певної платформи, тільки при відладочному білді і т.д.), для виконання певних макросів (як в тому ж C / C ++) і деяких інших подібних речей. Препроцесор не є обов'язковою частиною компілятора, оскільки багато мов програмування не потребують його.

1.3.2 Парсинг

Наступний етап - це синтаксичний аналіз, або парсинг. Цей етап компіляції виконується синтаксичним аналізатором, або парсером, і є, мабуть, найважливішим і, якщо можна так сказати, відповідальним етапом компіляції. Компілятор розглядає всі токени, і, в залежності від їх значення і положення в тексті програми, формує так зване дерево розбору. Тобто програма, що була до цього в надрах компілятора просто лінійним набором символів, стає деревом, елементи якого розташовані відповідно до граматики тієї мови програмування, для якої написаний даний конкретний компілятор.

1.3.3 Семантичний аналіз

Слідом за синтаксичним аналізом слідує етап семантичного аналізу. Якщо синтаксичний аналізатор будував скелет нашої програми, то семантичний допомагає цьому скелету обрости плоттю. Програма

наповнюється сенсом: змінні стають змінними, об'єкти - об'єктами, а баги - багами. Насправді, ніякого чаклунства не відбувається - просто дерево розбору, терпляче побудоване парсером, доповнюється семантичною інформацією про значення ідентифікаторів. Компілятор пройдеться по всіх оголошеннях «вищого рівня» в модулях і усвідомить їх існування. Глибше в блоки він не піде - він просто оголосить, які структури, функції і т.д. є в тому чи іншому модулі. Також на цьому етапі виникає і багато помилок компіляції - наприклад, такі, як невідповідність типів. Компілятор проходить по всім блокам коду в функціях і т.д. і дозволяє їх - тобто, знаходить символи, що вимагають дозволу. Хоча, звичайно, на парсинг теж доводиться чимало помилок, без яких, на жаль, текст свіжонаписаної програми обходиться вкрай рідко, навіть у дуже досвідчених програмістів.[4]

На наступному етапі компілятор проходить по всім блокам і функціям і перевіряє, що їх «змінні коректні», тобто ми не змінюємо те, що не слід, і що всі змінні, що передаються до певних функцій, є постійними або змінним там, де потрібно.

Це робиться за допомогою символьної інформації, яка зібрана за попередні проходи. Символьна таблиця, побудована за результатами семантичного проходу, містить імена токенів і ознаки змінності змінних. Вона може містити й інші дані, наприклад, в C ++ в таблиці може зберігатися інформація про те, є символ зовнішнім або статичним.

Символьна таблиця, або «stab», це таблиця для пошуку символів, які використовуються у вашій програмі. Для кожної області видимості створюється по одній таблиці, і всі вони містять інформацію про символи, присутніх в конкретній області видимості.

До цієї інформації належать такі властивості, як ім'я символу, тип, ознака змінності, наявність зовнішнього зв'язку, розташування в статичної пам'яті та інше.

1.3.4 Завершальний етап

Далі шляхи різних компіляторів розходяться. У більшості компіляторів слідом за етапом семантичного аналізу йде переклад програми в певний проміжний код, який може використовуватися для генерації коду під різні апаратні платформи. Якщо компілятор виконує компіляцію тільки для якоїсь однієї апаратної платформи, то програма може транслюватися в коди на мові Асемблера відповідної процесорної архітектури, або, якщо компілятор працює для якоїсь віртуальної машини (як, наприклад, в разі Java або Microsoft .NET), то переводитися програма може потім в спеціальний байт-код, зрозумілий відповідної віртуальної машині. [4] Проте, в більшості сучасних компіляторів немає безпосередньої трансляції в асемблерний код - навіть якщо в результаті компілятор не повинен намагатися для створення крос-платформних програм, все одно, спочатку йде трансляція програми в якийсь проміжний код, а тільки потім вже в виконуваний. Причина цього в оптимізації коду.

1.3.5Оптимізація

Сучасні компілятори, навіть самі слабкі, підтримують хоча б базову оптимізацію коду. Більш потужні комерційні компілятори містять в собі дуже потужні алгоритми оптимізації коду, які дозволяють при необхідності зробити її в рази швидше. Особливо потужними в плані оптимізації колись вважалися компілятори виробництва Watcom, які зараз, поступово відновлюють свою колишню славу у вигляді open-source продукту. Потім пальма першості перейшла компіляторам Intel, і зараз саме вони вважаються найкращими компіляторами в плані оптимізації. Втім, не важливо, погана оптимізація в компіляторі чи ні - головне, що в будь-якому оптимізуючому компіляторі є модуль, званий оптимізатором, який починає свою роботу після генератора

проміжного коду. Справедливості заради варто сказати, що оптимізатор може працювати і після генерації вже виконуваного коду, але в наші дні така схема зустрічається вже рідко, оскільки виробники компіляторів, як правило, випускають цілу лінійку подібних продуктів для різних мов програмування і намагаються робити оптимізатори, які можна вбудувати в будь-який з цих компіляторів.[4]

У будь-якому випадку, робота компілятора закінчується генерацією виконуваного коду. Це може бути код віртуальної машини або код на мові асемблера, але цей код вже придатний для виконання компільованою програми.

1.3.6 Етап побудови бінарного виконуваного файлу

Як було вказано раніше, за етапом семантичного аналізу може слідувати етап , де програма може транслюватися в коди на мові Асемблера відповідної процесорної архітектури, як наприклад у випадку таких мов програмування, як C чи C++. Наступним кроком у процесі компіляції є збирання отриманого коду збірки у файл об’єкта. Результатом цього етап являється об’єктний файл, який містить коди операцій та різних даних. [3]

Компонування – це завершальний етап, який створює виконуваний файл програми, під час цього об’єктний файл, згенерований компілятором, компонується з іншими об’єктними кодами, в тому числі і бібліотеками для отримання виконуваного файлу. [2] На цьому етапі виконується об'єднання всіх об'єктних файлів проекту, відкомпільованих за відповідними компіляційними листам проекту в єдину сутність. Це може бути додаток, статична або динамічна бібліотека. Різниця в бінарних заголовках цільових файлів і кілька різної внутрішньою організацією. Первинним завданням компонування слід назвати задачу по підстановці адрес виклику зовнішніх

об'єктів, які були утворені в об'єктних файлах проекту. Відповідні реалізації сутностей з адресами їх розміщення повинні знаходиться в видимості компонування. Ці сутності повинні бути або в об'єктних файлах, тоді вони повинні бути вказані в списку компонування, або в зовнішніх бібліотеках функцій, статичних або динамічних, тоді вони повинні бути вказані в списку зовнішніх бібліотек. [1]

На цьому етапі найпоширенішими помилками є відсутність визначень або повторювані визначення. Перше означає, що або визначення не існують (тобто вони не написані), або файли об’єктів або бібліотеки, де вони перебувають, не були надані компонуванню. Останнє очевидно: один і той же символ був визначений у двох різних об’єктних файлах або бібліотеках. [5]

Також під час компонування виконується і декілька додаткових корисних завдань. Програма поєднується з деякими стандартними процедурами, необхідними їй. Наприклад, на початку програми потрібен стандартний код, який встановлює робоче середовище, наприклад, передача параметрів командного рядка та змінних середовища. Крім того, є код, який потрібно запустити в кінці програми, щоб він міг передати назад код, а саме результат виконання програми. [3] Після закінчення генерації коду компілятор виділяє пам'ять для коду та даних у розділах; кожен розділ має різну інформацію і визначається назвою або атрибутами інформації, що зберігається в них.

1.4 GraalVM Native Image

Native Image – це застосунок для перетворення програм Java у повністю скомпільований двійковий код, який називається нативним образом або власним зображенням. Це новий підхід який зменшує час запуску та обсяг пам’яті: поєднання точкового аналізу, ініціалізації додатків під час побудови, моментального знімку купи та компіляції AOT(Ahead-of-Time). Це зберігає переваги екосистеми Java: поєднуючи існуючі бібліотеки, які слабко пов’язані

та розширюють одна одну, використовуючи, наприклад, інтерфейси та декларативні файли конфігурації. Система базується на припущенні про закритий світ, тобто всі класи Java повинні бути відомими та доступними під час побудови. Під час побудови додаток Java та всі його бібліотеки (включаючи JDK) обробляються точковим аналізом, для того щоб знайти доступні елементи програми (класи, методи та поля), в результаті чого компілюються лише необхідні методи класів Java .

Код ініціалізації програми може працювати під час побудови, а не під час виконання. Додаток може контролювати те, що ініціалізується, і виділяти об'єкти Java для побудови складних структур даних. Ці об'єкти доступні під час виконання за допомогою так званої купи зображень, попередньо ініціалізованої частини купи, яка є частиною виконуваного файлу та доступна відразу під час запуску програми. Точковий аналіз може зробити об'єкти доступними в купі зображення, а знімок, що створює купу зображень, може зробити нові методи доступними для точкового аналізу. Ці кроки виконуються ітеративно, поки не буде досягнута фіксована точка. З Native Image лише декілька кроків ініціалізації виконуються, перш ніж викликати основний метод програми, наприклад, відображення пам'яті купи зображення. Кілька процесів, що запускають один і той самий виконуваний файл, або кілька ізольованих екземплярів VM у межах одного процесу, використовують спільний доступ до копії на запис купи зображення. Це зменшує загальний обсяг пам'яті при запуску декількох екземплярів однієї програми.

1.4.1 Принципи роботи Native Image

Вхідними даними Native Image повинен бути байт-код Java, скомпільований з будь-якої мови, яка компілюється в байт-код Java, такий як Java, Scala або Kotlin. Додаток, його бібліотеки, компоненти JDK та VM обробляються однаково. Результатом є власний виконуваний файл для конкретної операційної системи та архітектури. Такий виконуваний файл називають власним образом.

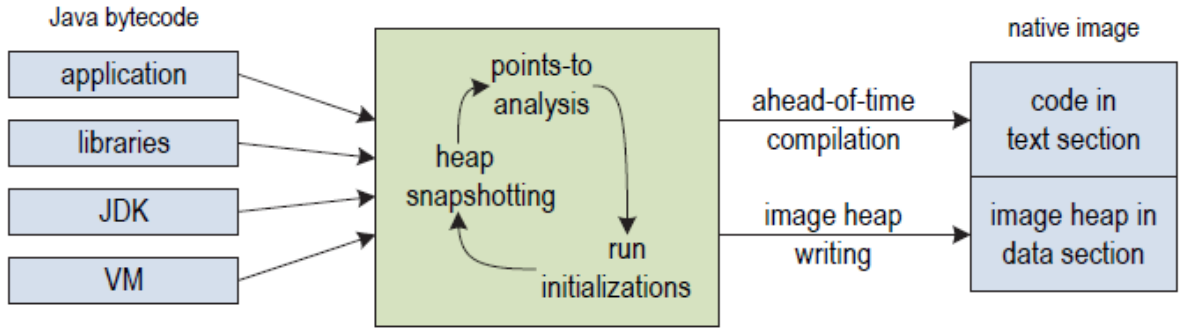

Рисунок 1.2 – Загальна структура побудови системи

Рисунок 1.2 показує загальну структуру системи часу побудови. По-перше, точковий аналіз та моментальний знімок купи виконуються ітеративно, поки не буде досягнута фіксована точка. Зворотні дзвінки, зареєстровані програмою, також виконуються як частина цього ітеративного підходу, тобто програма може брати участь.

Результатом цього етапу є перелік доступних класів, методів та полів; а також граф досяжних об’єктів. Потім доступні методи компілюються в машинний код, а граф об’єктів представляється як купа зображень у тому ж макеті, який використовується під час виконання для купи. Машинний код зберігається в текстовому розділі власного зображення, а купа зображень - у розділі даних.

Весь процес, що створює власне зображення, називають часом побудови зображення, щоб чітко відрізнити його від компіляції вихідного коду Java до байт-коду (часто називається “compile time” або “build time”).

Конструктор зображень - це програма Java. Та сама віртуальна машина Java, яка виконує точковий аналіз і компіляцію AOT, також запускає ініціалізатори класу та зворотні виклики ініціалізації програми. Це означає, що під час побудови зображення об’єкти, які згодом формують купу зображень, є звичайними об’єктами у купі Java конструктора зображень. Моментальний знімок купи виявляє об’єкти, доступні для купи зображення.

Однією з переваг цієї системи є розмита межа між часом збірки та часом виконання: легко переміщати ініціалізації між часом збірки та часом виконання без великих змін коду, тобто можна написати програму, яка може підтримувати обидві конфігурації на час побудови та конфігурація під час виконання.

Точковий аналіз

Ми використовуємо точковий аналіз, щоб визначити, які класи, методи та поля доступні під час виконання. Він починається з усіх точок входу, наприклад, основного методу програми, і ітераційно обробляє всі транзитивно доступні методи, поки не буде досягнута фіксована точка. Аналіз не чутливий до контексту, шляху та нечутливий до потоку полів, але чутливий до потоку локальних змінних, оскільки він базується на формі SSA [*].

Точковий аналіз використовує інтерфейс компілятора для аналізу байт-коду Java, щоб перетворити в проміжне представлення (Intermediate representation, IR) високого рівня для подальшої компіляції. Потім IR перетворюється у так званий граф потоку типів. Вузли на графі - це інструкції, що діють на типи об’єктів. Ребра - це спрямовані ребра, що використовуються між вузлами, тобто вони спрямовані від визначення до використання. Кожен вузол підтримує стан типу: список типів, які можуть досягти цього вузла, та інформацію, про її відсутність. Стани типів поширюються через ребра використання: якщо стан типу вузла змінено, зміна поширюється на всі

пов’язані вузли. Стани типів можуть лише зростати, тобто нові типи можна додавати до стану типів, але типи ніколи не видаляються. Це гарантує, що аналіз врешті-решт досягне фіксованої точки і закінчиться.

Граф потоку типів - це єдиний міжпроцедурний граф, який охоплює всю програму. Для викликів статичних методів викликаючий та викликаний зв’язуються, коли створюється граф потоку типів: задіяне ребро з’єднує фактичний вузол аргументу із відповідним формальним вузлом аргументу, тобто типи вливаються в методи, що використовують вузли аргументів. Вузол повернення методу використовує ребро до вузла виклику у абонента, тобто типи витікають із методів за допомогою вузлів повернення. Для викликів віртуальних та інтерфейсних методів, виклик та викликані методи динамічно зв’язуються під час запуску аналізу: стан типу одержувача визначає, як виклик можна отримати. Коли новий тип додається до стану типу приймача, новий тип використовується для вирішення виклику. Це гарантує правильне розповсюдження типів за допомогою викликів віртуальних методів, але лише необхідні виклики пов'язані. Точковий аналіз відстежує тип, якщо він інстанційований. Байт-коди розподілу позначають тип як екземпляр і є одними із джерел типів у графі потоку типів (вузли потоку типу без попередника). Для кожного поля точковий аналіз відстежує, чи поле читається чи записується. Читання та запис відстежуються окремо, щоб знайти поля, які лише читаються, але ніколи не записуються під час виконання. Такі поля згодом постійно складаються під час компіляції AOT.

Запуск коду ініціалізації

Коли точковий аналіз досягає локальної фіксованої точки (більше типів не додається до станів типів), виконується код ініціалізації. Він походить з двох різних джерел: По-перше, виконуються ініціалізатори класу. У Java

кожен клас може мати ініціалізатор класу (іноді його також називають “static initializer”), який представлений у вигляді методу <clinit> у файлі класу. Він обчислює початкове значення статичних полів. Розробнику дозволяється вирішувати, які класи ініціалізуються під час побудови зображення, а які класи залишаються неініціалізованими під час побудови зображення, а потім ініціалізуються під час виконання. По-друге, програма може реєструвати явні зворотні виклики, які викликаються під час побудови. Програма може запускати власний код, наприклад, до, під час та після етапу аналізу, реалізуючи хуки, надані інтерфейсом.

Відповідним сигналом для запуску коду перед моментальним знімком купи є " during analysis" сигнал: він виконується кожного разу, коли точковий аналіз доходить до локальної фіксованої точки, але не доходить до моментального знімка купи. На даний момент програма може запускати власний код, який, наприклад, розподіляє об’єкти та ініціалізує більші структури даних. Важливим моментом, який слід зауважити, є те, що код ініціалізації може отримати доступ до поточного стану точкового аналізу: він може запитувати, чи тип, метод або поле вже було позначено як доступне за допомогою точкового аналізу за допомогою різних методів isReachable (). Додаток може використовувати цю інформацію для побудови структур даних, оптимізованих для точно доступних частин програми.

Знімок купи

Знімок купи створює граф об’єктів, тобто транзитивне закриття доступних об’єктів, починаючи з кореневих покажчиків, таких як статичні поля. Цей граф об'єктів записується у власне зображення як купа зображень, тобто початкова купа при запуску рідного зображення.

Вхідні дані алгоритму - це стан точкового аналізу.

Кореневі вказівники купи зображень - це статичні поля об’єктів, які позначені точковим аналізом як прочитані, а також значення, які постійно складаються в такі методи, як наші одиночні зображення. Ці кореневі значення спочатку додаються до робочого списку, а потім робочий список обробляється, поки він не порожній. Кожен об’єкт лише один раз додається до робочого списку та обробляється. Для цього підтримується набір з усіма доступними об’єктами.

Щоб побудувати транзитивне закриття, за полями об’єкта слідують значення полів зчитування за допомогою відображення (пам’ятайте, що конструктор зображень - це програма Java). Враховуються лише поля екземпляра, які позначені як прочитані аналізом точок до, тобто, якщо клас має два поля екземпляра, але одне з них не позначене як прочитане точковим аналізом, об'єкт, доступний з цього поля, не є частиною купи зображення.

Ahead-of-Time (AOT) компіляція

Тільки методи, які позначені як доступні за допомогою точкового аналізу, AOT компілюються до машинного коду та розміщуються в текстовому розділі виконуваного файлу.

Код, який працює під час побудови зображення, часто недоступний під час виконання, а тому не компілюється.

Наприклад, ініціалізатори класів, які виконуються під час побудови зображення, не компілюються .

Однак допускається наявність коду, який працює як під час побудови зображення, так і під час виконання, наприклад, спільні методи утиліти.

Купа зображень

Виконання застосунку під час запуску починається з уже заповненої купи Java, яка створюється алгоритмом моментального знімку купи під час побудови зображення: купа зображень. Запуск виконуваного файлу повинен

бути швидким та мати низькі витрати пам’яті, щоб можна було створити багато екземплярів віртуальних машин для короткочасних завдань. Тому при завантаженні купи зображень не повинно бути дорогого кроку переміщення.

Для ізолятів повинна бути можливість зіставити купу зображень за різними адресами пам'яті в одному і тому ж адресному просторі.

Для досягнення цих цілей ми використовуємо посилання, які відносяться до початку купи зображення. Це означає, що для доступу до пам'яті, як-от завантаження поля від об'єкта чи елемента масиву, потрібно більше арифметики адрес, ніж звичайні абсолютні посилання: початок кучі повинен бути доданий до посилання перед доступом до пам'яті. Об'єкти купи зображення та об'єкти, виділені під час виконання, використовують однаковий формат, тобто також об'єкти, виділені під час виконання, використовують відносні посилання. Щоб зробити це якомога швидше, початок купи завжди доступний у фіксованому реєстрі.

Компоненти виконання

Власне зображення містить систему виконання для Java, наприклад, збирач сміття (GC); пересування стеком та обробка винятків; і підтримка потоків та синхронізації. Вся система виконання написана на Java, виявлена як доступна за допомогою точкового аналізу та скомпільована AOT.

Немає різниці між компонентами віртуальної машини, додатком, бібліотеками, що використовуються додатком, і стандартною бібліотекою Java. Як результат, компоненти віртуальної машини також отримують прибуток від оптимізації AOT на основі результатів аналізу, що вказує на точки, а компоненти віртуальної машини можуть, наприклад, бути вбудованими в код програми.

1.5Kotline Native

Kotlin Native - це сервер LLVM для компілятора Kotlin, який призначений для роботи в областях без VM. Він використовує ланцюжок інструментів LLVM для генерації реалізації та власного коду. Він в першу чергу розроблений для того, щоб дозволити компіляцію для платформ, де віртуальні машини не бажані чи можливі (наприклад, iOS та вбудовані системи), або де нам потрібно створити автономну програму, яка не вимагає додаткового виконання.

Він повністю підтримує взаємодію в межах власного коду, тобто коду мови C. Це працює подібно до того, як Kotlin / JVM взаємодіє з кодом мови Java. Для бібліотек платформи відповідні взаємодіючі бібліотеки доступні нестандартно, тоді як для інших бібліотек існує інструмент для створення взаємодіючої бібліотеки з заголовного файлу C, з повною підтримкою всіх функцій мови C. Він також підтримує взаємодію з кодом Objective / C для macOS та iOS.[8]

До переваг Kotlin Native можна віднести наступне:

- Він працює без віртуальної машини
- Він компілює рідний машинний код
- Він має здатність створювати автономний виконуваний файл
- Він крос-платформний
- Він пропонує сумісність із кодом

1.6 LLVM

Low Level Virtual Machine (LLVM) — універсальна система аналізу, трансформації і оптимізації програм, що реалізує віртуальну машину з RISC-подібними інструкціями. Може використовуватися як оптимізувальний

компілятор цього байт-коду в машинний код для різних архітектур або для його інтерпретації та JIT-компіляції (для деяких платформ).[9]

LLVM спрощує не тільки створення нових мов, але і покращує розвиток існуючих. Він надає інструменти для автоматизації багатьох найбільш невдячних частин завдання створення мови: створення компілятора, перенесення виведеного коду на кілька платформ та архітектур, генерація специфічних для архітектури оптимізацій, таких як векторизація, та написання коду для обробки загальномовних метафор, таких як винятки. Його ліберальне ліцензування означає, що його можна вільно використовувати повторно як компонент програмного забезпечення або використовувати як послугу.

У своїй основі LLVM - це бібліотека для програмного створення машинного коду. Розробник використовує API для генерації інструкцій у форматі, який називається проміжним поданням або IR. Потім LLVM може скомпілювати ІЧ у самостійний двійковий файл або виконати компіляцію JIT (точно вчасно) над кодом, який запускатиметься в контексті іншої програми, наприклад, інтерпретатора або середовища виконання мови.

API LLVM надають примітиви для розробки багатьох типових структур та шаблонів, що зустрічаються в мовах програмування. Наприклад, майже у кожній мові є поняття функції та глобальної змінної, а багато хто має інтерфейси зовнішньої функції C. LLVM має функції та глобальні змінні як стандартні елементи у своєму IR та має ресурси для створення спільних програм та взаємодії з бібліотеками C.

Висновки до розділу 1

В цьому розділі було розглянуто коротку історію створення компіляторів та інтерпретаторів починаючи з створення першої мови програмування асемблера. Також досліджено і розібрано етапи традиційного створення виконуваного файлу на основі компільованої мови програмування C++.

Native Image новий спосіб створення виконуваних файлів на основі Java, який зберігає всі переваги цієї мови програмування і набагато збільшує швидкість запуску застосунків написаних на основі цієї мови, чим самим значно покращуючи їх роботу.

В цьому розділі було розглянуто повністю кожен етап створення Native Image і особливості кожного з етапів.

2 АНАЛІЗ БІНАРНИХ ФАЙЛІВ

Сьогодні програмні компанії стикаються з серйозними ризиками, пов'язаними з безпекою програмного забезпечення, оскільки процес розробки програмного забезпечення завжди містить деякі помилки. У свою чергу, ці помилки можуть створити сприятливі умови для появи серйозних вразливостей програмного забезпечення. Використання вразливостей впливає не тільки на безпеку системи, яка використовує вразливий продукт, але і на конкурентні переваги компанії, що займаються розробкою програмного забезпечення. Слід зазначити, що процес дослідження вразливостей є дуже складним, оскільки в цьому процесі існує ряд фундаментальних технологічних проблем. Крім того, на практиці будь-яка техніка атаки на безпеку програмного забезпечення ґрунтується на попередньому розбиранні та аналізі механізму захисту двійкового коду. Для вирішення цих проблем широко використовуються методи статичного та динамічного двійкового аналізу.
Динамічний аналіз заснований на виконанні програми на центральному процесорі. У свою чергу, статичний аналіз заснований на аналізі техніки PE-модуля в пам'яті.

2.1 Статичний аналіз

Статичний аналіз коду - це процес виявлення помилок у вихідному коді програми до її виконання. Для реалізації техніки статичного аналізу, двійковий код повинен бути інтерпретований на проміжну мову.[13]
Можна виділити наступні кроки в роботі статистичного аналізатора:
1. Розбір тексту
1.1. Лексичний аналіз (текст програми розбивається на лексеми: зарезервовані слова, ідентифікатори та константи)
• виділення лексем;
• визначення, якій групі лексем, належить кожна з них;
• передача результатів далі, щоб зрозуміти;

1.2. Синтаксичний аналіз

• перевірка наявності синхронізуючих лексем: текст до синхронізуючої лексеми вважається правильним;

• локальна корекція (відбувається автоматичне додавання синхронізуючих лексем);

• розширення граматики помилками (початкова граматика доповнюється неправильними виразами і потім у потрібний момент пропонується варіант виправлення помилок);

1.3. Контекстний аналіз (перевіряється, чи допускається використання слів у даному контексті)

2. Застосування деяких правил

2.1.1. Переваги і недоліки статичного аналізу

Переваги статичного аналізу:

• Раннє виявлення помилок (під час розробки);

• Код при цьому може не збиратися і не запускатися;

• Повне покриття коду;

• Не потрібно задавати тестові вхідні дані;

• Краще перевіряє ділянки коду, складні для тестування (обробка граничних ситуацій);[13]

Недоліки статичного аналізу:

• Відсутність знань про вхідних значеннях (невизначені значення змінних).

• Експоненціальне зростання числа шляхів у програмі.

• Велика довжина окремих шляхів в програмі (цикли, рекурсивні виклики).

• Складність аналізу циклів і рекурсивних викликів

2.2 Динамічний аналіз

Динамічний аналіз коду - це спосіб аналізу програми безпосередньо при її виконанні. Звідси випливає, що з вихідного коду в обов'язковому порядку повинен бути отриманий виконуваний файл, тобто не можна таким способом проаналізувати код, що містить помилки компіляції або збірки. Динамічний аналіз виконується за допомогою набору даних, які подаються на вхід досліджуваної програмою. Тому ефективність аналізу безпосередньо залежить від якості і кількості вхідних даних для тестування. Саме від них залежить повнота покриття коду, яка буде отримана за результатами тестування.[14]

Використовуючи динамічне тестування, можна отримати наступні метрики і попередження:

- Використовувані ресурси: час виконання програми в цілому або її окремих модулів, кількість зовнішніх запитів (наприклад, до бази даних), кількість використовуваної оперативної пам'яті і інших ресурсів.
- Ступінь покриття коду тестами і інші метрики програми.

Програмні помилки: поділ на нуль, розіменування нульового показника, витоку пам'яті.

- Виявити деякі уразливості.

До основних переваг динамічного аналізу коду відносять:

- Можливість проводити аналіз програми без необхідності доступу до її вихідного коду. Тут варто зробити застереження, так як програми для динамічного аналізу розрізняють за способом взаємодії з перевіряючою програмою . Наприклад, поширений спосіб проведення динамічного аналізу шляхом попереднього інструментування вихідного коду, тобто додавання спеціального коду у вихідний текст програми для виявлення помилок. В цьому випадку доступ до коду програми, що перевіряється буде необхідний.
- Можливість виявлення складних помилок, пов'язаних з роботою з пам'яттю: вихід за межі масиву, виявлення витоків пам'яті.

- Можливість проводити аналіз багатопоточного коду безпосередньо в момент виконання програми, тим самим виявляти потенційні проблеми, пов'язані з доступом до спільних ресурсів, можливі deadlock ситуації.

У більшості реалізацій поява помилкових спрацьовувань виключено, так як виявлення помилки відбувається в момент її виникнення в програмі; таким чином, виявлена помилка є не пророкуванням, зробленим на основі аналізу моделі програми, а констатацією факту її виникнення.[14]

Перерахуємо недоліки, які притаманні динамічному аналізу коду:

- Не можна гарантувати повного покриття коду, тобто, швидше за все, відсоток коду програми, який був проаналізований в процесі динамічного тестування, далекий від ста відсотків.
- Майже не виявляються помилки логічного типу. Наприклад, з точки зору динамічного аналізатора, істина умова завжди не є помилкою, так як така некоректна перевірка просто зникає ще на етапі компіляції програми.
- Важко локалізувати місце з помилкою в вихідному коді.
- Більш висока складність використання в порівнянні зі статичним аналізом, так як для досягнення більшої ефективності динамічного аналізу тестованої програми потрібна достатня кількість вхідних даних, щоб отримати більш повне покриття коду.

Динамічне тестування найбільш важливо в тих областях, де головним критерієм є надійність програми, час відповіді або споживані ресурси. Це може бути, наприклад, система реального часу, керуюча відповідальною ділянкою виробництва, або сервер бази даних. У таких областях будь-яка допущена помилка може виявитися критичною.

2.3 Бінарний аналіз

2.3.1 Визначення

Бінарний аналіз - це тип огляду коду, який переглядає файли, що складаються з двійкового коду, та оцінює їх вміст та структуру, і все це без необхідності доступу до вихідного коду.

Деякі інструменти двійкового аналізу працюють аналогічно інспекторам диспетчера пакетів, які в основному читають “зміст” файлу, щоб з’ясувати, що знаходиться всередині. Цього базового аналізу може бути достатньо в деяких випадках, але вдосконалені інструменти двійкового аналізу можуть моделювати типи даних, потоки та шляхи керування, не потребуючи зворотного проектування.

Використовуючи цю модель, вдосконалені інструменти двійкового аналізу можуть глибше розглянути ідентифікувати відомі компоненти програмного забезпечення та виявити закономірності дефектів безпеки. Потім ці відкриття можна використовувати для складання звітів про безпеку та використання, а також порад щодо вирішення будь-яких проблем у коді.

Вихідний код не завжди доступний для аналізу. Наприклад, деякі компанії купують мікропрограми для інтеграції із апаратним забезпеченням своїх продуктів, а мікропрограми мають бінарний формат. Інший приклад - програмні компанії, які використовують сторонній код та бібліотеки, такі як фреймворки, контейнери, графічні інтерфейси та бази даних, для збільшення власного коду, і ці бібліотеки часто не містять вихідного коду. Як би там не було, все одно важливо, щоб споживачі цих двійкових файлів розуміли, що в них знаходиться.

Рішення для двійкового аналізу дозволяють організаціям перевіряти двійковий код без залучення постачальника, виявляти компоненти з відкритим кодом, уразливості системи безпеки, зобов'язання щодо ліцензії та додаткову конфіденційну інформацію, яка може призвести до порушення.[12]

Традиційно для аналізу програми в бінарному коді для розпізнавання

алгоритмів складових її функцій, з'ясування особливостей реалізації алгоритмів, виявлення недокументованих можливостей використовуються дизасемблери і засоби статичного аналізу потоків даних. Однак у випадках, коли укладачі програми вжили заходів для захисту своєї програми від аналізу (наприклад, використовували запакований код, який розпаковується під час виконання програми), статичний аналіз може не дати результатів. У таких випадках пропонується використовувати комбінований аналіз: динамічний і статичний.[18]

У бінарному коді відсутні змінні і типи даних в явному вигляді - замість цього інструкції оперують регістрами і осередками пам'яті, відсутні і явні опису прототипів функцій і їх кордонів в коді.

2.3.2 Структура виконуваного файлу

Виконуваний файл містить області коду і статичних даних програми, таблиці функцій, що імпортуються програмою з сторонніх, динамічно завантажуваних бібліотек, а також список функцій, що експортуються програмою, в разі, якщо програма сама є бібліотекою. Область коду містить послідовність функцій програми, яка не є безперервною, так як між функціями можуть виникати невикористовувані ділянки пам'яті внаслідок вирівнювання. Крім того, багато компілятори вставляють частина даних функції поряд з її кодом (статичні дані).[18]

Бібліотечні функції, які при складанні програми були статично пов'язані з нею, містяться в її коді і не відрізняються від функцій програми.

2.3.3 Етапи аналізу бінарного коду

Як було зазначено вище, при аналізі бінарного коду можуть застосовуватися ті ж методи статичного аналізу, що і для вихідного коду. Однак їх застосування вимагає попереднього отримання графа викликів функцій програми і набору графів потоку управління для кожної функції.

Деякі види аналізу потоку даних вимагають крім того побудови графа залежностей (за даними і управління). При різних аналізах потоку даних відслідковуються значення атрибутів, які зіставляються змінним, що призводить до необхідності наявності інформації про змінних, що використовуються в програмі, їх типах і інших атрибутах.[15]

Одним з найбільш поширених і розвинених засобів статичного аналізу бінарного коду є система інтерактивного дизасемблювання IDA Pro [16], що підтримує широкий ряд бінарних форматів і процесорних архітектур. При аналізі виконуваних файлів IDA Pro дозволяє проводити їх автоматичний аналіз.

Дизасемблерування виконуваного файлу

При статичному аналізі, виділення областей коду і даних, а також таблиць імпорту та експорту виконується в процесі розбору файлу - дані про зсув і розмір цих областей містяться в заголовку. Для того щоб виділити в області даних межі окремих змінних потрібно проаналізувати команди звернення (читання і запису) до цих даних з області коду. Аналогічно, для того щоб виділити межі функцій потрібно визначити їх початок, тобто адреса, на якій управління передається за допомогою спеціальної інструкції виведення, і кінець - адреса (точніше максимальна адреса, в разі якщо їх декілька) спеціальної команди повернення. Перед аналізом виконується дизасемблювання коду, тобто уявлення його у вигляді лістингу на мові асемблера цільової архітектури. Для дизассемберування потрібно відокремлювати машинні інструкції від даних (дизассемблеруванню повинні піддаватися тільки машинні інструкції). [15]

У традиційній моделі розробки програмного забезпечення компілятори, асемблери та лінкери використовуються самі по собі або в поєднанні для створення виконуваних програм. Щоб повернутися назад, ми використовуємо

інструменти для скасування процесів складання та компіляції. Не дивно, що такі інструменти називають десемблерами та декомпіляторами, і вони роблять майже те, що вказують їх імена. Дизасемблер скасовує процес складання, тому слід очікувати мову збірки як результат (і, отже, машинну мову як вхід). Декомпілятори прагнуть виробляти вихідні дані мовою високого рівня, якщо їм в якості введення введено збірку або навіть машинну мову.[19]

Сам процес декомпіляції являється дуже складним процес по декількох причинах:

- ***Процес складання є втратним***. На рівні машинної мови немає назв змінних чи функцій, а інформація про тип змінної може бути визначена лише за способом використання даних, а не за явними деклараціями типів. Коли ви спостерігаєте, як передаються 32 біти даних, вам доведеться виконати певну дослідницьку роботу, щоб визначити, чи представляють ці 32 біти ціле число, 32-бітове значення з плаваючою комою чи 32-бітний покажчик.
- ***Компіляція - це операція багато-до-багатьох.*** Це означає, що вихідну програму можна перекласти на асемблерну мову різними способами, а машинну мову можна перевести назад до вихідної різними способами. Як результат, компіляція файлу та його негайна декомпіляція зазвичай дає вихідний файл, який значно відрізняється від оригіналу.
- ***Декомпілятори залежать від мови та бібліотеки.*** Обробка двійкового файлу, створеного компілятором Delphi, за допомогою декомпілятора, призначеного для генерації коду C, може дати дуже дивні результати. Подібним чином, подача скомпільованого двійкового файлу Windows через декомпілятор, який не знає API програмування Windows, може не дати нічого корисного.
- ***Для точного декомпілюввання двійкового файлу потрібно майже ідеально дезасемблерувати двійковий код.*** Будь-які помилки або упущення на етапі дезасеблерування більш за все поширяться на декомпільований код. Розібраний код можна перевірити на правильність на

відповідні довідкові посібники процесора; однак канонічні довідкові посібники недоступні для перевірки правильності виводу декомпілятора.

Дизасемблерування починається з точки входу в програму, яка прописана в заголовку виконуваного файлу і йде послідовно зверху вниз. Існує два основних алгоритму дизассемблерування - лінійний алгоритм і алгоритм рекурсивного спуску.

Лінійний алгоритм дизассемблерування

Лінійний алгоритм не аналізує потік управління програми, припускаючи, що весь вміст області коду є кодом. Алгоритм здійснює лінійний аналіз починаючи з першого байта в кодовому розділі і лінійно переміщається по розділу, розбираючи одну інструкцію за іншою, поки не буде досягнуто кінець розділу. При розборі чергової інструкції, визначається її розмір, додається до її зміщення і виходить адреса наступної інструкції. При цьому не робиться жодних зусиль для розуміння потоку управління програмою через розпізнавання нелінійних інструкцій, таких як гілки. Перевагою лінійного алгоритму є висока швидкість дизассемблерування, а також те, що він забезпечує повне охоплення розділів коду програми, основним недоліком - помилки в дизассемблерування у випадках, коли код розташовується не безперервно внаслідок вирівнювання, або, коли в області коду містяться статичні дані програми.

Рекурсивний алгоритм дизассемблерування

Алгоритм рекурсивного спуску полягає в аналізі потоку управління, який починається з точки (точок) входу в програму. При аналізі інструкцій, які не здійснюють передачу управління, алгоритм поводиться так само, як лінійний. При аналізі інструкцій передачі управління обчислюється адреса переходу і додається в список адрес для подальшого аналізу. Після умовної передачі

управління або інструкції виклику функції аналіз триває лінійно, після безумовної передачі керування або інструкції повернення з функції послідовний аналіз переривається і здійснюється перехід на наступну адресу в списку раніше доданих, так як відразу за інструкцією може бути відсутнім код(вирівнювання, статичні дані). Таким чином, алгоритм рекурсивного спуску дозволяє автоматично визначати області коду, що відповідають окремим функцій.

IDA

Дизассемблерування в IDA здійснюється алгоритмом рекурсивного спуску, що призводить до необхідності обчислення адреси переходів та викликів. У разі використання часткової адресації обчислення адреси може виявитися неможливим, тому в IDA Pro вбудований інтерактивний засіб управління, що дозволяє отримувати значення динамічно обчислених адрес. Однак його вихід і процес експорту динамічної інформації не автоматизуються, що ускладнює використання IDA Pro для великих програм.

Висновки до розділу 2

В цьому розділі було розглянуто основні методи аналізу програм, такий як статичний та динамічний і вказано головні їх недоліки та переваги. А також описано метод комбінованого аналізу програми в тому випадку коли у нас немає доступу до вихідного коду. Досліджено як відбувається дизасемблерування програм і визначено в чому можуть виникнути проблеми. Було визначено програму для аналізу бінарних виконуваних файлів, а саме Ida, в якій є присутні всі необхідні засоби і функції для повноцінного аналізу виконуваних файлів.

3 СТВОРЕННЯ БІНАРНИХ ФАЙЛІВ

3.3 Створення бінарних файлів

Згідно з головною метою даної роботи потрібно провести аналіз бінарних виконуваних файлів. Перш зав все потрібно створити бінарні виконувані файли, над якими потім буде виконуватися аналіз. Для цього потрібно встановити необхідний застосунок, який можна отримати з офіційного ресурсу на GitHub GraalVM[17], а саме поліглот GraalVM.

Використовуємо GraalVM, встановлений компілятор, який компілює байт-код Java до машинного коду. Компілятор є модульним і може бути налаштований на різні віртуальні машини та сценарії компіляції, такі як динамічна компіляція та компіляція AOT. Однак підхід, запропонований у цій роботі, в основному не залежить від компілятора, і адаптація нашого компілятора для використання результатів точкового аналізу компіляції AOT замість профілювання інформація для динамічної компіляції не мала значного обсягу роботи.

Компілятор GraalVM виконує всі стандартні оптимізації, такі як вбудовування методів, постійне згортання та арифметичне, оптимізація циклу; та оптимізації, які особливо ефективні для об'єктно-орієнтованого коду Java, такого як частковий аналіз виходу.

Спеціальна оптимізація компілятора, яка вимагає деоптимізації, вимкнена для компіляції AOT. Натомість компілятор включає результати аналізу точок до покращення якості коду: поля, які не позначені як записані під час виконання, постійно складаються, незалежно від того, оголошені вони як остаточні чи ні.

Після чого додатково до GraalVM потрібно встановити інструмент для створення Native Image за допомогою команди ‘gu install native-image’. Після цього додаткового кроку функція Native Image стане доступною у каталозі GRAALVM_HOME / bin.

Було створено два виконуваних файли написаних на Java, скомпільованих в байт-код і після чого, за допомогою утиліти Native-Image було побудовано нативний додаток, який працює автономно, не потребуючи ні JVM ні інших додаткових засобів виконання Java програм. Один додаток, це проста програма, яка використовує різні функції для виводу випадкового числа. Другий застосунок це вже невеликий веб-додаток написаний на основі фреймворка Spring Boot, за допомогою збірника Apache Maven який підтримує побудову Native Image. Цей веб-додаток генерує в заданому діапазоні два випадкових числа, які являються верхніми і нижніми гранями, для виводу всіх простих чисел в цьому діапазоні. Його робота продемонстрована на рисунку 3.1 і 3.2.

Рисунок 3.1 – Робота сервера веб-додатка

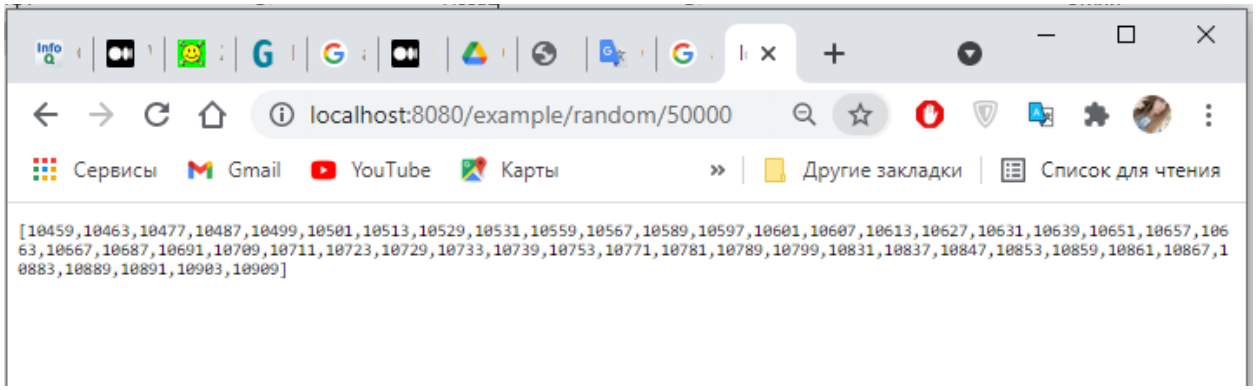

Рисунок 3.2 – Відповідь сервера на запит

3.4 Ініціалізація класів в Native Image

Семантика Java вимагає ініціалізації класу під час першого доступу до нього під час виконання. Ініціалізація класу має негативні наслідки для попередньої компіляції Java, оскільки:

- Це значно погіршує продуктивність власних зображень: кожен доступ до класу (через поле або метод) вимагає перевірки, якщо клас вже ініціалізований. Без спеціальних оптимізацій це може зменшити продуктивність більш ніж удвічі.
- Це збільшує обсяг роботи для запуску програми. Наприклад, просте “Hello, world!” Програма вимагає ініціалізації більше ніж 300 класів.

Щоб зменшити негативний вплив ініціалізації класу, Native Image підтримує ініціалізацію класу під час побудови: певні класи можуть бути ініціалізовані під час побудови зображень, робити ініціалізацію під час виконання та перевірку непотрібною. Вся інформація про статичний стан з ініціалізованих класів зберігається в зображенні. Доступ до статичних полів, які були ініціалізовані під час побудови, прозорий для програми і працює так, ніби клас було ініціалізовано під час виконання.

Вказівка політик ініціалізації класу може бути ускладненим через такі обмеження, які походять від семантики ініціалізації класу:

- Коли клас ініціалізований, усі суперкласи та суперінтерфейси із методами за замовчуванням також повинні бути ініціалізовані. Однак інтерфейси без методів за замовчуванням не ініціалізуються. Для опису цього використовується короткотерміновий "відповідний супертип", а також відповідний підтип для підтипів класів та інтерфейсів із методами за замовчуванням.
- Відповідні супертипи типів, ініціалізованих під час побудови, також повинні бути ініціалізовані під час побудови.

- Відповідні підтипи типів, ініціалізованих під час виконання, також повинні бути ініціалізовані під час виконання.
- На зображенні не повинно бути примірників класів, які ініціалізуються під час виконання.

Щоб насолодитися повним нестандартним досвідом роботи з Native Image і все одно отримати переваги ініталізації часу побудови, Native Image робить три речі:

- Ініціалізація часу створення власного зображення
- Автоматична ініціалізація безпечних класів
- Явно вказана ініціалізація класу

Для відстеження того, які класи були ініціалізовані та чому, можна використовувати прапор -H: + PrintClassInitialization. Цей прапор дуже допомагає налаштувати збірку зображень для роботи за призначенням. Мета полягає в тому, щоб якнайбільше класів було ініціалізовано під час побудови, але при цьому зберегти правильну семантику програми.[18]

Але програма самостійно не може визначити коректно чи потрібно ініціалізувати цей клас чи ні. І тому в цьому може критися небезпека, так як при побудові власного зображення може ініціальзуватися той статичний блок, який би не повинен бути ініціалізовнай. Так наприклад в застосунку з випадковими числами, було виявлено, що клас який містить статичний метод з генерацією випадкового числа, кожен раз видає те ж саме число, а не випадкове. Це продемонстровано на рисунку 3.3 та 3.4.

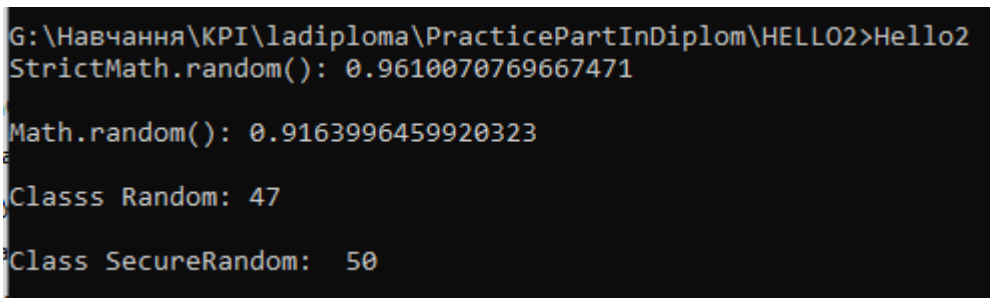

Рисунок 3.3 – Вивід випадково числа

```
G:\Навчання\KPI\ladiploma\PracticePartInDiplom\HELLO2>Hello2
StrictMath.random(): 0.9610070769667471

Math.random(): 0.9163996459920323

Classs Random: 4

Class SecureRandom:  38
```

Рисунок 3.4 – Вивід випадково числа вдруге

За допомогою функції, яка була вказана вище, були створені файли, в яких був записаний весь ланцюжок ініціалізації класів. Проаналізувавши ці файли, можна прослідкувати, що під час побудови нативного зображення ініціальзуються класи, які в цьому випадку не повинні були б ініціалізуватись. Рисунок 3.5 і Рисунок 3.6.

```
:_java_util_SplittableRandom -> BUILD_TIME reasons: substitutions are always initialized
```

Рисунок 3.5. – Відбувається ініціалізація класу і статичних блоків

```
' -> BUILD_TIME reasons: from feature com.oracle.svm.hosted.jdk.JDKInitial
'.SunJCE$SecureRandomHolder -> RERUN reasons: for substitutions
```

Рисунок 3.6. – Ініціалізація класу SecureRandom не відбувається

Статичне випадкове число може нести в собі велику небезпеку. Безпека криптографічних систем часто залежить від деяких даних, які повинні бути відомі лише авторизованим користувачам і які майже не можливо вгадати зловмисникові. Прикладами таких даних можуть бути - сеансові ключі, ініціалізуючі вектори, криптографічна сіль, унікальні параметри функції ЕЦП і тому подібне. Щоб досягти необхідного рівня непередбачуваності за умови частої генерації випадкових чисел, потрібно використовувати надійне джерело випадкових чисел. Тому дуже важливо, щоб випадкові числа були дійсно випадковими.

3.5 Пошук інформації в виконуваному файлі, яка потенційно може спричинити витік конфіденційних даних

Витік даних - це несанкціонована передача даних із організації чи приватної особи до зовнішнього пункту призначення або одержувача. Цей термін можна використовувати для опису даних, які передаються в електронному або фізичному плані.

Витік даних, також відомий як низьке і повільне викрадення даних, є величезною проблемою для безпеки даних, і шкода, заподіяна будь-якій організації, незалежно від розміру та галузі, може бути серйозною. Починаючи зі зменшення доходів до заплямованої репутації або масових фінансових покарань до скалічення судових процесів, це загроза, від якої будь-яка організація захоче захиститися.

Зрозуміло, що найвірнішим рішення для уникнення такої ситуації, це є обмеження доступу до цієї інформації користувачів.

Так як технологія створення Native Image за допомогою GraalVM є новою і, як вже було зазначено вище, під час побудови власного зображення самостійно визначає, що повинно бути вміщено в виконуваний файл, для того щоб він коректно працював, то можлива така ситуація, що воно помістить в цей виконуваний файл важливу користувацьку інформацію, що може спричинити витік конфіденційних важливих даних.

Для того щоб це визначити, веб застосунок було проаналізовано за допомогою IDA, де було дезасембльовано виконуваний файл та визначено велику кількість інформації, яку в собі вміщує цей виконуваний файл. Це продемонстровано на Рисунку 3.7, де видно, що аналізатор виявив аж 423 426 строчок інформації, яку вміщує в собі це виконуваний файл.

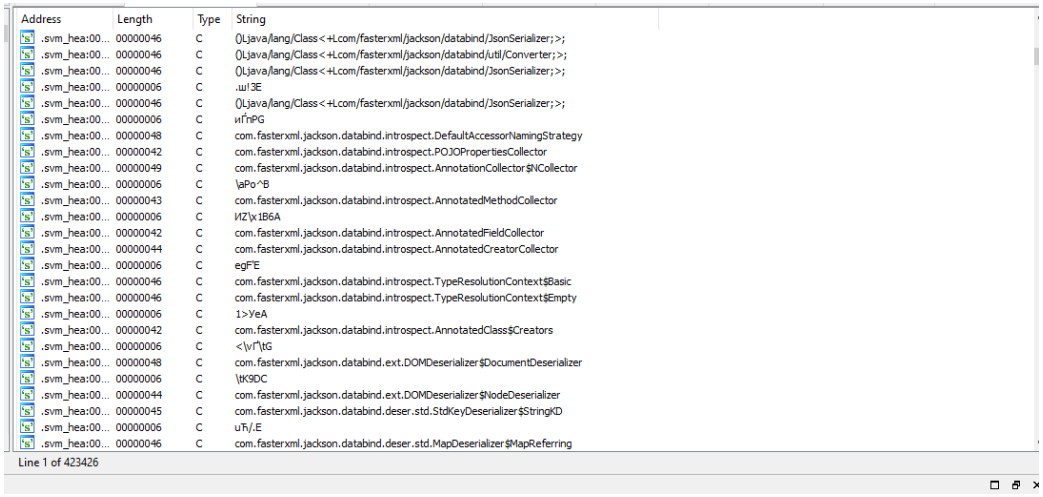

Рисунок 3.7 – Інформацію, яку містить в собі виконуваний файл

Проаналізувавши інформацію, яку визначила IDA було знайдено дані, які можуть спричинити витік конфіденційної інформації і становити загрозу. З цього зрозуміло, що ця технологія створення бінарних виконуваних файлів не є ідеальною і ще потребує допрацювання, вважаючи ці нюанси. Але зрозуміло, що в майбутньому це допоможе при роботи з різним програмним забезпеченням.

Висновки до розділу 3

В цьому розділі було створено два застосунки за допомогою GraalVM Native Image, які працюють повністю автономно. Один застосунок виводить випадкові числа різними методами, інший застосунок – це не великий веб-застосунок, який містить в собі свій власний сервер.

Було проведено аналіз створених виконуваних файлів, та виявлено баг в статичних блоках ініціалізації, в результаті якого метод, з отримання випадкового числа відпрацьовує під час побудови Native Image і випадкове число виводить те ж саме кожен раз при запуску програми.

Також в веб-застосунку було виявлено дані які можу спричинити витік конфіденційної інформації.

ВИСНОВКИ

У роботі було розглянуто традиційний спосіб створення бінарного виконуваного файлу, а також досліджено – GraalVM Native Image, створення бінарного виконуваного файлу на основі інтерпретованої мови програмування Java. Під час його створення застосунок вміщує всі переваги цієї мови програмування, я також через часткову ініціалізацію під час побудови застосунок має набагато швидший запуск, а також займає меншу кількість пам’яті під час виконання.

Для виконання аналізу бінарних виконуваних файлів було досліджено статичні та динамічні методи аналізу, та вибрано оптимальний підхід для аналізу бінарного виконуваного файлу.

Було створено і проаналізовано два бінарних виконуваних файлів. В результаті чого було виявлено баг при якому метод генерації випадкового числа виконується під час ініціалізації Native Image, це може становити загрозу під час генерації цифрових підписів і інших унікальних даних.

Також в веб-застосунку було виявлено дані які можу спричинити витік конфіденційної інформації.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Компіляция в C++ [Електронний ресурс] – Режим доступу до ресурсу: http://knzsoft.ru/cpp-bgr-ls1.
2. Сборка программ на C [Електронний ресурс] – Режим доступу до ресурсу:https://acm.bsu.by/wiki/C2017/%D0%A1%D0%B1%D0%BE%D1%80%D0%BA%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC_%D0%BD%D0%B0_C.
3. Examining the Compilation Process [Електронний ресурс] – Режим доступу до ресурсу: linuxjournal.com/content/examining-compilation-process-part-1.
4. Как устроен компилятор? [Електронний ресурс] – Режим доступу до ресурсу: https://www.kv.by/archive/index2008371109.htm.
5. The Compilation Process [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/coding-den/the-compilation-process-a1307824d40e.
6. А.А. Тюгашев ОСНОВИ ПРОГРАМУВАННЯ Частина I [Книга] – Режим доступу до ресурсу: https://books.ifmo.ru/file/pdf/1917.pdf.
7. Архітектура компиляторів [Електронний ресурс] – Режим доступу до ресурсу: https://habr.com/ru/company/mailru/blog/451894/.
8. What is Kotlin Native? [Електронний ресурс] – Режим доступу до ресурсу: https://learning.oreilly.com/library/view/kotlin-blueprints/9781788390804/216ea907-f532-4fd2-9554-3899a0542745.xhtml.

9. Low Level Virtual Machine [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Low_Level_Virtual_Machine.
10. What is LLVM? The power behind Swift, Rust, Clang, and more [Електронний ресурс] – Режим доступу до ресурсу: https://www.infoworld.com/article/3247799/what-is-llvm-the-power-behind-swift-rust-clang-and-more.html.
11. Maxim Shudrak, Vyacheslav Zolotarev; [Стаття] The technique of dynamic binary analysis and its application in the information security sphere.
12. Binary Code and Binary Analysis [Електронний ресурс] – Режим доступу до ресурсу: https://www.synopsys.com/glossary/what-is-binary-code-binary-analysis.html.
13. Why do you need dynamic code analysis if you have static analysis? [Електронний ресурс] – Режим доступу до ресурсу: https://pvs-studio.com/ru/blog/posts/0643/.
14. А.Ю.Тихонов, А.И. Аветисян, [Стаття] Комбінований (статичний і динамічний) аналіз бінарного коду.
15. A powerful disassembler and a versatile debugger [Електронний ресурс] – Режим доступу до ресурсу: https://hex-rays.com/ida-pro/.
16. GraalVM [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/graalvm/graalvm-ce-builds/releases/tag/vm-21.1.0.
17. Class Initialization in Native Image [Електронний ресурс] – Режим

доступу до ресурсу: https://www.graalvm.org/reference-manual/native-image/ClassInitialization/.

18. Chris Eagle, Kara Nance [Книга] The Gidra book. The Definitive Guide.

Додаток А

Застосунок виведення випадкових чисел

```
import java.security.SecureRandom;
import java.util.Random;


public class Hello2 {
    public static void main(String[] args) {
        System.out.println("StrictMath.random(): " + StrictMath.random());
        System.out.println();
        System.out.println("Math.random(): " + Math.random());
        System.out.println();
        Random random = new Random();
        System.out.println("Classs Random: " + random.nextInt(100));
        System.out.println();
        SecureRandom rand = new SecureRandom();
        System.out.println("Class SecureRandom:  " + rand.nextInt(100));
    }
     }
```

Додаток Б

Веб-застосунок

1. package com.example.nativeImage.nativeImage;

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;
import java.util.Random;
import java.util.stream.Collectors;
import java.util.stream.LongStream;

@RestController
@RequestMapping("/example")
public class NativeController {
    private Random r = new Random();
    @GetMapping("/random/{upperboand}")
    @ResponseBody
    public List<Long> random(int upperboand){
        int to = 2 + r.nextInt(upperboand-2);
```

```
        int from = 1 + r.nextInt(to-1);
        return primeSequence(from, to);
    }
    public static boolean isPrime(long n){
        return LongStream.rangeClosed(2, (long)Math.sqrt(n)).allMatch(i-
     >n%i!=0);
    }
    private List<Long> primeSequence(long min, long max) {
        return LongStream.range(min, max)
                .filter(NativeController::isPrime)
                .boxed()
                .collect(Collectors.toList());
    }
}
```

2.

```
package com.example.nativeImage.nativeImage;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class NativeImageApplication {

    public static void main(String[] args) {
```

```
SpringApplication.run(NativeImageApplication.class, args);      }}
```