МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Західноукраїнський національний університет
Факультет комп'ютерних інформаційних технологій
Кафедра кібербезпеки

Хомич Олександр Васильович

Алгоритм аудиту безпеки та реагування на інциденти комп'ютерних систем на базі ОС Linux / Algorithm of security audit and incident response of computer systems based on OC Linux

спеціальність: 125 – Кібербезпека

освітньо-професійна програма – Кібербезпека

Кваліфікаційна робота

Виконав студент групи КБм -21
О.В. Хомич

Науковий керівник
к.т.н., доцент С.В.Івасьєв

Кваліфікаційну роботу допущено
до захисту:

«____» _____ ___2022 р.

Завідувач кафедри
_____ В.В.Яцків

ТЕРНОПІЛЬ - 2022

Факультет комп'ютерних інформаційних технологій

Кафедра кібербезпеки

Освітній ступінь «магістр»

спеціальність: 125 - Кібербезпека

освітньо-професійна програма –Кібербезпека

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ В.В.Яцків

____" _____ 2021 року

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

ХОМИЧ Олександр Васильович

(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи:
Алгоритм аудиту безпеки та реагування на інциденти комп'ютерних систем на базі ОС Linux / Algorithm of security audit and incident response of computer systems based on OC Linux
керівник роботи к.т.н., доцент С.В.Івасьєв

затверджені наказом по університету від 31 грудня 2021 року № 606

2. Строк подання студентом закінченої випускної кваліфікаційної роботи 16 листопада 2022 року.

3. Вихідні дані до кваліфікаційної роботи: завдання на випускну кваліфікаційну роботу студента, наукові статті, технічна література.

4. Основні питання, які потрібно розробити:
– дослідити існуючі системи захисту інформації;
– провести аналіз вибору мови програмування для реалізації системи;
– дослідити симетричні, асиметричні і гібридні алгоритми шифрування;
– провести аналіз швидкодії синхронних і асинхронних задач при читанні з жорсткого диску;
– розробити алгоритм і систему реагування на інциденти кібербезпеки;
– провести тестування розробленого програмного засобу.

5. Перелік графічного матеріалу у роботі.
Схема роботи алгоритму реагування на інциденти.
Візуалізація модульної структури системи.

Синхронне і асинхронне виконання задач.
Характеристики тестового стенду.
Звіти швидкодії при тестуванні.

6. Консультанти розділів кваліфікаційної роботи

	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 11 жовтня 2021 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строки виконання етапів кваліфікаційної роботи	Примітка
1	Аналіз підходів до автентифікації в системах контролю доступу	12.2021 р. – 03.2022 р.	
2	Методи доступу до системи	03.2022 р. – 05.2022 р.	
3	Розробка системи контролю доступу на основі двофакторної автентифікації	05.2022 р. – 11.2022 р.	

Студент _____ Хомич О.В.
(підпис)

Керівник роботи _____ к.т.н., доцент С.В. Івасьєв
(підпис)

АНОТАЦІЯ

Магістерська робота на тему “ Алгоритм аудиту безпеки та реагування на інциденти комп'ютерних систем на базі ОС Linux ” зі спеціальності 125 – кібербезпека написана обсягом 77 сторінок і містить 9 ілюстрацій, 2 додатки та 51 джерел за переліком посилань.

Метою роботи є розробка алгоритму реагування на інциденти для інформаційної системи в операційній системі Linux, ціль якої повне шифрування чутливих даних на пристрої у разі інциденту, з використанням гібридного алгоритму шифрування а також видалення слідів існування системи реагування.

Алгоритм шифрування базується на використанні гібридного алгоритму шифрування. Для шифрування даних використовується AES, а для шифрування симетричного ключа використовується асиметричний алгоритм RSA з стандартною довжиною ключа 2048 байт.

Проаналізовано алгоритми симетричного і асиметричного шифрування, а також необхідні функції для реалізації системи реагування на інциденти. Розроблено алгоритм аудиту безпеки та реагування на інциденти для інформаційної систем на базі ОС Linux, з відкритим вихідним код а також модульною системою. Проведено тестування рішення, а також досліджено часові характеристики розробленого програмного продукту. Запропонований підхід визначається високою швидкодією про що свідчать результати тестування.

Результати роботи можуть бути використані в системах захисту інформації, як частина комплексного захисту даних або повноцінна система реагування на інциденти. Можливими напрямками подальших досліджень є дослідження можливостей інтеграції розробленої системи з іншими системами захисту, а також реалізацію більш ефективного алгоритму шифрування.

КЛЮЧОВІ СЛОВА: АЛГОРИТМ, КРИПТОГРАФІЯ, КІБЕРБЕЗПЕКА, СИСТЕМА РЕАГУВАННЯ НА ІНЦИДЕНТИ.

ABSTRACT

The master's thesis on the topic "Algorithm of security audit and incident response of computer systems based on OC Linux" from the specialty 125 - cyber security is written in the volume of 100 pages and contains 9 illustrations, 2 appendices and 51 sources according to the list of references.

The purpose of the work is to develop an incident response algorithm for an information system in the Linux operating system, the purpose of which is to fully encrypt sensitive data on the device in the event of an incident, using a hybrid encryption algorithm, as well as removing traces of the existence of the response system.

The encryption algorithm is based on the use of a hybrid encryption algorithm. AES is used for data encryption, and RSA asymmetric algorithm with a standard key length of 2048 bytes is used for symmetric key encryption.

The symmetric and asymmetric encryption algorithms, as well as the necessary functions for the implementation of the incident response system, are analyzed. A security audit and incident response algorithm for information systems based on OC Linux, with open source code and a modular system, has been developed. The solution was tested, and the time characteristics of the developed software product were also investigated. The proposed approach is determined by high speed, as evidenced by the test results.

The results of the work can be used in information protection systems, as part of comprehensive data protection or a full-fledged incident response system. Possible areas of further research are the study of the possibilities of integration of the developed system with other protection systems, as well as the implementation of a more effective encryption algorithm.

KEY WORDS: ALGORITHM, CRYPTOGRAPHY, CYBER SECURITY, INCIDENT RESPONSE SYSTEM.

ЗМІСТ

ЗМІСТ 5

ВСТУП 7

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ 11
1.1 Вибір мови програмування 11
1.2 Алгоритми виявлення загрози і реагування на інциденти 14
1.3 Вибір алгоритму шифрування для системи реагування на інциденти 16
1.4 Аналіз існуючих рішень 17
1.5 Постановка завдання 19

2 ПРОЕКТУВАННЯ АРХІТЕКТУРИ СИСТЕМИ 21
2.1 Декомпозиція архітектури 21
2.2 Асинхронність та багатопроцесність 27
2.3 Опис бібліотек використаних в проекті 32

3 ПРОГРАМНА РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ ТА БЕНЧМАРКИ 37
3.1 Реалізація ядра шифрування 37
3.2 Файли запуску системи і окремих модулів 39
3.3 Реалізація системи реагування 50
3.4 Файл налаштувань 59
3.5 Імплементація пакетів шифрування і дешифрування даних 61
3.6 Імплементація автоматизованих тестів. 67
3.6 Тестування швидкодії 68

ВИСНОВКИ 72

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 73

ДОДАТОК А. 78

ДОДАТОК Б. 79

ВСТУП

Актуальність роботи. З розвитком технологій і зміщення фокусу нашого життя у віртуальний світ, збільшується і кількість кібератак, які зараз стали звичним явищем у всьому світі, становлячи серйозну загрозу для окремих людей і організацій. Зросла не тільки кількість атак, а і їхня складність. Зросли ризики які пов’язані з цими атаками. Велика частка атак є пасивними, прихованими, які намагаються бути не поміченими. Водночас як компаніям, так і звичайним користувачам часто доводиться стикатися з економічними і психологічними наслідками кібератака, несанкціонованого доступу до інформації, а також її втрати.

У реаліях сучасного цифрового світу крадіжка персональних даних перетворилася на серйозну проблему, з якою зіштовхнулося чимало користувачів по всьому світу. Протягом декількох останніх років, через значне зростання кількості покупок в онлайн-магазинах, збільшення користування соціальними мережами та інтернет-шахрайств, кількість крадіжок персональних даних зросла в сотні разів.

Згідно з результатами досліджень, проведених компанією Verizon, 52% крадіжок даних були здійснені за допомогою злому хакерами системи захисту, 25% виконані за допомогою шкідливих програм, а 32% - за допомогою фішингу або соціальної інженерії.

Щодня величезна кількість людей стають жертвами інтернет-шахраїв. І хоч про більшість їхніх схем фахівці уже розповідали безліч разів, кіберзлочинцям вдається знаходити все нових і нових жертв.

Зараз є багато інтернет-ресурсів та статей, які діляться основними правилами, порадами яких варто дотримуватися, щоб знизити ризики крадіжки персональних даних. Головні серед таких правил це:

- використовувати складні паролі;
- використовувати лише ліцензійні програми;

- здійснювати регулярне резервне копіювання даних, зберігати резервні копії на зовнішніх носіях інформації (SDD, HDD тощо);
- уникати використання інтернет-банкінгу, електронних платіжних систем, введення аутентифікаційних даних під час доступу до інтернету через загальнодоступні (незахищені) безпроводові мережі (в кафе, барах аеропортах та інших публічних місцях);
- не переходити за сумнівними посиланнями;
- для завантаження необхідних файлів використовувати перевірені офіційні ресурси;
- не підключати флешки та зовнішні диски, CD та DVD тощо у ваш комп'ютер, якщо ви не довіряєте повністю їх джерелу;
- переконатися у правильності назви необхідного сайту;
- перевіряти невідомі номери на сайті кіберполіції;
- зберігати електронні фотокопії документів (паспортів нотаріальних свідоцтв та доручень тощо) лише на шифрованих або зовнішніх дисках.

Всі перелічені вище поради знижують шанси стати жертвою кіберзлочинців. Також є багато порад які менш універсальні і будуть корисні для користувачів які займаються специфічною діяльністю. Але дотримання всіх цих правил не гарантує 100% захисту, адже сучасні комп'ютерні системи дуже складні і завжди є ймовірність існування прогалини у безпеці системи. Також є ймовірність фізичного несанкціонованого доступу до вашого персонального комп'ютера чи ноутбука (крадіжка, втрата, тощо). Тому важливо подбати про конфіденційність ваших даних і в такій ситуації:

- використовувати пароль для входу в обліковий запис операційної системи;
- встановити автоматичне блокування пристрою при тривалій бездіяльності;
- встановити пароль на жорсткий диск (у операційній системі Windows ми можемо скористатися вбудованою програмою BitLocker);
- створювати резервні копії ваших даних;

- не залишати ноутбук без нагляду.

Найслабша ланка в кібербезпеці – це людина. Зокрема, успішність кібератак на 91% залежить від людських помилок (відкривання небезпечних посилань, успішний фішинг, завантаження шкідливих файлів, тощо) і лише на 9% від технічних проблем (слабка захищеність систем, відсутність спеціального софту, тощо). Тож не можна нехтувати ризиком отримання несанкціонованого доступу до файлів через звичайну неуважність, тощо. Тому дуже доречно захистити свої конфіденційні дані у разі виникнення ситуації несанкціонованого доступу, і вжити запобіжні заходи навіть у ситуації коли злочинець фізично або віртуально отримав доступ до вашого пристрою.

Мета роботи полягає в розробці програмного застосунку з відкритим вихідним кодом, для реагування на інциденти, який при настанні події (трігера) в автоматичному режимі, без втручання користувача, шифрує усі особисті дані на пристрої, за допомогою гібридного алгоритму шифрування який базується на симетричному шифруванні AES і шифруванні ключів AES за допомогою асиметричного алгоритму RSA. Також до якісних особливостей відноситься модульна незалежна структура яка дозволяє в будь-який момент використовувати власний модуль шифрування, реагування, тощо.

Для досягнення даної мети ставились наступні завдання:

- дослідити існуючі мови програмування та вибрати найбільш релевантну для цього завдання;
- дослідити наявні системи реагування на інциденти та виконати постановку завдання;
- провести порівняння системи з іншими;
- дослідити можливість розширення подій в системі реагування на інциденти.
- розробити модульну систему реагування з відкритим кодом;
- провести тестування розробленого програмного засобу на операційній системі Windows і Linux
- провести тестування швидкодії алгоритму шифрування

- створити автоматизовані тести для полегшення підтримки програмного продукту сторонніми розробниками.

Об’єкт дослідження – процес реагування на інциденти коли злочинець отримав фізичний або віртуальний доступ до пристрою та алгоритми шифрування даних.

Предмет досліджень – алгоритм та модульна система аудиту безпеки та реагування на інциденти комп'ютерних систем на базі ОС Linux.

Наукова новизна одержаних результатів визначається наступним чином:

- розроблено алгоритм аудиту безпеки та реагування на інциденти комп'ютерних систем на базі ОС Linux.

Практична цінність одержаних результатів полягає в тому, що:

- реалізовано систему аудиту безпеки та реагування на інциденти комп’ютерних систем на базі ОС Linux з використанням гібридного алгоритму шифрування на мові Python.

Публікації та апробація до кваліфікаційної роботи. За результатами наукових досліджень, проведених у кваліфікаційній роботі, підготовлено тези доповіді на проблемно-науковій міжгалузевій конференції «Кібербезпека та комп’ютерно-інтегровані технології» (КБКІТ – 2022).

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Вибір мови програмування

Платформою для реалізації програмного рішення вирішено вибрати Python.

Python - це мова програмування загального призначення, націлена в першу чергу на підвищення продуктивності програміста. Говорячи простою мовою, на Python можна написати практично що завгодно (веб- / настільні додатки, ігри, скрипти по автоматизації, комплексні системи розрахунків, системи управління життєзабезпеченням і багато іншого) без відчутних проблем [3-4].

Більш того, ця мова програмування має нижчу складність для розуміння, лаконічний і зрозумілий код навіть для тих, хто ніколи не працював з ним. За рахунок простого, не перевантаженого синтаксису, подальший супровід програм, написаних на Python, стає простішим в порівнянні з Java або C++. До головних переваг Python відносяться:

- сучасний мовний синтаксис, використання різноманітних парадигм програмування (ООП, функціональність, реактивність);
- велике, активне ком’юніті, багато відповідей на запитання на StackOverflow;
- Python має велику бібліотеку стандартних пакетів (не треба їх додатково інсталювати, щоб створювати навіть досить складні проєкти) та ще більшим списком пакетів Open Source, доступних в публічних репозиторіях (для легкого встановлення). У тому числі потужні пакети для обробки зображення, створення графіків, статистичного аналізу даних, мережевої комунікації, штучного інтелекту (особливо Deep Learning) та Machine Learning, опрацювання натуральної мови, взаємодії з базами даних, створення веб додатків, витягування даних з веб сайтів, графічних алгоритмів, тощо;

- кросплатформеність — програми в Python працюють на кожній платформі, для якої доступний інтерпретатор Python. Вистачає раз написати й можна запускати на майже кожній операційній системі, в тому числі і мобільних;
- Python є чудовою мовою програмування для прототипування, тобто швидкого написання пробних версій додатка, які мають перевірити підхід або довести, що якась концепція спрацює. А це завдяки стислому й багатому синтаксису, великі кількості готових модулів і мінімальним вимогам для початку створення додатка;
- величезна кількість матеріалів у мережі, чудова документація для великої кількості стандартних пакетів;
- керована пам’ять — інтерпретатор Python сам звільняє пам’ять, виділену програмі, але яка вже не використовується. Це полегшує продумування програми, скорочує код і унеможливлює допускання помилок, пов’язаних із передчасним звільненням ще потрібної пам’яті, які часто з’являються в програмах, написаних, наприклад, мовами C або C++.

Але жодна мова програмування не ідеальна, і має свої недоліки. Кожна перевага, особливість має свою ціну:

- система типізації — це одночасно перевага й недолік. Для програмістів-початківців певним шоком може бути те, що в програмі не треба подавати типи для змінних. Python розпізнає їх сам й відповідно перевіряє в ході діяльності програми, чи ми не пробуємо виконати на даних не дозволені операції (так звана сильна система типізації). Однак це відбувається лише на етапі виконання конкретного шматка коду, а тому про потенційну помилку дізнаємося лише після запуску. Це зумовлює те, що, створюючи програми на Python, більшу увагу слід приділити їхньому тестуванню. На відміну від компільованих мов програмування (C, C++, Java, C# та багато інших), компілятор не допоможе нам викрити певні помилки, тому ми повинні самі подбати про відповідне покриття коду тестуваннями. Відсутність явної типізації також є певного роду ускладненням, коли ми розширюємо велику систему. Тоді

явна типізація є важливою допомогою для програміста, який читає код, написаний кимось іншим. Тому також нові версії мови Python мають опційну можливість опису функцій і класів через типи, що є рекомендованою практикою у випадку більших додатків;

- продуктивність. Більшість розробників, та й сам творець мови, сходяться на думці, що Python не такий спритний, наскільки хотілося б. Це обумовлено тим, що Python інтерпретована мова. Але навіть у порівнянні з іншими інтерпретованими мовами помітно, що Python програє в продуктивності. Але це легко можна нівелювати за допомогою C реалізацій того чи іншого проблемного ділянки коду. В умовах сьогоднішніх потужностей — це не сильно помітно.
- Global Interpreter Lock. На даний момент це є основною проблемою продуктивності в Python, а також цим обумовлена погана реалізація багатопоточності. Код GIL не змінювався з першої версії мови. Це явно вказує на те, що він застарів.

Python чудово підходить для реалізації даного проекту через багато причин, серед яких, наявність предустановленного інтерпретатора в комплекті з усіма операційними системами на базі UNIX, свою величезну бібліотеку модулів, стандартних та сторонніх, зручність читання і написання коду, а також керованість пам’яттю. Звісно є і недоліки у виборі цієї мови програмування. Перш за все це швидкість, але її можна компенсувати використанням останніх версій інтерпретатора, профілюванням системи, та написанням найжадібніших до обчислень частин програми використовуючи розширення C, а також використовуючи багатопроцесність та асинхронність при роботі з IO Bound операціям. У системі реагування до таких операцій потенційно будуть належати хендлери які реагують на події, і які можна буде винести в окремий поток або процес, а також читання та запис у файл. На щастя починаючи з версію 3.5.x в Python за допомогою сторонніх бібліотек підтримує асинхронний доступ до файлів, тож процес шифрування і розшифрування не повинен бути вузьким

місцем в нашій системі. Тобто ми правильному написанні системи ми можемо уникнути недоліків і отримати усі переваги обраної мови програмування.

1.2 Алгоритми виявлення загрози і реагування на інциденти

Будь який алгоритм виявлення загроз базується на індикаторі кіберзагрози - показнику який включає технічні дані, виявлення кіберзагрози. Індикаторів кіберзагрози, які і способів несанкціонованого доступу до системи може бути багато, і всі вони повинні працювати одночасно, не перешкоджаючи і не блокуючи робочу іншого індикатора.

Оскільки ця система передбачає останню, крайню межу у захисті вашої інформації, у ситуаціях коли зловмисник несанкціоновано отримав доступ до вашого персонального девайсу, то важливо правильно ідентифікувати злочинця і його наявність в системі. Також, оскільки ми враховуємо можливість отримання фізичного доступу до девайсу шахраєм, і одночасну втрату доступу власника, ми повинні розділити наші індикатори на дві групи:

- події системи (автоматизовані);
- події ініційовані користувачем.

Події системи (автоматизовані) - це події які реєструють підозрілі дії всередині системи і в автоматичному режимі реагують на інцидент (без втручання користувача)

До подій ініційованих користувачем відносяться дії які не може задетектити машина (наприклад втрата персонального комп’ютера, тощо).

Для системи такого класу важливий високий відсоток точності при пошуку загрози, адже хибні спрацювання можуть заважати роботі, тож при проектуванні можливих подій для точної ідентифікації я обрав наступні події:

- honeypot;
- usb access key;

- hotkey (подія ініційована користувачем);
- telegram bot command (подія ініційована користувачем).

Honeypot - передбачає створення директорії/файла на цільовому пристрої, який буде звертати на себе увагу (за допомогою назви, даних, тощо), і доступ до якого буде постійно моніторитись у фоновому режимі. Оскільки ми знаємо, що це пастка, то і взаємодіяти з директорією/файлом не будемо, а отже шанс випадкового спрацювання майже відсутній. Моніторинг файлової системи в ОС Linux дозволяє отримувати інформацію про будь-яку взаємодію з файлом, будь це читання, запис, копіювання, видалення, тощо, тож будь-яка взаємодія викличе спрацювання системи безпеки.

USB access key - передбачає створення “прихованого” файлу ключа. У ролі ключа може виступати будь-який файл, з будь-якою назвою, який може знаходитись на будь-якому рівні вкладеності. Ідея полягає у використанні системи моніторингу доступу до файлової системи. При зверненні до цьової директорії з конфіденційною інформацією буде запускатися пошук доступних USB пристроїв на девайсі, і при їх виявленні буде розпочинатися пошук ключа. Якщо ключ відсутній, то спрацьовує тригер безпеки і всі файли шифруються.

Hotkey - передбачає використання гарячої клавіші для спрацювання системи безпеки і шифрування даних. Дуже просто але корисно у деяких, специфічних ситуаціях.

Telegram bot command - передбачає ініціалізацію, реалізацію і запуска екземпляру телеграм бота на локальній машині (1 машина = 1 телеграм бот), для прослуховування вхідних повідомлень і реалізацію секретної команди, при виклику якої бот шифрує всі файли на пристрої користувача. Дуже зручний спосіб запуску тривоги віддалено, наприклад при втраті девайсу.

Усі індикатори безпеки реалізуються як окремі незалежні модулі, тож при потребі покриття ще більшого спектру сценаріїв, або ж використання специфічних тригерів є можливість розробити і підключити сторонній модуль.

1.3 Вибір алгоритму шифрування для системи реагування на інциденти

Шифрування – перетворення даних у певний формат так, що тільки авторизовані користувачі можуть отримати доступ до інформації. Процес шифрування стає можливим завдяки криптографічним ключам в поєднанні з різними математичними алгоритмами. Розділяють два основних типи шифрування – симетричне і асиметричне, а також іноді в окрему категорію виділяють гібридне.

При симетричному шифруванні дані шифруються за допомогою ключа та алгоритму шифрування. У той час як зашифрована інформація перетворюється у відкриті дані, за допомогою того ж самого ключа, який був використаний для шифрування. Алгоритм симетричного шифрування виконується швидше і є менш складним, тому частіше використовуються для шифрування великих масивів даних. Загальновживаними алгоритмами симетричного шифрування є DES, 3 DES, AES, RC4.

Асиметричне шифрування - це тип шифрування, який використовує пару ключів (приватний ключ і публічний ключ) для шифрування та дешифрування. Публічний ключ є вільним для всіх, хто зацікавлений у використанні. Приватний ключ зберігається в таємниці. Виконання алгоритму асиметричного шифрування відбувається повільно, оскільки асиметричні алгоритми шифрування мають високу обчислювальну складність і їх важко застосовувати для шифрування потокових даних. Асиметричне шифрування зазвичай використовується для встановлення захищеного каналу через незахищений носій інформації, як Інтернет. Найпоширенішим алгоритмом асиметричного шифрування є RSA.

Гібридне шифрування - це режим шифрування, який об'єднує дві або більше систем шифрування. Він включає комбінацію симетричного та асиметричного шифрування, щоб отримати користь від сильних сторін кожної

форми шифрування. Для нас це відповідно швидкість шифрування та безпека при перехопленні відкритого ключа зловмисником.

Гібридне шифрування вважається високобезпечним типом шифрування до тих пір, поки загальнодоступні та приватні ключі повністю захищені.

1.4 Аналіз існуючих рішень

Шифрувальне програмне забезпечення — це програмне забезпечення, основним завданням якого є шифрування і дешифрування даних, як правило, у вигляді файлів (або секторів), жорстких дисків і змінних носіїв (дискет, компакт-дисків, USB-флеш-накопичувачів), повідомлень електронної пошти або у вигляді пакетів, що передаються через комп'ютерні мережі. Шифрування може бути застосовано до даних різними способами. Загальні категорії:

- Disk encryption software;
- File/folder encryption;
- Database encryption;
- Communication encryption software.

Величезну увагу розробники і ентузіасти відкритого програмного забезпечення і свободи слова сьогодні приділяють програмним засобам захисту від несанкціонованого доступу до інформаційних ресурсів і особливо до мережі Інтернет. Організаційні, технологічні й апаратні методи захисту, як правило, не можуть бути здійснені без програмної складової. При цьому слід мати на увазі, що вартість здійснення багатьох програмних системних рішень із захисту інформації суттєво перевищує за затратами апаратні, технологічні й організаційні рішення. Іншими недоліками програмних засобів захисту інформації є використання ресурсів системи, що призводить до зниження її ефективності, принципова можливість обходу такого захисту або його злочинної зміни в процесі експлуатації.

Найпоширенішими прикладами програмних засобів захисту інформації є такі:

- система контролю і управління доступом;
- антивірусні програми;
- шифрувальне програмне забезпечення;
- мережевий екран;
- система виявлення вторгнень;
- керування записами;
- пісочниця;
- система управління інформаційною безпекою;
- SIEM.

Є багато різних програмних рішень, різного класу для захисту інформації від зловмисників, безліч програм які шифрують дані, але під час пошуку програм для захисту даних я знайшов лише частково реалізований функціонал, який не повністю закриває потреби в безпеці. Є рішення з відкритим кодом, як наприклад Crypter, який написаний на JavaScript і надає крос-платформну криптографічну систему, яка надає зручний функціонал шифрування та дешифрування, водночас зберігаючи надійний захист, але цей програмний продукт і багато схожих йому не є незалежним і зручним, оскільки у повсякденному житті ми не будемо тримати дані у зашифрованому форматі і при потребі взаємодіяти з ними розшифровувати їх, читати і шифрувати знову, оскільки це зайвий клопіт, який неприйнятний для більшості користувачів. Тож Crypter і безліч програм того ж типу можуть бути чудовими рішення якщо інтегрувати їх у систему реагування на базі подій, де від користувача не потрібно жодних дій, або ж необтяжливий мінімум. Також у мережі є багато антивірусних рішень, наприклад ESET NOD32 Antivirus, Bitdefender GravityZone Business Security, Kaspersky Endpoint Security, тощо, але ці рішення зазвичай з закритим вихідним кодом, великі, і потребують аналізу і налаштування. Також більшість систем захисту розраховані на захист пристрою від кібератак, але не надають жодного захисту у випадку отримання фізичного

доступу до девайсу. Навіть пароль операційної системи не є захистом при потраплянні до рук зловмисника.

У цій роботі головною ціллю було розробити легку систему, з відкритим кодом і модульною структурою яка не буде потребувати налаштувань, догляду і додаткових дій з боку користувача. Ідея не у створенні власного аналога антивіруса, а у розробці останнього бар'єру між зловмисником і вашими даними. Продукт розрахований на використанні з іншими система захисту для збільшення захисту і зменшення % ймовірних інцидентів, а також продукт повинен бути ефективним у випадках фізичної взаємодії зловмисника з девайсом, або ж повної втрати доступу до пристрою. Також цей продукт може захистити дані у випадку якщо користувач не дотримувався усіх правил кібербезпеки і до прикладу не встановив обов'язкове шифрування жорсткого диску, або ж проігнорував встановлення складного пароль в операційній системі, тощо. При пошуку готових програмних рішень такого класу у пошуковій системі а також на хмарному сервісі зберігання програмного забезпечення GitHub знайдено не було.

1.5 Постановка завдання

Головною ціллю є розробка системи аудиту безпеки та реагування на інциденти комп'ютерних систем на базі ОС Linux. Передбачається використання цієї системи разом з іншими системами захисту (системи сканування, антивіруси, тощо) і рекомендаціями щодо кібер гігієни (шифрування жорсткого диску, автоматичне блокування ОС, пароль до ОС, тощо). Оскільки сучасні комп'ютерні системи дуже складні з безліччю підсистем, модулів, рівнів абстракції, тощо, то і організувати надійний захист даних складно. Більше того жодна система не гарантує і не може гарантувати 100% захисту, оскільки за даними компанії PwC, через співробітників фірма або компанія втрачає набагато

більше даних, ніж через хакерські атаки. Головною загрозою системи компанії є так звана соціальна інженерія, також відома як мистецтво “злому” окремої людини. Тож передбачається, що спроектована система система буде останнім бар’єром між зловмисником і вашими персональними або ж корпоративними даними.

Більшість систем кіберзахисту не передбачають втрати доступу до девайсу його власником і отримання несанкціонованого фізичного доступу зловмисником, але такий сценарій доволі поширений, і при побудові системи реагування на інциденти важливо передбачити і такий розвиток подій і завчасно прийняти міри захисту чутливої інформації.

Ця система є некомерційним рішенням, з відкритим кодом, модульною структурою, тож кожен охочий може використати його як основу і підлаштувати під себе додавши необхідний його організації функціонал. При реалізації використовуються технології, модулі, бібліотеки і рішення які використовують великими компанії в своїй проектах, тож є надійними, і відмовостійкими.

2 ПРОЕКТУВАННЯ АРХІТЕКТУРИ СИСТЕМИ

2.1 Декомпозиція архітектури

Декомпозиція - це поділ однієї великої мети на завдання для успішного досягнення цієї самої мети; простими словами декомпозицію застосовують для продуктивного розподілу часу та ресурсів перед великим завданням.

При проектуванні системи важливо зробити якісну декомпозицію задачі, щоб спроектувати гнучку, розширювану архітектуру. Перший етап - розділення системи на модулі. Під час декомпозиції програму було розділено на три незалежні модулі:

- система безпеки і реагування;
- генератор ключів;
- пакет дешифрування;

Усі ці три модулі можуть знаходитись поряд, оскільки злочинець навіть маючи доступ до програми генерації ключів і пакету дешифрування все одно не може взламати шифрування оскільки він базується на рандомній генерації ключі, (який не можливо відтворити, на відміну від псевдорандому), тому у репозиторії вони всі будуть знаходитись поряд, але для збільшення надійності і стійкості програми все ж рекомендується на цільовій машині зберігати лише систему безпеки і реагування разом з відкритим ключем шифрування.

Наступний крок - декомпозиція найскладнішої частини програмного продукту - системи безпеки і реагування. Було виділено наступні підмодулі:

- ядро (core);
- інтерфейс налаштувань;
- ядро шифрування;
- пакет алгоритмів шифрування;
- абстрактний інтерфейс системи реагування
- набір систем реагування (тригерів).

Схематичне зображення алгоритму роботи системи зображено на рисунку 2.1

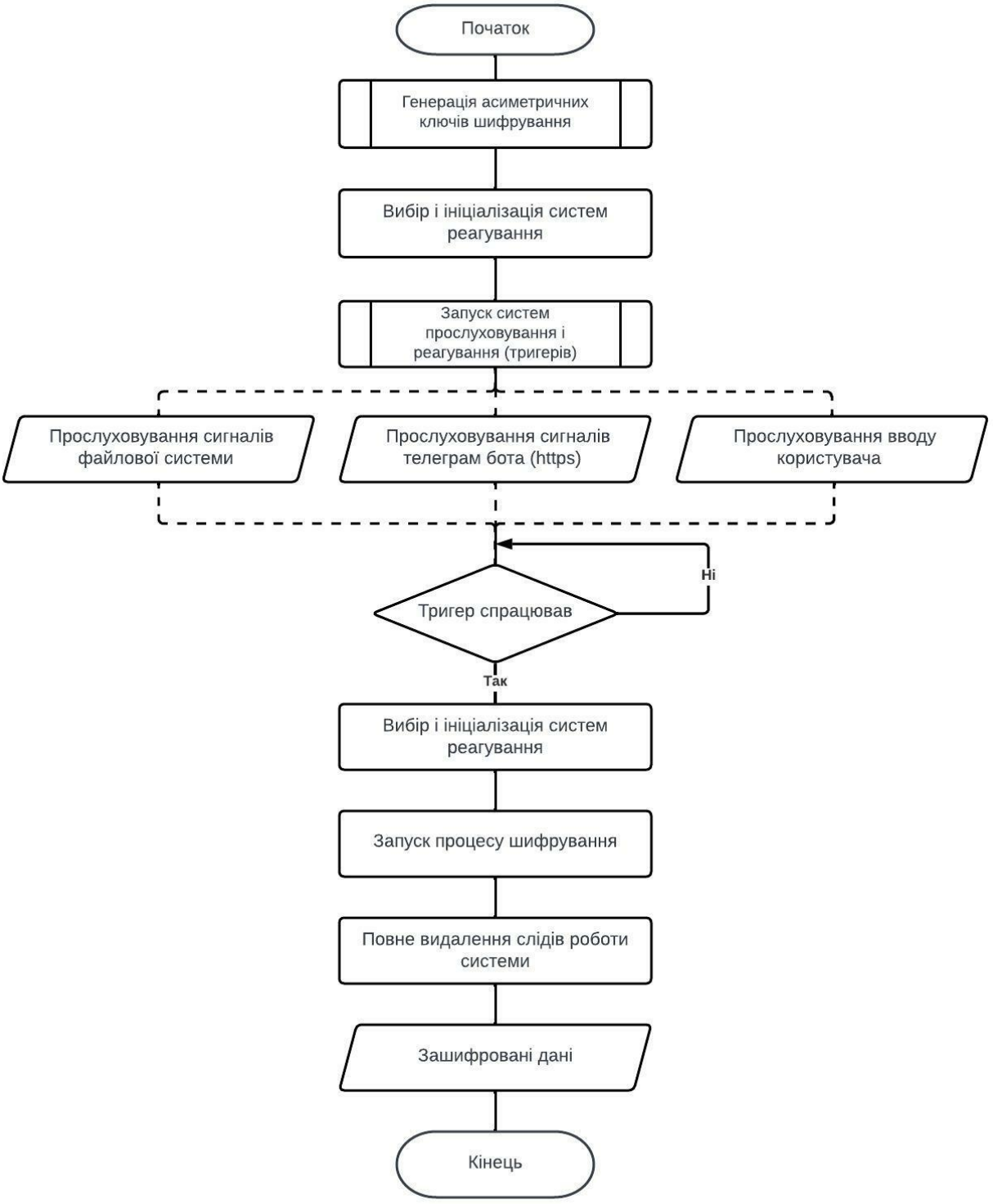


Рисунок 2.1 Схематичний алгоритм роботи системи реагування

Ядро (core) - це модуль який реалізує всередині себе контракт взаємодії усіх інших модулів системи. Для реалізації універсальної структури потрібно використати поведінковий паттерн під назвою “Шаблонний метод”.

Шаблонний метод — це поведінковий патерн проектування, який визначає кістяк алгоритму, перекладаючи відповідальність за деякі його кроки на підкласи. Патерн дозволяє дочірнім класам перевизначати кроки алгоритму, не змінюючи його загальної структури.

Патерн “Шаблонний метод” пропонує розділити алгоритм на послідовність кроків, описати ці кроки в окремих методах і викликати їх в одному шаблонному методі один за одним. Це дозволить підкласам перевизначити деякі кроки алгоритму, залишаючи без змін його структуру та інші кроки, які для цього підкласу не є важливими. Також головний процес (батьківський) буде запускати усі системи реагування в окремих процесах, для уникнення блокування та проблем з продуктивністю через наявність Global Interpreter Lock. Батьківський процес зберігає список усіх дочірніх процесів і з певної періодичністю перевіряє чи процес живий. Якщо він не отримує відповідь то вважає, що сталася подія вторгнення і запускає шифрування даних. Також важливо передбачити програмні збої які можуть виникати на будь-якій етапі роботи програми, і коректно їх обробляти. Оскільки на вхід будуть прийматися різні файли для шифрування, то ми не можемо завчасно знати яка інформація буде в них зберігатися, і тим самим не можемо бути впевнені, що файл існує, або має коректне кодування, тощо. Для перехоплення помилок і виключень в Python використовується конструкція try – except. Ця мова програмування побудована на основі об’єктної моделі, і все, що ми використовуємо це або посилання на об’єкт, або об’єкт. Винятки (exceptions) - це об’єкт, стандартний тип даних, який необхідний для того щоб сповіщати програміста про помилки. Всі винятки наслідуються від базового класу BaseException. Є багато типів винятків, наприклад ZeroDivisionError – ділення на нуль, або KeyError – неіснуючий ключ, і інші. За допомогою конструкції try – except ми можемо відловлювати всі винятки, або лише винятки певного класу, задавати команди які повинні виконуватися в разі помилки, виводити інформацію про помилку в один з потоків запису, тощо. Базовий синтаксис

наведено на рисунку нижче. В коді спеціально допущено помилку ділення на нуль, для демонстрації конструкції.

```
6   try:
7       a = 1 / 0
8   except:
9       print('Сталася помилка!')
10      a = None
11
```

Рисунок 2.1 - Базовий синтаксис конструкції try – except

Також в конструкції try – except передбачено використання ключових слів else і finally. Блок коду загорнутий в else виконається лише в тому випадку, якщо в блоці try не відбулася помилка. Finally – виконається у будь-якому випадку. На рисунку 2.2 проілюстровано конструкцію «except ZeroDivisionError as e:» яка дозволяє відловлювати не всі винятки, а лише певного класу, і також зберегти інформацію про цей виняток в змінну e (назва може бути довільною) за допомогою менеджеру контексту «as» для подальшої роботи з винятком, наприклад для запису в файл логів.

```
6   try:
7       a = 1 / 0
8   except ZeroDivisionError as e:
9       print(f'Error = {e}')
10      a = None
11  else:
12      print('Цей блок коду виконається, '
13            'якщо в процесі виконання конструкції не буде помилки')
14  finally:
15      print('Цей блок виконається в будь-якому випадку')
16
17
```

Рисунок 2.2 - Розширений синтаксис try – except

Отже при обробці файлів користувача потрібно загорнути критично важливий код, в якому може відбутися помилка в блок відловлення винятків, передбачити запис інформації про помилку в файл логів, і написати код який буде продовжувати коректну роботу системи, забезпечуючи високий рівень відмовостійкості.

Інтерфейс налаштувань - це модуль які інкапсулює в собі усю логіку імпорту, ініціалізації і використання налаштувань. Це може бути скрипт який сприймає вхідні з аргументів командного рядка, файл який зберігає в собі всі змінні з можливістю їх редагування, або ж графічний інтерфейс. Оскільки програма передбачає велику кількість налаштувань для забезпечення гнучкості, а з ростом індикаторів безпеки кількість цих змінних може збільшуватись, то було прийнято рішення реалізувати окремий файл конфігурації, для якого, при потребі можна реалізувати графічний інтерфейс, для зручності користувача. Для зручності імплементації, не порушуючи принцип “12 факторів”, було використано бібліотеку environs, яка дозволяє зберігати, чутливі дані, такі як токени за межами, репозиторію, наприклад у .env файлах.

Ядро шифрування - це модуль який імплементує контракт взаємодії з алгоритмами шифрування, функції для отримання, аналізу, взаємодії з усіма цільовими файлами на диску, а також інтерфейс який імплементує алгоритм раціонального розподілення обчислювальних ресурсів в залежності від доступних ресурсів системи, а також розподілення задач по пулах, визначення типу завдань (синхронні чи асинхронні функції) і запускає для всіх пулів окремі процеси, цикли подій (якщо потрібно) і відповідає за обробку виключень і поверненню результатів виконання кожної функції

Пакет алгоритмів шифрування - модуль який імплементує в собі функції для шифрування. Тут також важливо дотримуватися єдиного інтерфейсу. Функції повинні приймати на вхід два параметра:

- ключ шифрування;
- абсолютний або відносний шлях до файлу.

У разі успіху функція не повинна нічого повертати. А у разі помилки піднімати виключення. Аналогічний інтерфейс взаємодії повинен реалізовувати і метод дешифрування даних.

Абстрактний інтерфейс системи реагування - це інтерфейс взаємодії який реалізовано в файлі ініціалізації пакету хендлерів як абстрактний клас з інтерфейсом конструктора класу (__init__() метод) і інтерфейсом виклику класу (__call__() метод). Реалізація підмодулів буде базуватися на наслідуванні від абстрактного класу контракту “AbstractHandler” який зобов’язує наслідників описані вище методи. Перевагою використання абстрактного класу є однаковий інтерфейс який зобов’язуються перевизначити усі наслідники, і в разі невиконання контракту Python підніме виключення при ініціалізації дочірнього класу, а не при виклику відсутнього методу класу.

Набір систем реагування (тригерів) - це окремі, незалежні модулі які реалізують інтерфейс ініціалізації і виклику, працюють у фоновому режимі, в окремих, дочірніх потоках, і у разі настання події зупиняють виконання свого процесу. Цей модуль буде складатися з 4 підмодулів описаних вище:

- honeypot;
- usb access key;
- hotkey (подія ініційована користувачем);
- telegram bot command (подія ініційована користувачем).

Реалізація систем реагування (тригерів) буде базуватися на наслідуванні від абстрактного класу контракту “AbstractHandler” який зобов’язує наслідників перевизначити конструктор класу, метод __call__() для запуску хендлера і атрибут process_name який зберігає в собі назву процесу, для коректної ідентифікації процесу. Схематичне зображення модульної структури системи наведено на рисунку 2.2.

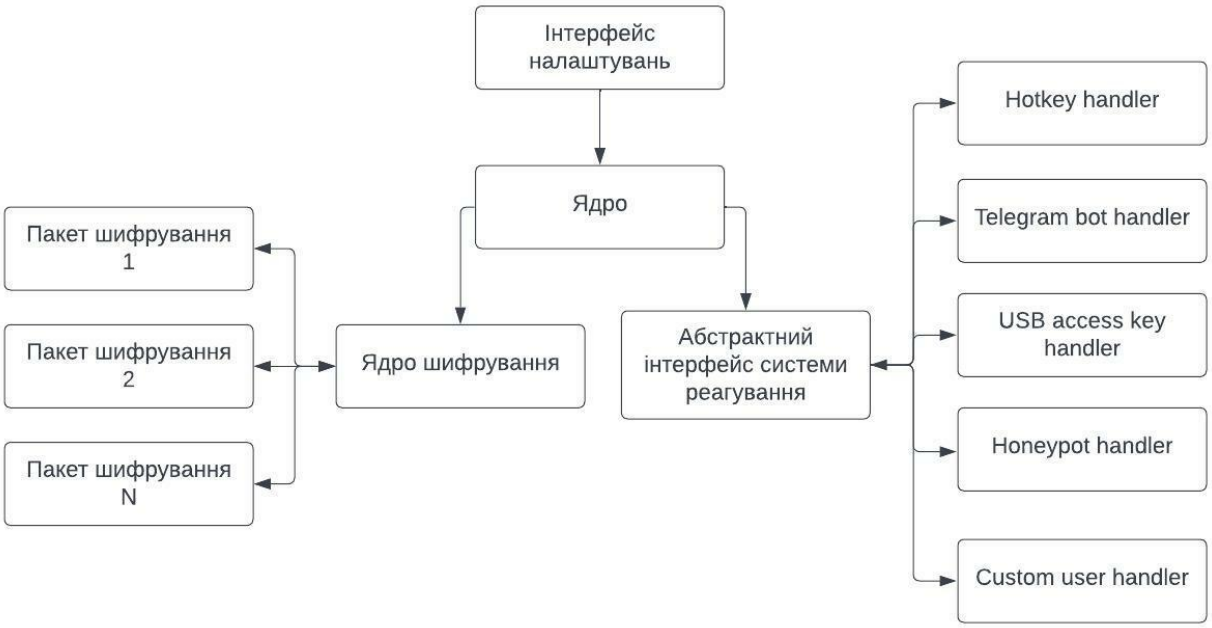


Рисунок 2.2 Візуалізація модульної структури системи

На рисунку зображено модульну структуру, всі модулі взаємодіють за допомогою одиного протоколу, і при потребі користувач може додавати, видаляти або редагувати будь-який модуль системи тим самим змінюючи його функціонал під власні потреби.

2.2 Асинхронність та багатопроцесність

Для уникнення головного недоліку Python, а саме його поганої продуктивності при роботі з математичними операціями і CPU bound задачами важливо використовувати асинхронність багатопроцесність та багатопоточність.

Перш за все багатопоточність або багатопроцесність важливо використовувати у блокуючих операціях. У нашій системі кожен хендлер є блокуючим. Наприклад телеграм-бот запускає свій екземпляр і у вічному циклі перевіряє надходження нових оновлень(повідомлень) від серверів телеграму. По аналогії працюють і інші систему аудиту безпеки які реагують події файлової системи, потік вводу, тощо. Тому для уникнення блокування ми запускаємо

кожен підмодуль в окремому процесі. Для цієї задачі могли б підійти потоки, але у реалізації мови програмування Python, побудованій на C, яка постачається з більшістю операційних систем Linux і є основною, існує GIL.

Python Global Interpreter Lock (GIL) – це своєрідне блокування, яке дозволяє лише одному потоку керувати інтерпретатором Python. Це означає, що у будь-який час буде виконуватися лише один конкретний потік. Робота GIL може здаватися не суттєвою для розробників, які створюють однопотокові програми. Але в багатопотокових програмах відсутність GIL може негативно позначатися на продуктивності процесорно-залежних програм. Оскільки GIL дозволяє працювати тільки одному потоку навіть у багатопотоковому додатку, він отримав репутацію «сумно відомої» функції. У нашій програмі критичним є час шифрування файлів, оскільки чим більше часу ми будемо витрачати на шифрування даних, тим більша ймовірність, що зловмисник бодай частково, але отримає доступ до конфіденційної інформації. Для збільшення швидкості шифрування чудовим рішенням є використання модуля multiprocessing, який надає програмний інтерфейс для створення процесів. Пакет пропонує як локальний, так і віддалений паралелізм, ефективно обходячи глобальне блокування інтерпретатора, використовуючи підпроцеси замість потоків. Завдяки цьому він дозволяє програмісту повністю використовувати потужність усіх ядер процесора на даній машині і працює як на Unix, так і на Windows. У нашій програмі він дозволить розпаралелити обчислення які навантажують центральний процесор, такі як процес перетворення відкритих даних у зашифровані і навпаки.

Наступним кроком проектування для збільшення продуктивності буде використання асинхронності для I/O bound операцій.

Різниця між синхронним і асинхронним виконанням коду полягає в черговості виконанні завдань. В синхронному програмуванні вони виконуються по черзі, одне після іншого. При асинхронному ж, операції виконуються незалежно одна від іншої. Створення процесів коштує дорого. Тому для I/O більше підходять потоки. Також ми знаємо що швидкість I/O-операцій залежать

від багатьох факторів — повільний диск або нестабільна мережа роблять I/O-операції довшими. Asyncio надає цикл подій та ще деякі можливості. Цикл подій реагує на різні I/O-події та перемикається на завдання, що можуть виконуватися і призупиняє ті, що чекають на I/O. Тобто ми не витрачаємо час на завдання, що ще не готові виконуватися.

Ідея дуже проста. У нас є цикл подій (event loop). А ще в нас є асинхронні функції, I/O-операції. Ми передаємо свої функції до циклу подій, щоб він запустив їх. Цикл подій повертає нам об'єкт Future. Можна сказати, що це обіцянка, що ми отримаємо якісь дані в майбутньому. Ми зберігаємо його і час від часу перевіряємо чи не має наш Future результату виконання. І якщо так, то використовуємо ці дані для подальшої обробки. Візуальну різницю між виконанням I/O bound задач при синхронному і асинхронному програмуванні зображену на рисунку 2.3.

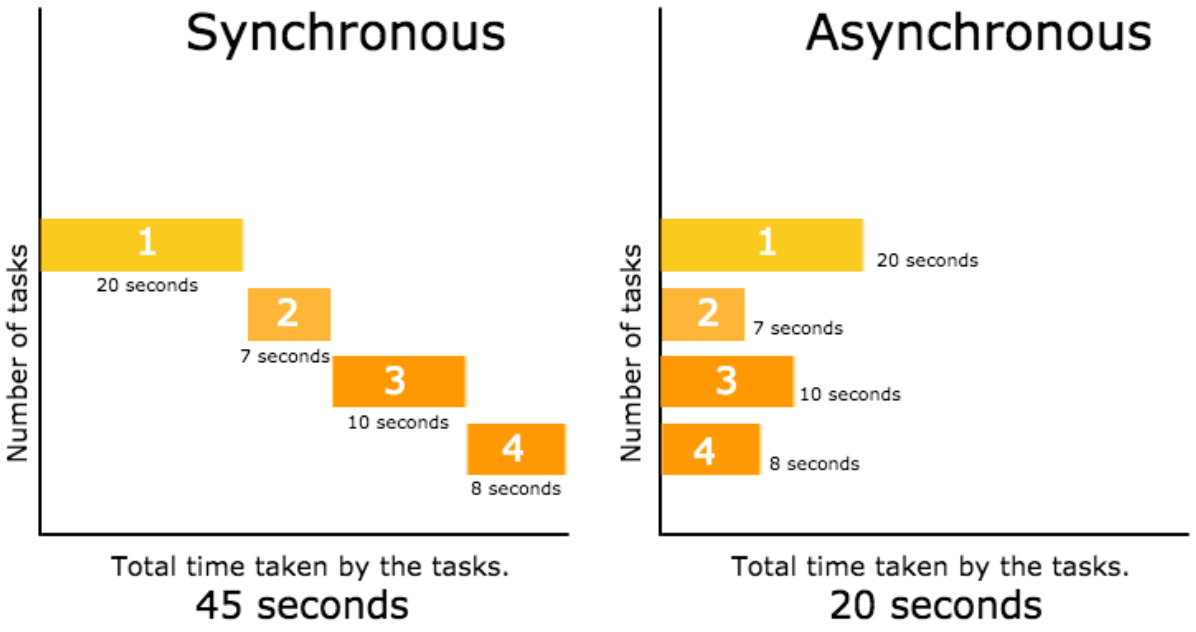


Рисунок 2.3 - Синхронне і асинхронне виконання задач.

Починаючи з версії 3.5.x Python реалізує нативну підтримку асинхронності за допомогою async/await. Вони спрощують асинхронне програмування на Python. Ключове async використовується для створення

корутини(асинхронної задачі) Python. Ключове слово await призупиняє виконання корутини, доки вона не завершиться та не поверне результат роботи. Корутина — це форма узагальнення підпрограми, яка повертає об'єкт Future, який є обіцянкою того, що програма зобов’язується повернути результат виконання, а також є індикатором для інтерпретатора, що до цього об’єкту потрібно повернутися згодом і запитати чи не завершився він. Зазвичай корутини використовуються для спільних завдань і поводяться як генератори Python. Для опитування корутин Python використовує “Цикли подій”. Цей механізм запускає корутини і перемикається між ними при використання ключового слова await. Це можна уявити як цикл while(True) , який відстежує корутини, отримує відгуки про те, які задачі в активній роботі, а які в режимі очікування відповіді, і шукає задачі, які можна виконати тим часом. У Python одночасно може виконуватися лише один цикл подій.

У нашій системі прикладо I/O bound операцій є взаємодія з диском (читання і запис файлів). Python не вміє працювати з файлами напряму, він віддає це завдання на виконання операційній системі в якій запущений, і очікує вивід який вона поверне. Тобто він працює з інтерфейсом під назвою “File-Like Objects”. Оскільки доступ до файлів відбувається за межами процесу Python, то ця операція є блокуючою, і її можна оптимізувати за допомогою асинхронності. Стандартний модуль бібліотеки Python asyncio не реалізовує асинхронну взаємодію з операціями вводу/виводу. Але для цих цілей ми можемо використати сторонню бібліотеку aiofiles. Справжній асинхронний файловий ввід/вивід, так званий не блокуючий, надається сучасними операційними системами, такими як Windows, MacOS і Linux, хоча надається не узгоджено. Це може бути причиною того, чому стандартна бібліотека Python наразі не підтримує асинхронний файл вводу-виводу, або, можливо, фокус asyncio зосереджений на мережевому програмуванні. Але Python славиться своїм великим ком'юніті розробників і величезною бібліотекою готових сторонніх модулів, до яких відноситься і бібліотека aiofiles.

Оскільки звичайний, нативний інтерфейс файлового вводу-виводу блокуючий, і його неможливо легко та портативно зробити асинхронним. Це означає, що виконання файлу IO може заважати асинхронним програмам, які не повинні блокувати потік виконання. aiofiles допомагає в цьому, запроваджуючи асинхронні версії файлів, які підтримують делегування операцій окремому пулу потоків. Вона не реалізує справжній асинхронний файл вводу-виводу, натомість вона імітує асинхронний файл вводу-виводу за допомогою робочих потоків під обкладинками. Файли відкриваються за допомогою aiofiles.open() корутини, яка крім віддзеркалення вбудованої функції open() приймає додаткові параметри цикла подій та виконавця(executor).

Приклад створення і запису даних у файл в асинхронному режимі за допомогою бібліотеки aiofiles зображено на рисунку 2.4.

```
# SuperFastPython.com
# example of writing to a file with asyncio and aiofiles
import asyncio
import aiofiles

# write a file
async def main():
    # open the file
    async with aiofiles.open('test_write.txt', mode='w') as handle:
        # write to the file
        await handle.write('Hello world')

# entry point
asyncio.run(main())
```

2.4 Відкриття та запис у файл у aiofiles

Якщо цикл подій відсутній, використовуватиметься цикл за замовчуванням відповідно до встановленої асинхронної політики. Якщо виконавець не вказаний, буде використано виконавець циклу подій за замовчуванням. У разі успіху асинхронний файловий об’єкт повертається з API, ідентичним звичайному файлу, за винятком деяких вузькоспеціалізований методів, які є співпрограмами та делегують виконавцю.

2.3 Опис бібліотек використаних в проекті

Опис бібліотек використаних в проекті потрібно розпочати з стандартних бібліотек Python, які постачаються в комплекті з пакетним менеджером Python (Pip) і інтерпретатором CPython.

sys - це модуль забезпечує доступ до деяких змінних та функцій, що взаємодіють з інтерпретатором pythona також забезпечує високорівневу взаємодію з операційною системою і її API.

inspect - цей модуль надає кілька корисних функцій, які допомагають отримати інформацію про існуючі об’єкти, такі як модулі, класи, методи, функції, трасування, об’єкти фрейму та об’єкти коду. Наприклад, це може допомогти вам вивчити вміст класу, отримати вихідний код методу, витягнути та відформатувати список аргументів для функції або отримати всю інформацію, необхідну для відображення детального відстеження. Цей модуль надає чотири основні види послуг: перевірка типу, отримання вихідного коду, перевірка класів і функцій і перевірка стека інтерпретатора.

multiprocessing - це пакет, який підтримує процеси породження за допомогою API, схожого на threading модуль. Пакет multiprocessing пропонує як локальний, так і віддалений паралелізм, ефективно обходячи глобальне блокування інтерпретатора, використовуючи підпроцеси замість потоків. Завдяки цьому multiprocessing модуль дозволяє програмісту повністю використовувати кілька процесорів або ядер на даній машині. Він працює як на Unix, так і на Windows. Модуль multiprocessing також представляє API, які не мають аналогів у threading модулі. Яскравим прикладом цього є Pool об’єкт, який пропонує зручний засіб розпаралелювання виконання функції для кількох вхідних значень, розподіляючи вхідні дані між процесами (паралелізм даних).

os - модуль який забезпечує портативний спосіб використання залежних від операційної системи модулів. Функція open() для доступу та запису у файл, os.path для маніпулювання шляхами. Для створення тимчасових файлів і

каталогів tempfile модуль, а для високорівневої обробки файлів і каталогів підмодуль shutil.

typing - це модуль, який забезпечує підтримку підказок типу під час виконання. Найбільш фундаментальна підтримка складається з типів Any, Union, Callable, TypeVari Generic. Важливо уточнити, що середовище виконання Python не вимагає анотацій типів функцій і змінних, але вони можуть використовуватися сторонніми інструментами, такими як засоби перевірки типів, IDE, лінтери тощо. Також підказки типів дозволяють краще розуміти і читати код програмісту.

Наступний блок модулів є розробками сторонніх розробників які постачаються окремо, і встановлюються за допомогою команди “pip install <package_name>”.

pycryptodome - це пакет низькорівневих криптографічних примітивів на Python, без додаткових залежностей. Для швидшої роботи з відкритими ключами в Unix вам слід встановити GMP у системі. PyCryptodome є форком PyCrypto. Він містить такі вдосконалення щодо останньої офіційної версії PyCrypto (2.6.1):

- Автентифіковані режими шифрування (GCM, CCM, EAX, SIV, OCB);
- Прискорений AES на платформах Intel через AES-NI;
- Першокласна підтримка PyPy;
- Криптографія еліптичних кривих (NIST P-криві; Ed25519, Ed448);
- Кращий і компактніший API (атрибути nonce і iv для шифрів, автоматична генерація випадкових nonces і IV, спрощений режим шифрування CTR тощо);
- Хеш-алгоритми SHA-3 (FIPS 202) і похідні функції (NIST SP-800 185);
- XOF SHAKE128 і SHA256;
- Аутентифіковані шифри ChaCha20-Poly1305 і XChaCha20-Poly1305;
- функції деривації scrypt, bcrypt і HKDF;

- Детермінований (EC)DSA та EdDSA;
- Контейнери ключів PKCS#8, захищені паролем;
- Схема обміну секретами Шаміра;
- Випадкові числа отримують безпосередньо з ОС (а не з CSPRNG у просторі користувача);
- Спрощений процес встановлення, включаючи кращу підтримку Windows;
- Чистіша генерація ключів RSA та DSA (головним чином на основі FIPS 186-4);
- Основні очищення та спрощення бази коду.

PyCryptodome не є оболонкою для окремої бібліотеки C, як OpenSSL. Максимально можливою мірою алгоритми реалізовані на чистому Python. Лише частини, які надзвичайно критичні для продуктивності (наприклад, блокові шифри), реалізуються як розширення C. Він підтримує Python 2.7, Python 3.5 і новіші версії, а також PyPy.

environs - це бібліотека Python для аналізу змінних середовища. Це дозволяє зберігати конфігурацію окремо від вашого коду відповідно до методології The Twelve-Factor App.

psutil (процеси та системні утиліти) — це міжплатформна бібліотека для отримання інформації про запущені процеси та використання системи (ЦП, пам’ять, диски, мережа, сенсори) у Python. Це корисно в основному для моніторингу системи, профілювання та обмеження ресурсів процесу та керування запущеними процесами. Він реалізує багато функцій, які пропонуються класичними інструментами командного рядка UNIX, такими як ps, top, iotop, lsof, netstat, ifconfig, free та інші. Наразі psutil підтримує такі платформи:

- Linux;
- Windows;
- macOS;
- FreeBSD, OpenBSD, NetBSD;

- Sun Solaris;
- AIX.

Підтримуються версії Python 2.7, 3.4+ і PyPy.

pynput - це стороння бібліотека, яка дозволяє контролювати пристрої введення. Наразі підтримуються введення та моніторинг за допомогою миші та клавіатури. У нашій системі ця бібліотека використовується для імплементації хендлера для моніторингу гарячих клавіш.

pyinotify - це модуль Python для моніторингу змін файлової системи. Pyinotify покладається на функцію ядра Linux (об'єднану в ядро 2.6.13), яка називається inotify. inotify — це сповіщувач, керований подіями, його сповіщення експортуються з простору ядра в простір користувача за допомогою трьох системних викликів. pyinotify прив'язує ці системні виклики та забезпечує реалізацію поверх них, пропонуючи загальний та абстрактний спосіб маніпулювання цими функціями.

aiogram - це досить проста та повністю асинхронна структура для API Telegram Bot, написана на Python 3.7 з asyncio та aiohttp. Вона допомагає створювати телеграм ботів швидкими та простими. Також ця бібліотека славиться величезною популярністю, швидкими оновленнями (слідом за API телеграму), зручною, зрозумілою архітектурою, і величезним ком'юніті, а це є величезною перевагою при розробці і вирішенням можливих проблем під час розробки. Розробка базується на веб-сервері aiohttp і є дуже гнучкою і продуктивною.

aiofiles - це ліцензована бібліотека Apache2, написана мовою Python, для обробки файлів локального диска в програмах asyncio. Звичайний локальний файл вводу-виводу блокує, і його неможливо легко та портативно зробити асинхронним. Це означає, що виконання файлу IO може заважати асинхронним програмам, які не повинні блокувати потік виконання. aiofiles допомагає в цьому, запроваджуючи асинхронні версії файлів, які підтримують делегування операцій окремому пулу потоків. Також підтримується асинхронна ітерація. Особливості:

- файловий API, дуже схожий на стандартний у Python, блокуючий API;
- підтримка буферизованих і небуферизованих двійкових файлів, а також буферизованих текстових файлів;
- підтримка конструкцій async/ await;
- асинхронний інтерфейс до модуля tempfile.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ ТА БЕНЧМАРКИ

3.1 Реалізація ядра шифрування

Будь-яка серйозна розробка обов’язково повинна використовувати систему контролю версій, для збереження прогресу розробки, історії змін, а також щоб мати можливість відмінити певні зміни. Під час розробки весь код і історія зберігалася з використанням системи контролю версій Git.

Git — це розподілена система керування версіями для відстеження змін у вихідному коді під час розробки програмного забезпечення. Він призначений для координації роботи програмістів, але його можна використовувати для відстеження змін у будь-якому наборі файлів. Його цілі включають швидкість, цілісність даних і підтримку розподілених нелінійних робочих процесів.

Сам ж вміст репозиторію зберігався на GitHub. GitHub — це веб-сайт і хмарний сервіс, який допомагає розробникам зберігати свій код і керувати ним, а також відстежувати та контролювати зміни в коді. Інтерфейс GitHub досить зручний, тому навіть початківці програмісти можуть скористатися Git. Без GitHub використання Git зазвичай вимагає трохи більше технічної підкованості та використання командного рядка. Проте GitHub настільки зручний, що деякі люди навіть використовують GitHub для керування іншими типами проектів, наприклад для написання книг. GitHub вважається стандартом для розміщення проектів з відкритим вихідним кодом (open source projects), там зберігаються і продовжують розроблятися тисячі програм щодня. Розробка була розпочата згідно стандартів індустрії створення програмного забезпечення на Python. Перша за все, як було описано вище був ініціалізований пустий git проект, а також створений GitHub репозиторій. Наступним кроком було додано файл .gitignore в якому перелічуються всі файли, які не повинні потрапити в систему контролю версій. Розробка проекту розпочалась на версії Python 3.8.5.

Версія 3.8 є частиною стандартного набору програм для більшості операційних систем на базі Linux, в тому числі на операційних системах якими

користуюсь я: Linux Mint і Ubuntu. При розробці проекту активно використовувалась утиліта pyenv, яка доступна в операційних системах Linux, Windows, а також macOS, для зручного менеджменту версій Python. За допомогою цієї утиліти можна перемикатися на будь-яку версію інтерпретатора за допомогою всього однієї команди: pyenv local <MAJOR.MINOR.PATCH>. Також ця утиліта автоматично створює файл .python-version у якому зберігає версію інтерпретатора яка використовується у цьому проекті. Локальна розробка відбувалася у віртуальному середовищі, яке можна створити за допомогою утиліти venv однією командою: python -m venv venv, і активувати за допомогою команди source venv/bin/activate. Віртуальні оточення дозволяють ізолювати різноманітні проекти один від одного і уникнути проблем з залежностями.

Для зручності використання програми під час розробки, відкладки і використання було реалізовано файл скорочених команд на базу Makefile:

Від дозволяє виконувати завчасно описані команди за допомогою одного короткого виклику. Для початку потрібно створити команду яка буде активовувати віртуальне середовище, а у разі якщо його немає то створювати його. Це реалізується за допомогою наступної інструкції в файлі Makefile:

```
venv/bin/activate: ## alias for virtual environment
python -m venv venv
```

Далі оголошуються команди, кожна з яких використовує описану вище команду для активації віртуального оточення і реалізують корисний для користувача або розробника функціонал. Список усіх доступних команд:

- setup (встановлення усіх залежностей);
- keys (генерація ключів для RSA);
- en (запуск процесу шифрування);
- de (запуск процесу дешифрування);
- run (запуск системи);

- tests (запуск усіх тестів);
- help (виведення доступних команд Makefile).

Код наведено нижче:

```
setup: venv/bin/activate ## project setup
	. venv/bin/activate; pip install wheel setuptools
	. venv/bin/activate; pip install --exists-action w -Ur requirements.txt
keys: venv/bin/activate ## Create public and private keys
	. venv/bin/activate; python generate_keys.py
en: venv/bin/activate ## Create public and private keys
	. venv/bin/activate; python run.py encrypt
de: venv/bin/activate ## Create public and private keys
	. venv/bin/activate; python run.py decrypt
run: venv/bin/activate ## Create public and private keys
	. venv/bin/activate; python entry.py
tests: venv/bin/activate ## run tests
	. venv/bin/activate; pytest
help: ## display this help screen
	@grep -h -E '^[a-zA-Z_-]+:.*?## .*$$' $(MAKEFILE_LIST) | awk 'BEGIN {FS = ":.*?## "}; {printf "\033[36m%-30s\033[0m %s\n", $$1, $$2}'
```

Усі ці команди роблять взаємодію з програмою зручнішою, і дозволяють не запам’ятовувати специфічні параметри запуску скриптів, а також дозволяють стандартизувати команди для запуску однакових завдань при використанні різних технологій.

3.2 Файли запуску системи і окремих модулів

Наступним кроком було встановлення бібліотеки pycryptodome==3.15.0 для використання реалізації RSA, і створення першої точки входу в нашу програму, для цього було створено файл generate_keys.py. Для реалізації використовувались наступні імпорти:

```
import os
from pathlib import Path
from Crypto.PublicKey import RSA
from src.config import RSA_KEY_SIZE, keys_dir_path, public_key_file_name, private_key_file_name
```

У файлі оголошується функція generate_keys(), яка приймає чотири аргументи: шлях до папки в якій потрібно зберегти файли, а також назви відкритого та приватного ключів і розмір ключа (стандартно це 2048 біт). Код оголошення функції наведено нижче:

```
def generate_keys(
        dir_path: str = '_keys',
        public_key_file_name: str = 'public.pem',
        private_key_file_name: str = 'private.pem',
        key_size: int = RSA_KEY_SIZE,
):
    dir_path = Path(dir_path)
    os.makedirs(dir_path, exist_ok=True)
    rsa = RSA.generate(key_size)
    private_key: bytes = rsa.export_key()
    public_key: bytes = rsa.public_key().export_key()
    with open(dir_path / private_key_file_name, "wb") as private_file:
        private_file.write(private_key)
    with open(dir_path / public_key_file_name, "wb") as public_file:
```

```
public_file.write(public_key)
```

Також скрипт створює під директорії, якщо вони не були створені в системі. Для ініціалізації розміру ключа оголошено окрему змінну RSA_KEY_SIZE, дефолтне значення якої 2048 байт. Такий розмір ключа вважається захищеним у сучасних реаліях, і його розмір не вплине негативно на швидкість роботи програми оскільки ним ми шифруємо лише ключ шифрування симетричного алгоритму. Після генерації ключів на цільовій машині цей скрипт можна сміливо видаляти, а приватний ключ переміщувати на змінний носій інформації. Наступним кроком імплементації було створення функціоналу, який міг би працювати з різними алгоритмами шифрування, реалізацію універсального ядра шифрування. Для цього я створив директорію src, так прийнято, щоб відділяти код проекту від допоміжних файлів, і на верхньому рівні репозиторію залишати лише файли точки входу у програму, і у ній розмістив файл Cryptographer.py який реалізовує ядро шифрування. Для реалізації були використані наступні імпорти:

```
import os
from inspect import iscoroutinefunction
from pathlib import Path
from typing import Callable
import multiprocessing
import asyncio
from concurrent.futures import wait
from concurrent.futures import ProcessPoolExecutor
```

Далі у цьому файлі оголошується клас з однойменною назвою, який реалізує конструктор класу (метод з назвою __init__.py), який приймає на вхід 5 значень, серед яких функція шифрування, функція дешифрування, відкритий і

закритий ключі, а також ліміт файлів для однопоточного шифрування, код наведено нижче

```
class Cryptographer:
    def __init__(
            self, path: str,
            # TODO: Replace by module/class with methods ?
            encryption_method: Callable,
            decryption_method: Callable,
            public_key_file_path: str,
            private_key_file_path: str,
            single_stream_files_limit: int = 50,
            ):
        self.path = Path(path)
        self.encryption_method = encryption_method
        self.decryption_method = decryption_method
        self.public_key_file_path = public_key_file_path
        self.private_key_file_path = private_key_file_path
        self.single_stream_files_limit = single_stream_files_limit
```

Також клас реалізує два основних метода encrypt і decrypt, для шифрування і розшифрування даних відповідно, які є обгортками для виклику функції _base зі специфічними аргументами, для шифрування і розшифрування відповідно. Код наведено нижче:

```
def encrypt(self) -> tuple[list, dict]:
    """ Main function for encryption """
    return self._base(self.public_key_file_path, self.encryption_method)
```

Функція decrypt(), по аналогії з функцією encrypt() викликає метод класу _base(). Різниця реалізації є список переданих функції аргументів. У випадку дешифрування ми передаємо у функцію метод який реалізує функціонал дешифрування разом з приватним ключем. Код наведено нижче:

```
def decrypt(self) -> tuple[list, dict]:
    """ Main function for decryption """
    return self._base(self.private_key_file_path, self.decryption_method)
```

Функція _base реалізує виклик шифрування і приймає на вхід ключ для здійснення операції і метод шифрування. Завдяки перевикористанню коду і застосування ООП такий функціонал вийшов доволі лаконічним і зрозумілим. Головна функція викликає ряд допоміжних функцій, для рекурсивного пошуку і сортування списку файлів, розподілу файлів по пулах задач, і створення процесів для запуска задач.

```
def _base(self, key: str, method: Callable) -> tuple[list, dict]:
        key: str = open(key).read()
        files: list = self._get_file_list(self.path)
        # sort all files by size
        files: list = sorted(files, key=lambda item: os.stat(item).st_size)
        pools = self._separate_files_by_pools(files)
        with ProcessPoolExecutor() as executor:
                futures = [executor.submit(Cryptographer._run_tasks, method, key, tasks_pool) for tasks_pool in pools]
        wait(futures)
        successful, exceptions = [], {}
        for f in futures:
            success, exc = f.result()
            successful.extend(success)
```

```
            exceptions.update(exc)
        return successful, exceptions
```

Назви всіх допоміжних функцій починаються з нижнього підкреслення і є угодою між програмістами про те, що такі методи є внутрішніми і використовуються всередині класу і виклик їх зовні не є правильним використанням класу. Функція _get_file_list() приймає на вхід шлях, і рекурсивно аналізує усі вкладені директорії та файли і повертає список усіх файлів. Код функцій наведено нижче:

```
@staticmethod
def _get_file_list(path: Path) -> list:
    file_list = []
    for root_dir, dirs, files in os.walk(path):
        for file in files:
            file_list.append(os.path.join(root_dir, file))
    return file_list
@staticmethod
```

Допоміжна функція _run_tasks() викликається з функції base() і отримує на вхід метод шифрування, ключ і також список завдань. Функція також аналізує тип отриманого методу. Якщо метод - це синхронна функція то під її виконання виділяється окремий процес. Якщо ж це асинхронна функція то створюється потік, в якому ініціалізується цикл подій, в який передається список усіх асинхронних задач. Код наведено нижче:

```
def _run_tasks(method: Callable, key: str, tasks: list) -> tuple[list, dict]:
    if iscoroutinefunction(method):
        loop = asyncio.new_event_loop()
        asyncio.set_event_loop(loop)
```

```
                successful, exceptions = loop.run_until_complete(Cryptographer._run_pool_tasks_async(method, key, tasks))
            else:
                successful, exceptions = Cryptographer._run_pool_tasks_sync(method, key, tasks)
            return successful, exceptions
```

Функція _separate_files_by_pools(), отримує список усіх файлів, попередньо посортованих за розміром і які потрібно зашифрувати, після чого усі ці файли рівномірно розподіляються на пули, кількість яких визначається за формулою: кількість ядер процесора * 2 + 1. Код наведено нижче:

```
    def _separate_files_by_pools(self, files: list) -> list[list, list]:
        if len(files) < self.single_stream_files_limit:
            return [files]
        cores: int = multiprocessing.cpu_count() * 2 + 1
        pools = [[] for _ in range(cores)]
        for index, file in enumerate(files):
            pool = index % cores
            pools[pool].append(file)
        return pools
```

Клас Cryptographer вміє працювати з потоками, процесами, синхронними і асинхронними задачами і є універсальним. Скрипт аналізує кількість ядер процесора і базуючись на цій інформації підкріплений формулою cpu_count * 2 + 1 створює процеси якісь запускають задачі шифрування. Процеси будуть створені якщо кількість файлів, які потрібно зашифрувати більша ніж значення змінної single_stream_files_limit, яка по дефолту дорівнює 50, в іншому випадку шифрування буде відбуватися в одному потоці. Якщо передані функції для шифрування і дешифрування є асинхронними (корутинами), то метода _base

запускає у кожному процесі свій, повністю незалежний асинхронний цикл подій (event loop) і передає всі завдання, після чого очікує їх результати. Також клас під капотом використовує обробники виключень, так звану конструкцію try -except для відловлення неочікуваних виключень, щоб помилка при якійсь дії не перервала виконання всієї програми і тим самим не поставили під загрозу конфіденційність усіх файлів. Код наведено нижче:

```
@staticmethod
async def _run_pool_tasks_async(method: Callable, key: str, tasks: list) -> tuple:
        results: tuple = await asyncio.gather(*[method(key, task) for task in tasks], return_exceptions=True)
    successful = []
    exceptions = {}
    for item, task in zip(results, tasks):
        if issubclass(type(item), Exception):
            exceptions[task] = item
        else:
            successful.append(item)
    return successful, exceptions
```

Функція _run_pool_tasks_sync(), запускає усі синхронні задачі послідовно, одна за одною, а також реалізує обгортку для перехоплення виключень і збору результатів шифрування. Результатом виконання функції є кортеж з двома елементами, перший з яких це список успішно виконаних завдань, а другий це словник в якому ключем виступає назва завдання, а значенням серилізований за допомогою модуля pickle об’єкт помилки. Код наведено нижче:

```
@staticmethod
def _run_pool_tasks_sync(method: Callable, key: str, tasks: list) -> tuple[list, dict]:
```

```
    successful = []
    exceptions = {}
    for task in tasks:
        try:
            successful.append(method(key, task))
        except Exception as e:
            exceptions[task] = e
    return successful, exceptions
```

Наступним кроком було створення другої точки входу, файлу run.py, який приймає аргументи командної строки і виконує функцію скрипта який може зашифрувати дані, наприклад в цілях тестування або ж розшифрувати, попередньо приховані файли. Для цього ми імпортуємо стандартну бібліотеку argparse а також наші допоміжні модулі і аналізуємо отримані програмною аргументи командної строки. Код наведено нижче:

```
import argparse
from src.Cryptographer import Cryptographer
from src.algorithms.sync_version import encryption_method, decryption_method
from src.config import public_key_file_path, private_key_file_path, TARGET_PATH
cryptographer = Cryptographer(TARGET_PATH, encryption_method, decryption_method, public_key_file_path, private_key_file_path)
if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Script so useful.')
    parser.add_argument("option")
    args = parser.parse_args()
    if args.option in ('en', 'encrypt'):
        cryptographer.encrypt()
```

```
    elif args.option in ('de', 'decrypt'):
        cryptographer.decrypt()
```

Наступна, і остання точка входу це файл entry.py який, як і попередні знаходиться у кореневому каталозі репозиторію і відповідає за ініціалізацію і запуск системи аудиту безпеки і реагування на інциденти. У файлі оголошується і запускається функція main(), за допомогою модуля asyncio, який реалізує асинхронність в Python. Функція імпортує тригери (у нашій реалізації їх чотири, при потребі їх можна вмикати, вимикати, додавати нові і т.д.). Реалізація кожного з тригерів є блокуючою, оскільки вони слухають події (повідомлення в телеграм, натиск гарячих клавіш, події файлової системи, тощо), тож для кожного з ним ми створюємо окремий потік, в якому і запускаємо прослуховування. Код функції main() наведено нижче:

```
    import sys
    import multiprocessing
    import time
    import asyncio
    from src.auto_remove import remove_yourself
    from src.config import triggers, auto_remove_trigger
    from src.Cryptographer import Cryptographer
    from run import cryptographer
    async def main(triggers: dict, cryptographer: Cryptographer) -> None:
        processes = []
        for trigger in triggers.values():
                        new_process = multiprocessing.Process(target=trigger,
name=trigger.process_name)
            processes.append(new_process)
            new_process.start()
        while True:
```

```
        # TODO: Add try except and logs to avoid error messages
        # TODO: Remove? No need to use in production ?
        time.sleep(0.5)
        if check_processes_is_killed(processes):
            stop_all_process(processes)
            cryptographer.encrypt()
            if auto_remove_trigger:
                remove_yourself()
            print('The script is complete...')
            sys.exit(0)
```

Після успішного запуску систем прослуховування подій ми перевіряємо кожен з запущений процесів прослуховування на існування за допомогою функції check_processes_is_killed(), і в разі, якщо хоча б один з них завершив свою роботу ми зупиняємо і решту, щоб уникнути повторних сигналів про завершення за допомогою функції stop_all_process(), і запускаємо шифрування файлів. Код функцій наведено нижче:

```
def stop_all_process(processes: list) -> None:
    for process in processes:
        if process.is_alive():
            process.terminate()
def check_processes_is_killed(processes: list) -> bool:
    return any(list(map(lambda process: not process.is_alive(), processes)))
```

Після успішного шифрування запускається процес видалення програмою самої себе і усіх слідів за собою. На Python це легко реалізується оскільки весь код модулів лише один раз завантажується в оперативну пам’ять, і в подальшій програмі читається саме з оперативної пам’яті, без звернення до файлової системи, тож щодних помилок не виникає. В цілях зручного тестування також

була добавлена умова if яка видаляє програму якщо змінна auto_remove_trigger має значення True. Головна функція запускається з конструкції if __name__ == '__main__'. Код наведено нижче:

```
if __name__ == '__main__':
    print(f'entry.py running...')
    asyncio.run(main(triggers, cryptographer))
```

Ця умова перевіряє, що цей файл був запущений, а не імпортований і тим самим дозволяє нам перевикористати код і об’єкти файлу у інших модулях програми без ризику задублювати програму у пам’яті. Код запуску головної функції наведено нижче:

3.3 Реалізація системи реагування

Для реалізації хендлерів було створено модуль handlers у директорії src. Всередині оголошено файл __init__,py, в якому оголошується клас AbstractHandler який наслідується від класу ABC, модуля abc, який реалізує угоду абстрактного класу. Всі класи які наслідуються від AbstractHandler повинні реалізувати метод __call__(), інакше буде викликано виключення TypeError: Can't instantiate abstract class <class_name> with abstract method __call__. Великий плюс в тому, що виключення буде викликане не в момент виклику не імплементованого методу, а при ініціалізації дочірнього класу. Код реалізації абстрактного класу наведено нижче:

```
from abc import ABC, abstractmethod
class AbstractHandler(ABC):
    process_name = None
    @abstractmethod
```

```
    def __call__(self, *args, **kwargs):
        pass
```

Заплановані хендлери можна умовно розділити на дві групи:

- автоматичного спрацювання;
    - Hotkeys;
    - Telegram bot;
- ініційовані користувачем;
    - Usb_access_key;
    - Honeypot.

Імплементацію розпочато з групи хендлерів ініційованих користувачем.

HotkeyHandler який реагує на користувацьку комбінацію клавіш базується на використанні бібліотеки pynput яка фіксує усі натиски клавіш. Комбінацію можна вказати при ініціалізації класу. Сам клас інкапсулює в собі 3 методи, два з яких магічні, __init__() і __call__() і _on_activate() який буде викликатися лише внутрішнім методом __call__(). Оскільки згідно умови реалізації і принципу SOLID, об’єкт повинен мати тільки одну зону відповідальності, то у класі реалізується метод, який закриває потік у разі спрацювання тригера, а вже основна програма моніторить стан потоку і запускає шифрування у разі завершення одного з них. Код наведено нижче:

```
import sys
from pynput import keyboard
from src.handlers import AbstractHandler
class HotkeyHandler(AbstractHandler):
    process_name = 'hotkey_listener'
    def __init__(self, hotkey):
        self.hotkey = hotkey
```

Запуск системи аудиту відбувається за допомогою магічної функції __call__(), яка дозволяє викликати об’єкт класу як звичайну функцію, і тим самим реалізувати один стандарт виклику. Код наведено нижче:

```
def __call__(self, *args, **kwargs):
    """Start listen hotkey"""
    print('Start listen hotkey')
        with keyboard.GlobalHotKeys({self.hotkey: self._on_activate}) as listener:
            listener.join()
```

Функція _on_activate() запускається при активації гарячої клавіші, і ініціює завершення потоку.

```
def _on_activate(self):
    print('Global hotkey activated!')
    print(f'{self.hotkey} pressed!')
    print('The program closes')
    sys.exit(0)
```

Наступний хендлер буде відповідати за старт шифрування чутливих даних при отриманні ключової команди у телеграм-бота. Клас хендлера TelegramBotHandler реалізований у файлі telegram_bot.py який знаходить в пакеті src/handlers/. Головною сторонньою бібліотекою для реалізації було використано aiogram. Це асинхронна бібліотека побудована на базі asyncio і aiohttp, яка має зручний інтерфейс і відмовостійкість. Код реалізації наведено нижче:

```
import sys
import asyncio
```

```
from aiogram import executor
from aiogram import Bot
from aiogram.dispatcher.dispatcher import Dispatcher
from aiogram import types
from src.handlers import AbstractHandler
class TelegramBotHandler(AbstractHandler):
    process_name = 'telegram_bot'
```

Метод __init__() є конструктором класу і приймає 3 аргументи, токен телеграм бота, айді чату адміністратора необов’язковий аргумент який вказує ключову команду для запуску шифрування. Код наведено нижче:

```
    def __init__(self, bot_token: str, admin_id: str, commands: list = ('encrypt',)):
        self.bot = Bot(token=bot_token)
        self.dp = Dispatcher(self.bot)
        self.admin_id = admin_id
        self.dp.message_handler(commands=commands)(self.start_encrypt)
```

Запуск системи аудиту відбувається за допомогою магічної функції __call__(), яка дозволяє викликати об’єкт класу як звичайну функцію, по аналогії з класом HotkeyHandler реалізується один стандарт виклику. Код наведено нижче:

```
    def __call__(self, *args, **kwargs):
        loop = asyncio.new_event_loop()
        asyncio.set_event_loop(loop)
        executor.start_polling(self.dp, on_startup=self.on_startup)
    async def on_startup(self, *args, **kwargs):
```

```
        me = await self.bot.me
                    # To avoid triggering on a recently sent message (https://github.com/aiogram/aiogram/issues/418)
        await self.bot.delete_webhook(drop_pending_updates=True)
            await self.bot.send_message(self.admin_id, f'<b>{me.first_name} ({me.username})</b> start!', parse_mode='HTML')
    async def start_encrypt(self, message: types.Message):
        await message.answer(f"Encryption start!")
        self.dp.stop_polling()
        await self.dp.wait_closed()
        print('Bot operation completed succeed!')
        sys.exit(0)
```

Наступний хендлер базується на реагуванні на події файлової системи для певних директорій, і відноситься до класу автоматичних, тобто для його активації не потрібно жодних дій з боку користувача. При ініціалізації він приймає аргумент з шляхом до ключового файлу і шлях до директорії, події в якій потрібно моніторити. Після запуску він прослуховує всі події файлової системи для певної директорії, і якщо фіксує подію, то запускає рекурсивний пошук ключового файлу на змінних носіях інформації (флеш-пам’ять, зовнішні жорсткі диски, CD-диски, тощо). Усі ці пристрої монтуються в каталог /media/. В разі, якщо файл знайдений, то система ігнорує подію, інакше завершується. Пошук файла відбувається по його хеш-сумі за допомогою функції sha1 з вбудованої бібліотеки hashlib. Прослуховування подій файлової системи відбувається за допомогою бібліотеки pyinotify. Для цього було реалізовано клас FileSystemEventHandler який наслідується від pyinotify.ProcessEvent, і перевизначає дії при подіях файлової системи. Цей клас винесено в окремий файл inotify_handler.py, оскільки цю бібліотеку буде використано і для honeypot хендлера. Специфічний код хендлера реалізовано у файлі usb_key.py в каталозі src/handlers. У файлі оголошується клас _USBAccessKeyFileSystemEventHandler,

який реалізує ряд функцій для відслідковування подій файлової системи, пошуку файла-ключа. Перша за все клас ініціалізується за допомогою функції конструктора __init__() яка приймає шлях до файла ключа, як обов'язковий аргумент і два необов'язкових аргумента для налаштування рекурсивного пошуку і фільтрування файлів по їх розширенні. У тілі конструктора ми викликаємо метод hash_file() який отримує шлях до файлу а повертає хеш-суму цього файлу. Код методів наведено нижче:

```
class _USBAccessKeyFileSystemEventHandler(FileSystemEventHandler):
    def __init__(self, key_filename: str, recursive_find_key: bool = True, filtering_by_file_extension: bool = True):
        self.filtering_by_file_extension = filtering_by_file_extension
        self.key_file_extension = Path(key_filename).suffix
        self.hash_sum = _USBAccessKeyFileSystemEventHandler.hash_file(key_filename)
        self.recursive = recursive_find_key
        super().__init__()
```

Метод hash_file() приймає строку, шлях до файла і робить отримує хеш суму файлу для подальшого порівняння і ідентифікації файлу ключа. Код наведено нижче:

```
@staticmethod
  def hash_file(filename: str) -> str:
    """Хеш сума"""
    h = hashlib.sha1()
    with open(filename, 'rb') as file:
      chunk = 0
      while chunk != b'':
```

```
                chunk = file.read(4096)
                h.update(chunk)
        return h.hexdigest()
```

Також клас реалізує методи trigger_actions(), find_valid_usb_key() і get_external_disks() для отримання списку сторонніх дисків, пошуку ключового файлу на сторонніх дисках та активації тригера шифрування. Код наведено нижче:

```
class _USBAccessKeyFileSystemEventHandler(FileSystemEventHandler):
    def trigger_actions(self) -> None:
        process_name = current_process().name
                                                print(f'{process_name} : Event
{inspect.getouterframes(inspect.currentframe())[1].function} register!')
        if not self.find_valid_usb_key():
            sys.exit(0)
        print(f'Key on USB device successfully find! Anxiety relief!')
```

Метод find_valid_usb_key() інкапсулює логіку пошуку файлу ключа серед списку змонтованих сторонніх носіїв інформації підключених через інтерфейс USB. Код наведено нижче:

```
    def find_valid_usb_key(self) -> bool:
                                                external_disks =
_USBAccessKeyFileSystemEventHandler.get_external_disks()
        for disk in external_disks:
            try:
                for root_dir, dirs, files in os.walk(disk.mountpoint):
                    for file in files:
                        file_path = os.path.join(root_dir, file)
```

```
                    if self.filtering_by_file_extension and not Path(file_path).suffix
== self.key_file_extension:
                        continue
                                                                                    if
_USBAccessKeyFileSystemEventHandler.hash_file(file_path) == self.hash_sum:
                        return True
                except:
                    return False
            return False
        @staticmethod
```

Метод get_external_disks() повертає список усіх жорстких дисків підключених через інтерфейс USB.

```
        def get_external_disks() -> list:
            drps = psutil.disk_partitions()
                                external_disks = [disk for disk in drps if
disk.mountpoint.startswith('/media/')]
            return external_disks
```

Також у файлі оголошується клас USBAccessKeyHandler який є хендлером системи реагування і використовує _USBAccessKeyFileSystemEventHandler як дескриптор для обробки подій файлової системи. Код наведено нижче:

```
        class USBAccessKeyHandler(AbstractHandler):
            process_name = 'usb_access_key'
            def __init__(self, key_filepath: str, target_path: str, recursive: bool = True):
                wm = WatchManager()
                wm.add_watch(target_path, ALL_EVENTS, rec=recursive)
```

```
                fsh = _USBAccessKeyFileSystemEventHandler(key_filepath,
recursive_find_key=True)
            self.notifier = Notifier(wm, fsh)
```

Останній хендлер реалізовує функціонад пастки. Він так само як і USBAccessKeyHandler прослуховує події файлової системи і реагує на них, але на відміну від попереднього, він приймаєш шлях до директорії, яку наш користувач ніколи не відкриє, тому що буде знати, що не паста, а зловмисник ні. Варто розмістити цю директорію, або ярлик на неї на видному місці (робочий стіл, коренева папка диску, тощо) і надати назву, яка може зацікавити зловмисника (наприклад “Паролі”, “Секретні документи”, тощо). Код обробника наведено нижче:

```
from src.handlers import AbstractHandler
from src.inotify_helper import FileSystemEventHandler, WatchManager,
Notifier, ALL_EVENTS
class HoneypotHandler(AbstractHandler):
    process_name = 'honeypot'
    def __init__(self, path: str, recursive: bool = False):
        wm = WatchManager()
        wm.add_watch(path, ALL_EVENTS, rec=recursive)
        fsh = FileSystemEventHandler()
        self.notifier = Notifier(wm, fsh)
```

Метод __call__(), ініціалізує і запускає цикл реагування на події файлової системи.

```
    def __call__(self, *args, **kwargs):
        """Start listen file system events"""
```

```
self.notifier.loop()
```

3.4 Файл налаштувань

Наступний крок це імплементація файлу налаштувань. Це єдиний файл в системі який повинен редагуватися кінцевим користувачем системи. В цьому файлі ми використовуємо модуль environs для імпортування змінної токена бота та id адміністратора (або адміністративного чату) в телеграмі. Ці змінні винесені як змінні оточення тому що для кожного користувача вони точно будуть унікальні. Інші ж змінні можуть використовуватись без змін, це наприклад шлях до файла відкритого ключа, розмір асиметричного ключа шифрування, цільовий шлях до файлів, комбінація гарячих клавіш, тощо. А також у цьому файлі реалізується функція яка ініціює тригери з врахування операційної системи. Система розроблялася для ОС Linux, проте під час використовувалися найкращі практики для можливості запуску системи і на інших операційних системах (MacOS, Windows, тощо). Єдиної відмінністю буде відсутність тригерів HoneyPotHandler і USBAccessKeyHandler, оскільки вони опираються на бібліотеку inotify, яка в свою чергу спирається на особливості реалізації файлової системи в ОС Linux, але при бажанні можна зробити аналогічний механізм і в інших операційних системах, адеж протокол дескриптора використовується всюди. Це стандарт взаємодії з файловою системою. Код файлу розділений на блоки. Перший блок імпортує залежності і зчитує усі змінні оточення. Код наведено нижче:

```
import sys
from pathlib import Path
from environs import Env
env = Env()
```

```
env.read_env()
```

Наступний блок файлу налаштувань виділений коментарем та відповідає за налаштування хендлера взаємодії з телеграм-ботом. Код наведено нижче:

```
# ----------------------------- Telegram Bot Settings -----------------------------
BOT_TOKEN = env.str("TOKEN")
ADMIN_ID = env.str("admin_id")
TELEGRAM_BOT_COMMANDS = ['encrypt']
```

Наступний блок відповідає за налаштування інших хендлерів а також алгоритмів шифрування. Код наведено нижче:

```
keys_dir_path = '_keys'
public_key_file_name = 'public.pem'
private_key_file_name = 'private.pem'
RSA_KEY_SIZE = 2048     # bits
public_key_file_path: str = str(Path(keys_dir_path) / public_key_file_name)
private_key_file_path: str = str(Path(keys_dir_path) / private_key_file_name)
# the path to the folder where you want to encrypt the files
TARGET_PATH = 'test_dir'
# Hotkey for encrypt
HOTKEY = '<ctrl>+<alt>+.'
auto_remove_trigger: bool = False
```

Також у файлі ініціалізується і викликається функція initialize_triggers(), яка оголошує всі хендлери для системи реагування, спираючись на версію операційної системи, оскільки деякі з них недоступні для Windows. Код функції наведено нижче:

```
def initialize_triggers():
    """Return dict of triggers"""
    from src.handlers.telegram_bot import TelegramBotHandler
    from src.handlers.hotkeys import HotkeyHandler
    _triggers = {
        # Detected by user
        'telegram_bot': TelegramBotHandler(BOT_TOKEN, ADMIN_ID,
TELEGRAM_BOT_COMMANDS),
        'hotkey': HotkeyHandler(HOTKEY)}
    if sys.platform not in ('win32', 'cygwin'):
        from src.handlers.honeypot import HoneypotHandler
        from src.handlers.usb_key import USBAccessKeyHandler
        # Automation detection
        _triggers['honeypot'] = HoneypotHandler(TARGET_PATH)
     _triggers['usb_access_key'] = USBAccessKeyHandler('/tests/fixtures/cat.jpeg',
    TARGET_PATH)
    return _triggers
    triggers = initialize_triggers()
```

3.5 Імплементація пакетів шифрування і дешифрування даних

Остання відносно незалежна частина системи це алгоритми шифрування. Було досліджено симетричні і асиметричні системи шифрування. Оскільки розробка системи велася з урахуванням теоретичної можливості повної втрати контролю над пристроєм, ми не можемо зберігати ключ розшифрування даних на пристрої. Тому найкращим вирішенням було використання асиметричного алгоритму шифрування з великим розміром ключа, але тоді ми стикаємо з проблемою асиметричного шифрування, а саме з високою обчислювальною

складністю і тим самим повільнішим процесом шифрування, що може бути критичним фактором на великій кількості даних, адже зловмисник може встигнути отримати доступ до певної частини незашифрованих даних. Тому ідеальним рішенням буде використання гібридного шифрування, а саме шифрувати всі данні випадковим чином згенерованим ключем, за допомогою алгоритму AES, а цей ключ шифрувати завчасно згенерованим відкритим ключем алгоритму RSA. Отже ми зможемо зберігати приватний ключ на іншій системі і отримувати переваги швидкості шифрування алгоритму AES, і бути впевненим, що нашій дані в безпеці.

Для тестування і порівняння продуктивності я написав два алгоритма які базуються на гібридному методі шифрування з відмінністю того, що одина з реалізацій синхронна і виконується послідовно, а інша асинхронна, і виконується асинхронно в циклі подій, з використанням асинхронним бібліотек для читання файлів з диску. Сам процес шифрування даних це задача яка навантажує центральний процесор комп'ютера, а от отримання, читання цих даних з файлу і запис у файл це блокуюча операція (Input/Output bound), яка не навантажує процесор і чекає на відповідь від файлової системи, тож на цьому етапі при використанні асинхронності і неблокуючої бібліотеки доступу до даних можна отримати суттєвий приріст продуктивності на великій кількості файлів.

Перший алгоритм з синхронним виконанням реалізовано у директорії src/algorithms у файлі sync_version.py. У файлі оголошується дві функції: encryption_method і decryption_method, які відповідно шифрують і розшифровують дані. Сигнатури функції ідентичні і приймають на вхід два аргументи:

- key - ключ який буде використовуватись для шифрування або розшифрування даних, типу string;
- file_path - змінна типу pathlib.Path яка вказує повний шлях до цільового файла.

Код функції encryption_method() наведено нижче:

```
import os
from pathlib import Path
from Crypto.PublicKey import RSA
from Crypto.Random import get_random_bytes
from Crypto.Cipher import AES, PKCS1_OAEP
```

Функція encryption_method приймає на вхід ключ шифрування і абсолютний шлях до цільового файлу. Отримує набір з 16 випадкових байтів за допомогою функції get_random_bytes(), який використовується як ключ шифрування для алгоритму AES. За допомогою цього алгоритму відбувається шифрування файлу, після чого в кінець файлу добавляється ключ шифрування AES, зашифрований відкритим ключем RSA. У результатом успішного завершення функції повертається шлях до файлу.

```
def encryption_method(key: str, file_path: Path) -> str:
    output_file_path = f"{file_path}.bin"
    key: RSA.RsaKey = RSA.import_key(key)
    session_key: bytes = get_random_bytes(16)
    # Encrypt the session key with the public RSA key
    cipher_rsa: PKCS1_OAEP.PKCS1OAEP_Cipher = PKCS1_OAEP.new(key)
    enc_session_key: bytes = cipher_rsa.encrypt(session_key)
    with open(file_path, "rb") as input_file, open(output_file_path, "wb") as output_file:
        data = input_file.read()
        # Encrypt the data with the AES session key
        cipher_aes = AES.new(session_key, AES.MODE_EAX)
        cipher_text, tag = cipher_aes.encrypt_and_digest(data)
        [output_file.write(x) for x in (enc_session_key, cipher_aes.nonce, tag, cipher_text)]
```

```
    os.remove(file_path)
    print(f"File {file_path} encrypted!")
    return str(file_path)
```

Обидві функції повертають змінну file_path у разі успішного завершення функції або ж породжуть виняток який піднімається вверх по стеку.

Функція decryption_method() має аналогічну сигнатуру і виконує зворотні дії. Код наведено нижче:

```
def decryption_method(key: str, file_path: Path) -> str:
    output_file_path = str(file_path)[:-4]
    private_key = RSA.import_key(key)
      with open(file_path, "rb") as input_file, open(output_file_path, "wb") as output_file:
          enc_session_key, nonce, tag, ciphertext = [input_file.read(x) for x in (private_key.size_in_bytes(), 16, 16, -1)]
        cipher_rsa = PKCS1_OAEP.new(private_key)
        session_key = cipher_rsa.decrypt(enc_session_key)
        cipher_aes = AES.new(session_key, AES.MODE_EAX, nonce)
        data = cipher_aes.decrypt_and_verify(ciphertext, tag)
        output_file.write(data)
    os.remove(file_path)
    print(f"File {file_path} decrypted!")
    # TODO: Change file_path?
    return str(file_path)
```

Другий файл знаходиться за ідентичним шляхом і має назву async_version.py.

У цій реалізації було використано сторонню бібліотеку aofiles для реалізації асинхронної взаємодії з файловою системою, і звичайно самі функції

тепер виступають в ролі корутин і оголошуються за допомогою конструкції async def, а всередині себе використовуються асинхронні менеджери контексту з використанням ключого слова await який запускає на виконання корутину, добавляє її в існуючий цикл подій і передає керування іншій корутині, паралельно з певним таймаутом опитуючи корутину на наявність результатів виконання функції. Використання асинхронних менеджерів контексту зображено нижче:

```
async with aiofiles.open(file_path, "rb") as input_file:
    data = await input_file.read()
async with aiofiles.open(output_file_path, "wb") as output_file:
    # Encrypt the data with the AES session key
    cipher_aes = AES.new(session_key, AES.MODE_EAX)
    cipher_text, tag = cipher_aes.encrypt_and_digest(data)
      [await output_file.write(x) for x in (enc_session_key, cipher_aes.nonce, tag, cipher_text)]
    await aiofiles.os.remove(file_path)
```

Асинхронну функцію шифрування реалізовано під назвою encryption_method() у файлі src/algorithms/async_version.py. Код наведено нижче:

```
async def encryption_method(key: str, file_path: Path) -> str:
    output_file_path = f"{file_path}.bin"
    key: RSA.RsaKey = RSA.import_key(key)
    session_key: bytes = get_random_bytes(16)
    # Encrypt the session key with the public RSA key
                    cipher_rsa: PKCS1_OAEP.PKCS1OAEP_Cipher = PKCS1_OAEP.new(key)
    enc_session_key: bytes = cipher_rsa.encrypt(session_key)
    async with aiofiles.open(file_path, "rb") as input_file:
```

```
            data = await input_file.read()
        async with aiofiles.open(output_file_path, "wb") as output_file:
            # Encrypt the data with the AES session key
            cipher_aes = AES.new(session_key, AES.MODE_EAX)
            cipher_text, tag = cipher_aes.encrypt_and_digest(data)
              [await output_file.write(x) for x in (enc_session_key, cipher_aes.nonce,
tag, cipher_text)]
        await aiofiles.os.remove(file_path)
        print(f"File {file_path} encrypted!")
        return str(file_path)
```

У тому ж модулі знаходиться асиметричний алгоритм дешифрування. Код наведено нижче:

```
    async def decryption_method(key: str, file_path: Path) -> str:
     output_file_path = str(file_path)[:-4]
     private_key = RSA.import_key(key)
     # Important: This version v1 don't work,
     # cause aiofiles doesn't support opening two files at the same context manager
```

Асинхронне читання і шифрування файлу відбувається в контекстному менеджері async with aiofiles.open(). Код наведено нижче.

```
        async with aiofiles.open(file_path, "rb") as input_file:
            enc_session_key, nonce, tag, ciphertext = [await input_file.read(x) for x in
(private_key.size_in_bytes(), 16, 16, -1)]
        async with aiofiles.open(output_file_path, "wb") as output_file:
            cipher_rsa = PKCS1_OAEP.new(private_key)
            session_key = cipher_rsa.decrypt(enc_session_key)
            cipher_aes = AES.new(session_key, AES.MODE_EAX, nonce)
```

```
            data = cipher_aes.decrypt_and_verify(ciphertext, tag)
            await output_file.write(data)
        await aiofiles.os.remove(file_path)
        print(f"File {file_path} decrypted!")
        return str(file_path)
```

Для зручності імпорту в директорії algorithm створено файл __init__.py який виконуєтсья при першому імпорті модуля і імпортує з вкладених модулів функції. Це використовується для спрощення вкладеності.

```
import src.algorithms.sync_version as sync_algorithms
import src.algorithms.async_version as async_algorithms
```

3.6 Імплементація автоматизованих тестів.

Оскільки цей проект з відкритим вихідним кодом і в перспективі передбачає долучення до розробки зацікавлених користувачів важливо написати тести, які користувач може запустити і впевнитись, що програма працює коректно. Також тести допомагають зрозуміти як працює певний функціонал, якщо документації недостатньо. Як і прийнято в open-source спільноті тести знаходяться в репозиторії проекту в папці tests.

Для зручності реалізації тестів було використано стронню бібліотеку pytest, яка надає потужний функціонал для підготовки тестового середовища, запуску, відладки, та підтримки тестів. Також були використані фікстури та моки.

Fixtures - це функції, що виконуються pytest до (а іноді і після) фактичних тестових функцій. Код у фікстурі може робити все, що вам потрібно. Ви можете використовувати Fixtures, щоб отримати набір даних для тестування. Ви можете використовувати Fixtures, щоб отримати систему у відомому стані перед

запуском тесту. Fixtures також використовуються для отримання даних для кількох тестів.

unittest.mock - це бібліотека для тестування на Python. Це дозволяє вам замінити частини вашої тестової системи на фіктивні об’єкти та зробити твердження про те, як вони використовувалися.

Тести порівнюють очікувану поведінку автоматизованих хендлерів з фактичною у файлі tests/test_automation_handlers.py. Код наведено у додатку А.

У файлі tests/test_encrypt_decrypt.py тестуються всі методи шифрування і дешифрування реалізовані у проекті. Код модуля наведено у додатку А.

Запустити усі файли можна за допомогою команди : “ pytest “ за умови попередньо створеного віртуального оточення і встановлених усіх залежностей.

3.6 Тестування швидкодії

Фінальний етап після тестування проекту це проведення замірів швидкості виконання скрипта шифрування і розшифрування. Результативність в бенчмарках залежить від багатьох змінних, наприклад швидкодія процесора, кількість запущених фонових процесів, об’єм оперативної пам’яті, швидкість доступу до постійної пам’яті, версії інтерпретатора, рівень температури в кімнаті, тощо. Тож синтетично однакові тести провести дуже складно, але реально. Для цього потрібно прогнати тестовий сценарій N-разів на одній і тій ж самій машині. Для тестування я використовував персональний ноутбук з характеристиками які зображені на рисунку 3.1

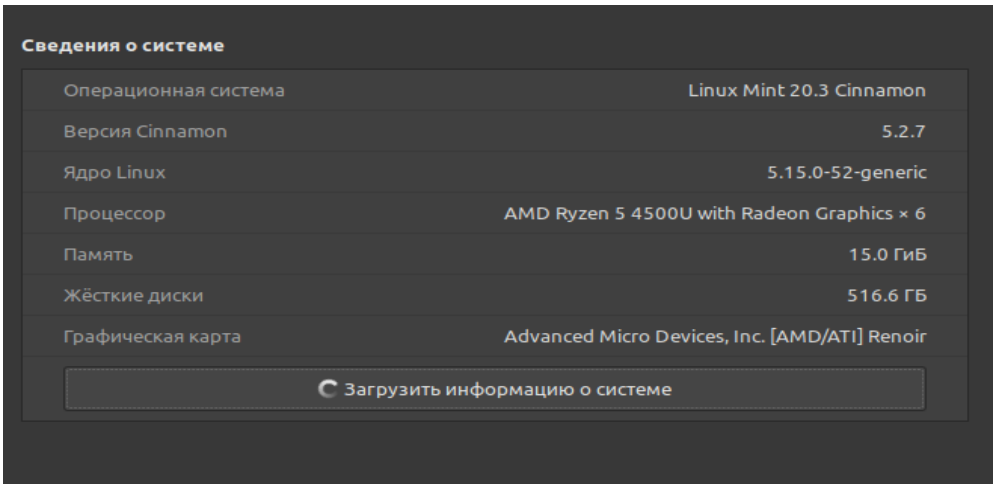


Рисунок 3.1 - Характеристики тестового стенду

Було вирішено вибрати ключові змінні в бенчмарках і провести різні тестування з їх урахуванням і вибором оптимальних значень. Отже впродовж тестування я вважаю є сенс протестувати швидкість при зміні кількості процесів, версії Python, синхронного і асинхронного шифрування. Всі тести варто проводити на великій кількості файлів (даних), щоб різниця була більш відчутною, адже якщо програма працює короткий час, то на неї більший вплив мають сторонні фонові процеси операційної системи (пошук оновлень, кешування, системі виклики, тощо).

Для генерації тестових файлів було реалізовано допоміжну функцію з наступною сигнатурою: create_file(filename: str, size: int, data: str = 'My test data\n') -> None.

Перш за все варто протестувати різницю у швидкодії між версіями Python, оскільки розробники заявляють, що остання версія Python отримала значне збільшення продуктивності, і головний вектор розвитку цієї мови програмування це збільшення швидкості виконання задач залежний від центрального процесору.

Було проведено два тести в наступних умовах:

- SSD;
- 100 тестових файлів розміром по 10 Мегабайт;
- 3 тестових запуска для усереднення результатів;

- 1 процес виконання;
- Синхронний і асинхронний запуск.

Єдиною відмінністю була версія інтерпретатора. Перший замір базується на версії інтерпретатора 3.9.15 і показав наступні результати:

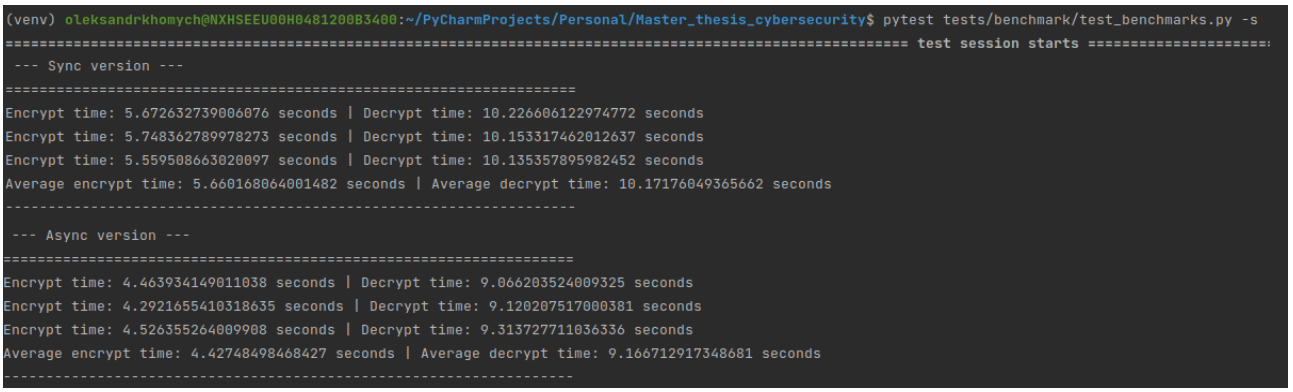
Середній час шифрування синхронним алгоритмом: 5.66 секунд.

Середній час дешифрування синхронним алгоритмом: 10.17 секунд.

Середній час шифрування асинхронним алгоритмом: 4.66 секунд.

Середній час дешифрування асинхронним алгоритмом: 9.16 секунд

Звіт бенчмарку зображено на рисунку 3.2



3.2 - Звіт бенчмарку версії 3.9.15

Наступний замір базується на версії інтерпретатора 3.11.0 і показав наступні результати:

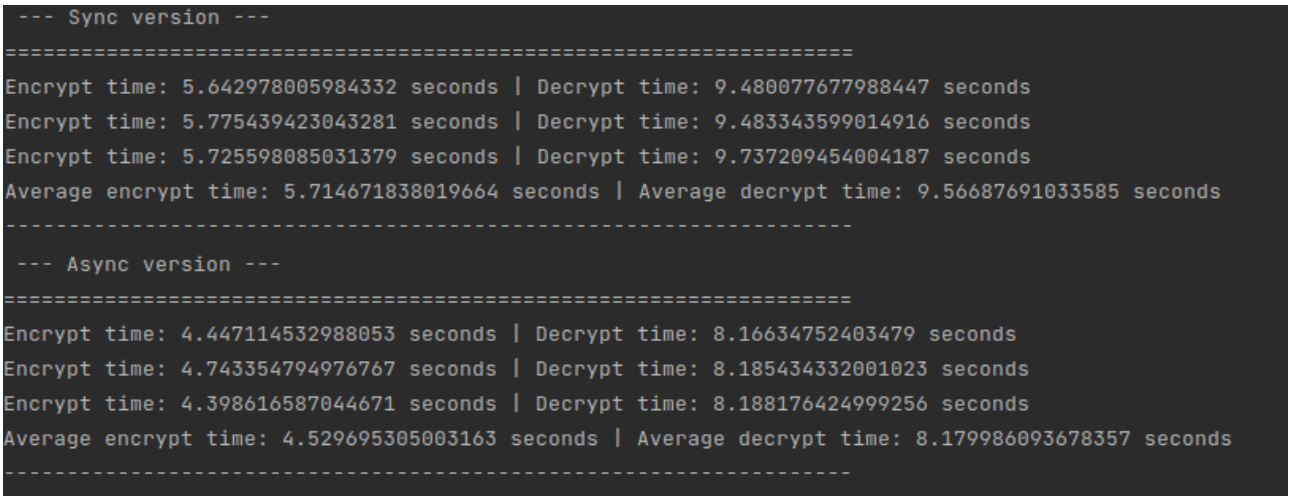
Середній час шифрування синхронним алгоритмом: 5.71 секунд.

Середній час дешифрування синхронним алгоритмом: 9.56 секунд.

Середній час шифрування асинхронним алгоритмом: 4.52 секунд.

Середній час дешифрування асинхронним алгоритмом: 8.17 секунд

Звіт бенчмарку зображено на рисунку 3.3



3.3 - Звіт бенчмарку версії 3.11.0

Заміри демонструють схожі результати але на версії інтерпретатора 3.11.0 спостерігається стабільний приріст у всіх бенчмарках на 3-15%

Отже по результатах тестів ми бачимо, що на 100 файлах які шифруються в один поток є приріст продуктивності близько 25% при використанні асинхронного шифрування і 16% при дешифруванні. Тут важливо згадати, що тестування відбувалося на швидкому SSD диску, і на системі з HDD різниця буде суттєвішою і може показувати приріст продуктивності до 100%.

Замір продуктивності з використанням багатопоточності не проводились оскільки це не доцільно, адже окремі процеси працюють повністю незалежно і заміри будуть залежати лише від кількості ядер процесора.

ВИСНОВКИ

Досліджено процес реагування на інциденти коли злочинець отримав фізичний або віртуальний доступ до пристрою а також проаналізовано існуючі рішення захисту персональних даних. Досліджено алгоритми шифрування з симетричним і асиметричним ключем, їхні переваги, недоліки а також гібридні методи шифрування для уникнення недоліків кожної з систем.

Розроблено систему аудиту безпеки та реагування на інциденти комп'ютерних систем на базі ОС Linux. Запропонована система є останнім бар’єром для зловмисника при отриманні несанкціонованого віртуального або фізичного доступу до девайсу. Реалізована гнучка модульна система реагування на події в автоматичному або ручному режимі, а також реалізовано два гібридних алгоритма шифрування на основі злиття AES і RSA на базі синхронного та асинхронного шифрування в мові програмування Python, які визначаються високою швидкодією про що свідчать результати замірів тестів продуктивності.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Касянчук М., Карпінський М., Казмірчук С. Методологія опрацювання багаторозрядних чисел в асиметричних криптосистемах/ Захист інформації. 2019. Т.21, №2. С. 65- 73.

2. Краснобаев В.А., Янко А.С., Кошман С.А., Курчанов В.Н., Бендес Ю.П. Расчет и сравнительный анализ производительности компьютерной системы обработки целочисленных данных, представленных в системе остаточных классов. Системи обробки інформації. 2015. В 3 (128). С. 57-61.

3. Zadiraka V., Yakymenko I., Kasianchuk M., Ivasiev S. Theoretical and numerical Krestenson's basis and its application to problems of cryptographic protection and factorization of multidigit numbers, Computer technologies in information security: collective monograph, By edited V.Zadiraka, Ya.Nykolaichuk, Ternopil: Kart-blansh, 2015. P. 216-260. Ch. 5.

4. Ivasiev S., Yakymenko I., Kasianchuk M., Shevchuk R., Karpinski M., Gomotiuk O. Effective algorithms for finding the remainder of multi-digit numbers. Advanced Computer Information Technology (ACIT–2019): Proceedings of the International Conference. Ceske Budejovice (Czech Republic). 2019. P. 175-178

5. Ivasiev S., Yakymenko I., Kasianchuk M., Shevchuk R., Tymoshenko L. The Method of Factorizing Multi-Digit Numbers Based on the Operation of Adding Odd Numbers. Advanced Computer Information Technology (ACIT–2018): Proceedings of the International Conference. Ceske Budejovice (Czech Republic). 2018. P. 232-235.

6. Rajba T. Klos-Witkowska A., Ivasiev S., Yakymenko I., Kasianchuk M. Research of Time Characteristics of Search Methods of Inverse Element by the Module. Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications: Proceedings of the 2017 IEEE 9 th

International Conference. Bucharest, Romania. V.1. September, 2017. P.82– 85 (Scopus).

7. Касянчук М., Карпінський М., Казмірчук С. Методологія опрацювання багаторозрядних чисел в асиметричних криптосистемах/ Захист інформації. 2019. Т.21, №2. С. 65- 73.

8. Касянчук М.М., Якименко І.З., Івасьєв С.В., Момотюк О.В. Експериментальне дослідження програмної реалізації методів пошуку оберненого елемента за модулем. Інформатика та математичні методи в моделюванні. 2017. Т.7, №3. С. 178–186.

9. Глинська М.Л., Лісковецький Д.В., Івасьєв С.В. Збірник матеріалів наукової конференції «Кібербезпека та комп'ютерноінтегровані технології» (КБКІТ - 2019). –Тернопіль. –2019. С. 21-24.

10. Стенли Б. Липпман. С++ для начинающих, Пер. с англ. 2тт. Москва: Унитех. Рязань: Гэлион, 1992, 304-345 с.

11. Николайчук Я.М. Теорія джерел інформації. Тернопіль: ТзОВ„Терно–граф", 2010. – 536 с.

12. Николайчук Я.Н., Божнев В.П., Зевелев С.Я. Применение методов теории чисел для сжатия измерительной информации в системах телеконтроля процессов бурения. Материалы Всесоюзной конференции молодых ученых нефтяных ВУЗов. М.:МИНХиГП, 1975. С. 134-138.

13. Якименко І.З., Касянчук М.М., Тимошенко Л.М., Івасьєв С.В., Николайчук Я.М. Алгоритм знаходження системи модулів модифікованої досконалої форми системи залишкових класів, Матеріали МНПК СІЕТ.- Одеса. 2014. С. 115-117.

14. Ященко В.В., Варновский Ю.В. , Нестеренко Г.А. Кабатянский Введение в криптографию. Питер, 2001. 271 с.

15. Kozaczko D., Kasianchuk M., Yakymenko I., Ivasiev S.Vector Module Exponential in the Remaining Classes System. Proceedings of the 2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications

(IDAACS–2015). Warsaw, Poland. V.1, September, 2015. P.161–163.

16. Lakhani G. Some Fast Residual Arithmetic Adders. International Journal of Electronics. - 1994. - P. 225-240.

17. Lenstra H. W. Divisors in residue classes. Math. Comput. 1984. Vol. 42. N 165. P.331-340.

18. Nykolaychuk Ya. M., Kasianchuk M.M., Yakymenko I.Z., Theoretical Foundations of the Modified Perfect form of Residue Number System. Cybernetics and Systems Analysis, 2016, P. 219-223.

19. Tsmots I., Teslyuk V., Teslyuk T., Ihnatyev I. Basic Components of Neuronetworks with Parallel Vertical Group Data Real-Time Processing. Advances in Intelligent Systems and Computing II. CSIT 2017. Advances in Intelligent Systems and Computing, vol. 689, Springer, Cham. P. 558 – 576.

20. Yakymenko I., Kasyanchuk M., Nykolajchuk Ya. Matrix Algorithms of Processing of the Information Flow in Computer Systems Based on Theoretical and Numerical Krestenson's Basis. Proceedings of the X–th International Conference "Modern Problems of Radio Engineering, Telecommunications and Computer Science" (TCSET–2010). L'viv–Slavske. 2010. – P.241.

21. Марк Лутц. Программирование на Python / Пер. с англ. – 4-е изд. – СПб.: СимволПлюс, 2011. – Т. II. 3. Марк Лутц. Изучаем Python, 4-е издание. – Перевод с английского. – СПб.: СимволПлюс, 2010. – 1280 с.

22. Дэвид М. Бизли. Python. Подробный справочник, 4-е издание. – Перевод с английского. – СПб.: Символ-Плюс, 2010. – 864 с

23. Марк Саммерфилд. Программирование на Python 3. Подробное руководство. – Перевод с английского. – СПб.: Символ-Плюс, 2009.– 608 с.

24. Ноа Гифт, Джереми М. Джонс. Python в системном администрировании UNIX и Linux. – Перевод с английского. – СПб.: Символ-Плюс, 2009. 512 с

25. Бизли, Дэвид М. Язык программирования Python. Справочник. – К.: ДиаСофт, 2000. – 336 с.
26. Сузи Р. А. Python. Наиболее полное руководство (+CD). – СПб.: БХВ-Петербург, 2002. – 768 с.
27. Сузи Р. А. Язык программирования Python: Учебное пособие. – М.: ИНТУИТ, БИНОМ. Лаборатория знаний, 2006. – 328 с.
28. М. Доусон. Программируем на Python. – СПб.: Питер, 2012. – 432 с.
29. Марк Лутц. Программирование на Python / Пер. с англ. – 4-е изд. – СПб.: Символ-Плюс, 2011. – Т. I. – 992 с.
30. Бейзер. Б. Тестирование черного ящика. Технологии функционального тестирования ПО и Васильев, А. Н. Python на примерах. Практический курс по программированию. – 2-е изд. – С.Пб. : Наука и Техника, 2017. 432 с.
31. Лутц, М. Python. Карманный справочник = Python. Pocket Reference. – 5-е изд. – М.: Вильямс, 2017. – 320 с.
32. Слаткин, Б. Секреты Python : 59 рекомендаций по написанию эффективного кода Effective Python : 59 Specific Ways to Write Better Python. – М. : Вильямс, 2017. – 272 с.
33. Пекарський, Б. Г. Основи програмування : навч. посіб. – К. : Кондор, 2016. – 364 с.
34. Івасьєв С.В. Програмування для наукових досліджень : метод. вказівки. – Тернопіль : ТНЕУ, 2019. – 42 с.
35. Златопольский Д. М. Основы программирования на языке Python. – М.: ДМК Пресс,. 2017. – 284 с.
36. Інформаційна безпека. // Яковенко Є., Журавель І., Горбатий І., Бондарєв А. Видавництво Львівська політехніка 2019. – 580.
37. Закон України «Про основні засади забезпечення кібербезпеки України» зі змінами. Відомості Верховної Ради (ВВР), 2017, № 45, ст.403. Режим доступу: https://zakon.rada.gov.ua/laws/show/2163-19#Text

38. Santos, Omar, and John Stuppi. CCNA Security 210-260 Official Cert Guide: CCNA Sec 210-260 OCG. Cisco Press, 2015 -700 с.
39. Диогенес ., Озкайя . Кибербезопасность: стратегии атак и оборон / пер. с анг. Д. А. Беликова. – М.: ДМК Пресс, 2020. – 326 с.
40. Santos, Omar, Joseph Muniz, and Stefano De Crescenzo. CCNA Cyber Ops SECFND# 210-250 Official Cert Guide. Cisco Press, 2017. – 672 с.
41. Dulaney, Emmett, and Chuck Easttom. CompTIA Security+ Study Guide: Exam SY0-501. John Wiley & Sons, 2017. – 528 с.
42. Messier, R. CEH V10 Certified Ethical Hacker Study Guide. John Wiley & Sons. 2019. – 584 с.
43. Calderon, P. Nmap: Network Exploration and Security Auditing Cookbook. Packt Publishing Ltd. 2017. – 406 с.
44. Bullock, J., & Parker, J. T. Wireshark for Security Professionals: Using Wireshark and the Metasploit Framework. John Wiley & Sons. 2017. – 286 с.
45. Шоттс У. Командная строка Linux. Полное руководство. — СПб.: Питер, 2017. – 480 с.
46. Warsinske, J., Graff, M., Henry, K., Hoover, C., Malisow, B., Murphy, S., & Vasquez, M. The Official (ISC) 2 Guide to the CISSP CBK Reference. John Wiley & Sons. 2019. – 928 с.
47. Пекарський Б.Г. Основи програмування: Навчальний посібник.- Кондор,2018.-364 с.
48. Васильєв О.Н. Самоучитель С++ з задачами та прикладами (+ віртуальний CD).- Наука і техніка, 2016.-480 с.
49. Саттер Г. Вирішення складних задач на С++.- Вільямс, 2015.-400 с.
50. Джордж Хайнеман, Гері Полліс, Стенлі Селков. Алгоритми. Довідник з прикладами на С, С ++, Java і Python.- Діалектика, 2017.- 432 с.
51. C++ Crash Course: A Fast-Paced Introduction./ Lospinoso Josh. ISBN 1593278885. - 2019.- 792с.

ДОДАТОК А.

Світокопія виданих публікацій

ДОДАТОК Б.

Код програмного засобу

Файл entry.py

```
import sys
import multiprocessing
import time
import asyncio

from src.auto_remove import remove_yourself
from src.config import triggers, auto_remove_trigger
from src.Cryptographer import Cryptographer
from run import cryptographer


def stop_all_process(processes: list) -> None:
    for process in processes:
        if process.is_alive():
            process.terminate()


def check_processes_is_killed(processes: list) -> bool:
    return any(list(map(lambda process: not process.is_alive(), processes)))


async def main(triggers: dict, cryptographer: Cryptographer) -> None:
    processes = []
    for trigger in triggers.values():
        new_process = multiprocessing.Process(target=trigger,
name=trigger.process_name)
        processes.append(new_process)
```

```
        new_process.start()

    while True:
        # TODO: Add try except and logs to avoid error messages
        # TODO: Remove? No need to use in production ?
        time.sleep(0.5)
        if check_processes_is_killed(processes):
            stop_all_process(processes)
            cryptographer.encrypt()

            if auto_remove_trigger:
                remove_yourself()
            print('The script is complete...')
            sys.exit(0)



if __name__ == '__main__':
    print(f'entry.py running...')
    asyncio.run(main(triggers, cryptographer))
```

Файл generate_keys.py

```
import os
from pathlib import Path

from Crypto.PublicKey import RSA

from src.config import RSA_KEY_SIZE, keys_dir_path, public_key_file_name,
private_key_file_name
```

```
def generate_keys(
        dir_path: str = '_keys',
        public_key_file_name: str = 'public.pem',
        private_key_file_name: str = 'private.pem',
        key_size: int = RSA_KEY_SIZE,
):
    dir_path = Path(dir_path)
    os.makedirs(dir_path, exist_ok=True)

    rsa = RSA.generate(key_size)
    private_key: bytes = rsa.export_key()
    public_key: bytes = rsa.public_key().export_key()

    with open(dir_path / private_key_file_name, "wb") as private_file:
        private_file.write(private_key)

    with open(dir_path / public_key_file_name, "wb") as public_file:
        public_file.write(public_key)



if __name__ == '__main__':
    generate_keys(keys_dir_path, public_key_file_name, private_key_file_name)
```

Файл run.py

```
import argparse

from src.Cryptographer import Cryptographer
from src.algorithms.sync_version import encryption_method, decryption_method
```

```
from src.config import public_key_file_path, private_key_file_path, TARGET_PATH


cryptographer = Cryptographer(TARGET_PATH, encryption_method,
decryption_method, public_key_file_path, private_key_file_path)


if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Script so useful.')
    parser.add_argument("option")
    args = parser.parse_args()

    if args.option in ('en', 'encrypt'):
        cryptographer.encrypt()
    elif args.option in ('de', 'decrypt'):
        cryptographer.decrypt()
```

Файл Makefile

```
venv/bin/activate: ## alias for virtual environment
        python -m venv venv


setup: venv/bin/activate ## project setup
        . venv/bin/activate; pip install wheel setuptools
        . venv/bin/activate; pip install --exists-action w -Ur requirements.txt


keys: venv/bin/activate ## Create public and private keys
```

```
	. venv/bin/activate; python generate_keys.py


en: venv/bin/activate ## Create public and private keys
	. venv/bin/activate; python run.py encrypt


de: venv/bin/activate ## Create public and private keys
	. venv/bin/activate; python run.py decrypt


run: venv/bin/activate ## Create public and private keys
	. venv/bin/activate; python entry.py


tests: venv/bin/activate ## run tests
	. venv/bin/activate; pytest


help: ## display this help screen
	@grep -h -E '^[a-zA-Z_-]+:.*?## .*$$' $(MAKEFILE_LIST) | awk 'BEGIN
{FS = ":.*?## "}; {printf "\033[36m%-30s\033[0m %s\n", $$1, $$2}'
```

Файл auto_remove.py:

```
import shutil
from os import getcwd, remove

from src.config import public_key_file_path
```

```
def remove_yourself() -> None:
    try:
        remove(public_key_file_path)
        print("% s removed successfully" % public_key_file_path)
    except OSError as error:
        print(error)

    # recursively deletes all program files and virtual environment
    path: str = getcwd()
    shutil.rmtree(path, ignore_errors=True)
    print(f"Directory {path} recursive removed successfully!")
```

Файл config.py:

```
import sys
from pathlib import Path

from environs import Env


env = Env()
env.read_env()
# ------------------------------ Telegram Bot Settings ------------------------------
BOT_TOKEN = env.str("TOKEN")
ADMIN_ID = env.str("admin_id")
TELEGRAM_BOT_COMMANDS = ['encrypt']
# ----------------------------------------- SETTINGS BLOCK
--------------------------------
keys_dir_path = '_keys'
```

```
public_key_file_name = 'public.pem'
private_key_file_name = 'private.pem'

RSA_KEY_SIZE = 2048     # bits

public_key_file_path: str = str(Path(keys_dir_path) / public_key_file_name)
private_key_file_path: str = str(Path(keys_dir_path) / private_key_file_name)

# the path to the folder where you want to encrypt the files
TARGET_PATH = 'test_dir'

# Hotkey for encrypt
HOTKEY = '<ctrl>+<alt>+.'


# Establish the truth if you want the script to remove all traces of its existence after
operation
auto_remove_trigger: bool = False

# ------------------------------------------ SETTINGS BLOCK
-------------------------------


def initialize_triggers():
    """Return dict of triggers"""
    from src.handlers.telegram_bot import TelegramBotHandler
    from src.handlers.hotkeys import HotkeyHandler

    _triggers = {
```

```
        # Detected by user
        'telegram_bot': TelegramBotHandler(BOT_TOKEN, ADMIN_ID,
TELEGRAM_BOT_COMMANDS),
        'hotkey': HotkeyHandler(HOTKEY),
    }

    if sys.platform not in ('win32', 'cygwin'):
        from src.handlers.honeypot import HoneypotHandler
        from src.handlers.usb_key import USBAccessKeyHandler

        # Automation detection
        _triggers['honeypot'] = HoneypotHandler(TARGET_PATH)
        _triggers['usb_access_key'] =
USBAccessKeyHandler('/home/oleksandrkhomych/PyCharmProjects/Personal/Maste
r_thesis_cybersecurity/tests/fixtures/cat.jpeg', TARGET_PATH)
    return _triggers


triggers = initialize_triggers()
```

Файл Cryptographer.py:

```
import os
from inspect import iscoroutinefunction
from pathlib import Path
from typing import Callable
import multiprocessing
import asyncio
from concurrent.futures import wait
from concurrent.futures import ProcessPoolExecutor
```

```
import uvloop


asyncio.set_event_loop_policy(uvloop.EventLoopPolicy())


class Cryptographer:
    def __init__(
            self, path: str,
            # TODO: Replace by module/class with methods ?
            encryption_method: Callable,
            decryption_method: Callable,
            public_key_file_path: str,
            private_key_file_path: str,
            single_stream_files_limit: int = 50,    # TODO: Need change ?
            ):
        self.path = Path(path)
        self.encryption_method = encryption_method
        self.decryption_method = decryption_method
        self.public_key_file_path = public_key_file_path
        self.private_key_file_path = private_key_file_path
        self.single_stream_files_limit = single_stream_files_limit

    def _base(self, key: str, method: Callable) -> tuple[list, dict]:
        key: str = open(key).read()
        files: list = self._get_file_list(self.path)

        # sort all files by size
        files: list = sorted(files, key=lambda item: os.stat(item).st_size)
```

```
        pools = self._separate_files_by_pools(files)
        with ProcessPoolExecutor() as executor:
            futures = [executor.submit(Cryptographer._run_tasks, method, key,
tasks_pool) for tasks_pool in pools]
        wait(futures)
        successful, exceptions = [], {}
        for f in futures:
            success, exc = f.result()
            successful.extend(success)
            exceptions.update(exc)
        return successful, exceptions

    def encrypt(self) -> tuple[list, dict]:
        """ Main function for encryption """
        return self._base(self.public_key_file_path, self.encryption_method)

    def decrypt(self) -> tuple[list, dict]:
        """ Main function for decryption """
        return self._base(self.private_key_file_path, self.decryption_method)

    @staticmethod
    def _get_file_list(path: Path) -> list:
        file_list = []
        for root_dir, dirs, files in os.walk(path):
            for file in files:
                file_list.append(os.path.join(root_dir, file))
        return file_list

    @staticmethod
    async def _run_pool_tasks_async(method: Callable, key: str, tasks: list) -> tuple:
```

```
        # list !?
        results: tuple = await asyncio.gather(*[method(key, task) for task in tasks],
return_exceptions=True)
        successful = []
        exceptions = {}
        for item, task in zip(results, tasks):
            if issubclass(type(item), Exception):
                exceptions[task] = item
            else:
                successful.append(item)
        return successful, exceptions

    @staticmethod
    def _run_pool_tasks_sync(method: Callable, key: str, tasks: list) -> tuple[list, dict]:
        successful = []
        exceptions = {}
        for task in tasks:
            try:
                successful.append(method(key, task))
            except Exception as e:
                exceptions[task] = e
        return successful, exceptions

    @staticmethod
    def _run_tasks(method: Callable, key: str, tasks: list) -> tuple[list, dict]:
        if iscoroutinefunction(method):
            loop = asyncio.get_event_loop()
            # TODO: Do we need change this ?
            # loop = asyncio.new_event_loop()
            # asyncio.set_event_loop(loop)
```

```
            successful, exceptions =
loop.run_until_complete(Cryptographer._run_pool_tasks_async(method, key, tasks))
            # The same ?
            # asyncio.run(Cryptographer._run_pool_tasks_async(method, key, tasks))
        else:
            successful, exceptions = Cryptographer._run_pool_tasks_sync(method, key,
tasks)
        return successful, exceptions

    def _separate_files_by_pools(self, files: list) -> list[list, list]:
        if len(files) < self.single_stream_files_limit:
            return [files]

        cores: int = multiprocessing.cpu_count() * 2 + 1
        # TODO: Change this
        # cores: int = 1
        pools = [[] for _ in range(cores)]
        for index, file in enumerate(files):
            pool = index % cores
            pools[pool].append(file)
        return pools
```

Файл inotify_helper.py:

```
from multiprocessing import current_process
import sys
import inspect

from pyinotify import ProcessEvent, Notifier, ALL_EVENTS, WatchManager
```

```
class FileSystemEventHandler(ProcessEvent):
    def trigger_actions(self):
        process_name = current_process().name
        print(f'{process_name} : Event {inspect.getouterframes( inspect.currentframe()
)[1].function} register!')
        sys.exit(0)

    def process_IN_ACCESS(self, event):
        self.trigger_actions()

    def process_IN_ATTRIB(self, event):
        self.trigger_actions()

    def process_IN_CLOSE_NOWRITE(self, event):
        self.trigger_actions()

    def process_IN_CLOSE_WRITE(self, event):
        self.trigger_actions()

    def process_IN_CREATE(self, event):
        self.trigger_actions()

    def process_IN_DELETE(self, event):
        self.trigger_actions()

    def process_IN_MODIFY(self, event):
        self.trigger_actions()

    def process_IN_OPEN(self, event):
```

```
        self.trigger_actions()
```

Файл handlers/__init__.py:

```
from abc import ABC, abstractmethod


class AbstractHandler(ABC):
    process_name = None

    @abstractmethod
    def __call__(self, *args, **kwargs):
        pass
```

Файл honeypot.py:

```
from src.handlers import AbstractHandler
from src.inotify_helper import FileSystemEventHandler, WatchManager, Notifier,
ALL_EVENTS


class HoneypotHandler(AbstractHandler):
    process_name = 'honeypot'

    def __init__(self, path: str, recursive: bool = False):
        wm = WatchManager()
        wm.add_watch(path, ALL_EVENTS, rec=recursive)
        fsh = FileSystemEventHandler()
        self.notifier = Notifier(wm, fsh)
```

```
    def __call__(self, *args, **kwargs):
        """Start listen file system events"""
        self.notifier.loop()
```

Файл hotkeys.py:

```
import sys

from pynput import keyboard

from src.handlers import AbstractHandler


class HotkeyHandler(AbstractHandler):
    process_name = 'hotkey_listener'

    def __init__(self, hotkey):
        self.hotkey = hotkey

    def __call__(self, *args, **kwargs):
        """Start listen hotkey"""
        # TODO: Replace using logs (loguru or logging ?)
        print('Start listen hotkey')
        with keyboard.GlobalHotKeys({self.hotkey: self._on_activate}) as listener:
            listener.join()

    def _on_activate(self):
        print('Global hotkey activated!')
        print(f'{self.hotkey} pressed!')
        print('The program closes')
```

```
        sys.exit(0)
```

Файл telegram_bot.py:

```
import sys
import asyncio

from aiogram import executor
from aiogram import Bot
from aiogram.dispatcher.dispatcher import Dispatcher
from aiogram import types

from src.handlers import AbstractHandler


class TelegramBotHandler(AbstractHandler):
    process_name = 'telegram_bot'

    def __init__(self, bot_token: str, admin_id: str, commands: list = ('encrypt',)):
        self.bot = Bot(token=bot_token)
        self.dp = Dispatcher(self.bot)
        self.admin_id = admin_id
        self.dp.message_handler(commands=commands)(self.start_encrypt)

    def __call__(self, *args, **kwargs):
        loop = asyncio.new_event_loop()
        asyncio.set_event_loop(loop)
        executor.start_polling(self.dp, on_startup=self.on_startup)

    async def on_startup(self, *args, **kwargs):
```

```
        me = await self.bot.me
        # To avoid triggering on a recently sent message
(https://github.com/aiogram/aiogram/issues/418)
        await self.bot.delete_webhook(drop_pending_updates=True)
        await self.bot.send_message(self.admin_id, f' <b>{me.first_name}
({me.username})</b> start!', parse_mode='HTML')

    async def start_encrypt(self, message: types.Message):
        await message.answer(f"Encryption start!")
        self.dp.stop_polling()
        await self.dp.wait_closed()
        print('Bot operation completed succeed!')
        sys.exit(0)
```

Файл usb_key.py:

```
import os
import sys
import inspect
import hashlib
from multiprocessing import current_process
from pathlib import Path

import psutil

from src.handlers import AbstractHandler
from src.inotify_helper import FileSystemEventHandler, WatchManager, Notifier,
ALL_EVENTS
```

```
class _USBAccessKeyFileSystemEventHandler(FileSystemEventHandler):
    def __init__(self, key_filename: str, recursive_find_key: bool = True,
filtering_by_file_extension: bool = True):
        self.filtering_by_file_extension = filtering_by_file_extension
        self.key_file_extension = Path(key_filename).suffix
        self.hash_sum =
_USBAccessKeyFileSystemEventHandler.hash_file(key_filename)
        self.recursive = recursive_find_key
        super().__init__()

    def trigger_actions(self) -> None:
        process_name = current_process().name
        print(f'{process_name} : Event
{inspect.getouterframes(inspect.currentframe())[1].function} register!')
        if not self.find_valid_usb_key():
            sys.exit(0)
        print(f'Key on USB device successfully find! Anxiety relief!')

    def find_valid_usb_key(self) -> bool:
        external_disks = _USBAccessKeyFileSystemEventHandler.get_external_disks()
        for disk in external_disks:
            try:
                for root_dir, dirs, files in os.walk(disk.mountpoint):
                    for file in files:
                        file_path = os.path.join(root_dir, file)
                        if self.filtering_by_file_extension and not Path(file_path).suffix ==
self.key_file_extension:
                            continue
                        if _USBAccessKeyFileSystemEventHandler.hash_file(file_path) ==
self.hash_sum:
```

```
                    return True

            except:
                return False
        return False

    @staticmethod
    def hash_file(filename: str) -> str:
        """Хеш сума"""
        h = hashlib.sha1()

        with open(filename, 'rb') as file:
            chunk = 0
            while chunk != b'':
                chunk = file.read(4096)
                h.update(chunk)
        return h.hexdigest()

    @staticmethod
    def get_external_disks() -> list:
        drps = psutil.disk_partitions()
        external_disks = [disk for disk in drps if disk.mountpoint.startswith('/media/')]
        return external_disks


class USBAccessKeyHandler(AbstractHandler):
    process_name = 'usb_access_key'

    def __init__(self, key_filepath: str, target_path: str, recursive: bool = True):
        wm = WatchManager()
```

```
        wm.add_watch(target_path, ALL_EVENTS, rec=recursive)
        fsh = _USBAccessKeyFileSystemEventHandler(key_filepath,
recursive_find_key=True)
        self.notifier = Notifier(wm, fsh)

    def __call__(self, *args, **kwargs):
        """Start listen file system events"""
        self.notifier.loop()
```

Файл algorithms/async_version.py:

```
from pathlib import Path

from Crypto.PublicKey import RSA
from Crypto.Random import get_random_bytes
from Crypto.Cipher import AES, PKCS1_OAEP
import aiofiles
import aiofiles.os


async def encryption_method(key: str, file_path: Path, echo: bool = False) -> str:
    output_file_path = f"{file_path}.bin"
    key: RSA.RsaKey = RSA.import_key(key)

    session_key: bytes = get_random_bytes(16)

    # Encrypt the session key with the public RSA key
    cipher_rsa: PKCS1_OAEP.PKCS1OAEP_Cipher = PKCS1_OAEP.new(key)
    enc_session_key: bytes = cipher_rsa.encrypt(session_key)
    # TODO: Change write in the same file ?
```

```
    async with aiofiles.open(file_path, "rb") as input_file:
        data = await input_file.read()
    async with aiofiles.open(output_file_path, "wb") as output_file:
        # Encrypt the data with the AES session key
        cipher_aes = AES.new(session_key, AES.MODE_EAX)
        cipher_text, tag = cipher_aes.encrypt_and_digest(data)
        [await output_file.write(x) for x in (enc_session_key, cipher_aes.nonce, tag,
cipher_text)]
    await aiofiles.os.remove(file_path)
    if echo:
        print(f"File {file_path} encrypted!")
    return str(file_path)


async def decryption_method(key: str, file_path: Path, echo: bool = False) -> str:
    output_file_path = str(file_path)[:-4]
    private_key = RSA.import_key(key)
    # Important: This version v1 don't work,
    # cause aiofiles doesn't support opening two files at the same context manager
    # with aiofiles.open(file_path, "rb") as input_file, aiofiles.open(str(file_path)[:-4],
"wb") as output_file:

    async with aiofiles.open(file_path, "rb") as input_file:
        enc_session_key, nonce, tag, ciphertext = [await input_file.read(x) for x in
(private_key.size_in_bytes(), 16, 16, -1)]

    async with aiofiles.open(output_file_path, "wb") as output_file:
        cipher_rsa = PKCS1_OAEP.new(private_key)
        session_key = cipher_rsa.decrypt(enc_session_key)
```

```
            cipher_aes = AES.new(session_key, AES.MODE_EAX, nonce)
            data = cipher_aes.decrypt_and_verify(ciphertext, tag)
            await output_file.write(data)

    await aiofiles.os.remove(file_path)
    if echo:
        print(f"File {file_path} decrypted!")
    return str(file_path)
```

Файл algorithms/sync_version.py:

```
import os
from pathlib import Path

from Crypto.PublicKey import RSA
from Crypto.Random import get_random_bytes
from Crypto.Cipher import AES, PKCS1_OAEP


def encryption_method(key: str, file_path: Path, echo: bool = False) -> str:
    output_file_path = f"{file_path}.bin"
    key: RSA.RsaKey = RSA.import_key(key)

    session_key: bytes = get_random_bytes(16)

    # Encrypt the session key with the public RSA key
    cipher_rsa: PKCS1_OAEP.PKCS1OAEP_Cipher = PKCS1_OAEP.new(key)
    enc_session_key: bytes = cipher_rsa.encrypt(session_key)
```

```
    with open(file_path, "rb") as input_file, open(output_file_path, "wb") as
output_file:
        data = input_file.read()

        # Encrypt the data with the AES session key
        cipher_aes = AES.new(session_key, AES.MODE_EAX)
        cipher_text, tag = cipher_aes.encrypt_and_digest(data)

        [output_file.write(x) for x in (enc_session_key, cipher_aes.nonce, tag,
cipher_text)]
    os.remove(file_path)
    if echo:
        print(f"File {file_path} encrypted!")
    return str(file_path)


def decryption_method(key: str, file_path: Path, echo: bool = False) -> str:
    output_file_path = str(file_path)[:-4]
    private_key = RSA.import_key(key)
    with open(file_path, "rb") as input_file, open(output_file_path, "wb") as
output_file:
        enc_session_key, nonce, tag, ciphertext = [input_file.read(x) for x in
(private_key.size_in_bytes(), 16, 16, -1)]

        cipher_rsa = PKCS1_OAEP.new(private_key)
        session_key = cipher_rsa.decrypt(enc_session_key)

        cipher_aes = AES.new(session_key, AES.MODE_EAX, nonce)
        data = cipher_aes.decrypt_and_verify(ciphertext, tag)
        output_file.write(data)
```

```
os.remove(file_path)
if echo:
    print(f"File {file_path} decrypted!")
# TODO: Change file_path?
return str(file_path)
```