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

Янік Іван Іванович

Алгоритми шифрування на мобільних пристроях з використанням криптографічних методів / Encryption Algorithms on Mobile Devices Using Cryptographic Methods

спеціальність: 125 – Кібербезпека

освітньо-професійна програма –Кібербезпека

Кваліфікаційна робота

Виконав студент групи КБм-22
І.І. Янік

_____

Науковий керівник
к.т.н., доцент Т.Г. Цаволик

_____

Кваліфікаційну роботу допущено
до захисту:

«____» _____ ___2023 р.

Завідувач кафедри
_____ В.В.Яцків

ТЕРНОПІЛЬ – 2023

РЕФЕРАТ

Кваліфікаційна робота на тему «Алгоритми шифрування на мобільних пристроях з використанням криптографічних методів» на здобуття освітнього ступеня «Магістр» зі спеціальності 125 «Кібербезпека» освітньо-професійної програми «Кібербезпека» написана обсягом 87 сторінок і містить 31 рисунків, 12 таблиць та 49 джерел за переліком посилань.

Метою кваліфікаційної роботи є розробка криптографічної системи, яка забезпечує шифрування і дешифрування потоку інформації для обміну між портативними засобами комунікації через широкий спектр засобів обміну.

Методи досліджень базуються на положеннях математичного апарат теорії чисел, теорії ймовірностей, теорії алгоритмів та теорії паралельних обчислень.

Розроблено криптографічну систему, яка забезпечує кращу безпеку шифрування, передачу та дешифрування інформації в порівняні з існуючими системати. Архітектурна реалізація структури пакетів у вигляді окремих компонентів в стилі об’єктно-орієнтованої та багаторівневої архітектури забезпечує можливість подальшого розширення продукту або заміни окремих компонентів у випадку розвитку нових алгоримів та методів ширування.

Реалізовано мобільний криптографічний пакет КСБМП для організації безпеки для мобільних пристроїв, який можна використовувати для безпечного застосування криптографії в середовищі мобільних додатків, зберігаючи при цьому простий API та ефективне використання криптографії. Можливими напрямками подальших досліджень є продовження додаткових криптографічних методів та розробка нових протоколів обміну інформацією.

Ключові слова: КРИПТОГРАФІЯ З СЕМЕТРИЧНИМ КЛЮЧЕМ, КРИПТОГРАФІЯ З АСИМЕТРИЧНИМ КЛЮЧЕМ, АЛГОРИТМ AES, ХЕШ-ФУНКЦІЯ, КОД АВТЕНТИФІКАЦІЇ ПОВІДОМЛЕННЯ, СЕРТИФІКАТ ВІДКРИТОГО КЛЮЧА, ЦИФРОВИЙ ПІДПИС.

REZUME

Qualification work on "Encryption algorithms on mobile devices using cryptographic methods" for the degree of "Master" in the specialty 125 "Cybersecurity" educational and professional program "Cybersecurity" is written in 87 pages and 31 figures, 12 tables and 49 sources for references.

The goal of the qualification work is the development of a cryptographic system that provides encryption and decryption of the flow of information for exchange between portable means of communication through a wide range of means of exchange.

Research methods are based on the provisions of the mathematical apparatus of number theory, probability theory, algorithm theory, and parallel computing theory.

A cryptographic system has been developed that provides better security of encryption, transmission and decryption of information compared to existing systems. The architectural implementation of the package structure in the form of individual components in the style of object-oriented and multi-level architecture provides the possibility of further product expansion or replacement of individual components in the event of the development of new algorithms and methods of spreading.

Implemented KSBMP mobile cryptographic package for mobile security organization, which can be used to securely apply cryptography in mobile application environments while maintaining a simple API and efficient use of cryptography. Possible areas of further research are the continuation of additional cryptographic methods and the development of new information exchange protocols.

Keywords: SYMMETRIC KEY CRYPTOGRAPHY, ASYMMETRICAL KEY CRYPTOGRAPHY, AES ALGORITHM, HASH FUNCTION, MESSAGE AUTHENTICATION CODE, PUBLIC KEY CERTIFICATE, DIGITAL SIGNATURE

ЗМІСТ

ВСТУП

Актуальність. Найважливішим автоматизованим інструментом безпеки зв'язку є шифрування [39]. Існує багато різних алгоритмів шифрування, але більшість з них можна згрупувати за однією з двох схем: симетричного та асиметричного ключа. Однак шифрування не може запобігти перехопленню та модифікації даних на каналах зв’язку. Шифрування може захистити канал зв’язку лише від перехоплювачів, які витягують конфіденційні дані. Для активних атак, таких як модифікація повідомлення, код автентифікації повідомлення (MAC) використовується для перевірки цілісності повідомлення. Для того, щоб сектор мобільного зв’язку міг процвітати в епоху інформаційно-комунікаційних технологій, безпека є важливим аспектом.

Традиційні криптографічні пакети для настільних комп’ютерів приховують безліч специфічних для алгоритму функцій під єдиним набором методів API із часто складними критеріями вибору алгоритму, у деяких випадках вимагаючи встановлення до десятка параметрів для вибору режиму роботи [11]. У результаті розробники тісно зв’язують програму з базовою реалізацією криптографії, порушуючи таким чином принцип поділу проблем. Крім того, виникнуть складні проблеми, пов’язані з тим, як криптографію застосовувати в програмі, і ймовірність неправильного використання методу API розробником збільшуються. На жаль, мобільні пристрої обмежені такими ресурсами, як швидкість ЦП, обсяг пам’яті, тому багато криптографічних пакетів, створених для мобільних пристроїв, іноді є підмножиною оригінальних криптографічних пакетів для настільних ПК. Пакети мобільної криптографії мають легкі методи API, щоб зберегти малий розмір [2, 12, 30, 43], тому мобільні набори інструментів не містять багатьох корисних методів для підтримки використання та ініціалізації криптографічних алгоритмів.

Надійні криптографічні алгоритми не завжди підтримуються в мобільному криптографічному пакеті. Наприклад, Windows CAPI для Pocket PC і Windows CE не мають AES, а .NET Compact Framework 2.0 не підтримує SHA-256 [25, 26]. Це буде обмеженням для програм, які потребують таких

алгоритмів. Відсутність підтримки алгоритмів також може стати причиною недоступності певних криптографічних функцій. Як наслідок, розробнику доводиться перебирати API та імітувати недоступні функції за допомогою інших методів.

Складність методів API пов'язана з тим, як криптографічні алгоритми реалізовані в пакетах. Багато сучасних пакетів мобільної криптографії реалізовано за допомогою алгоритмів, які мають кілька варіантів ініціалізації, якими розробники можуть зловживати. Наприклад, вектор ініціалізації за замовчуванням може бути ініціалізований порожнім масивом байтів, який може бути помилково повторно використаний [9, 41]. Іншим прикладом є генерація секретного ключа за допомогою генераторів псевдовипадкових чисел, які підтримуються в пакеті. Якщо припустити, що зовнішній API здатний охопити будь-які хитросплетіння застосування криптографії від розробника, якщо будь-який із внутрішніх модулів та/або компонентів було неправильно визначено, взаємодіяно чи реалізовано, уся система безпеки, яка використовує цю бібліотеку, вийшла б з ладу. Просте розгортання сильного алгоритму, такого як AES, не обов'язково гарантує безпеку програмного продукту [20]. Суть полягає в тому, що впровадження та застосування криптографії є складним процесом, до якого не слід підходити випадково. Існують основи криптографії, які не можна випадково нехтувати. Під цими основами ми маємо на увазі найкращі практики застосування криптографії (незалежно від середовища).

Таким чином, мобільні криптографічні пакети є більш складними у використанні порівняно з їхніми настільними аналогами, оскільки вони вразливі всередині пакета, де надійні криптографічні основи не часто застосовуються, і зовні в API, де обмежені методи ускладнюють їх використання. Отже, розробка криптографічної системи безпеки із використанням сучасних криптографічних засобів на даний час є надзвичайно актуальною задачею.

Мета кваліфікаційної роботи полягає у розробці криптографічної системи, яка забезпечує шифрування і дешифрування потоку інформації для

обміну між портативними засобами комунікації через широкий спектр засобів обміну.

Для досягнення поставленої мети було вирішено такі завдання.

1. Дослідження сучасні методи захисту інформації та встановити їхні переваги та недоліки.

2. Теоретичне обґрунтування вибору криптологічних алгоритмів ширфування для створювальної криптографічної системи.

3. Експериментальне дослідження процесів шифрування та дешифрування криптографічними алгоритмами розроблювальної системи.

4. Розробка та експериментальна програмна апробація криптографічної системи для портативних засобів комунікації.

Об'єктом дослідження кваліфікаційної роботи є процеси програмного шифрування і дешифрування інформаційних потоків даних в критографічній системі при застосуванні в мобільних та комп’ютерних системах.

Предметом дослідження є методи та алгоритми криптографії з симетиричним та асиметричним ключем. Засоби автентифікації повідомлень такі, як хеш-функції, код автентифікації повідомлення, сертифікати відкритого ключа та цифрові підписи.

Методи досліджень. Для вирішення поставлених наукових завдань використовувався математичний апарат теорії чисел, теорії ймовірностей, теорії алгоритмів та теорії паралельних обчислень.

Достовірність та обґрунтованість отриманих у дипломній роботі результатів та сформульованих на їх основі висновків забезпечуються строгістю та коректним використанням методологічного апарату досліджень. Справедливість висновків щодо ефективності та коректності запропонованих методів та алгоритмів підтверджена комп'ютерним моделюванням та експериментами.

Наукова новизна. Розроблено криптографічну систему, яка забезпечує кращу безпеку шифрування, передачу та дешифрування інформації в порівняні з існуючими системати. Архітектурна реалізація структури пакетів у вигляді окремих компонентів в стилі об’єктно-орієнтованої та багаторівневої

архітектури забезпечує можливість подальшого розширення продукту або заміни окремих компонентів у випадку розвитку нових алгоримів та методів ширування.

Практичне значення отриманих результатів.

Реалізовано мобільний криптографічний пакет КСБМП для організації безпеки для мобільних пристроїв, який можна використовувати для безпечного застосування криптографії в середовищі мобільних додатків, зберігаючи при цьому простий API та ефективне використання криптографії.

КСБМП схожий за концепцією на cryptlib [11], який зосереджується на безпеці внутрішнього об’єкта шляхом визначення списків контролю доступу. Такі методи можуть виходити за рамки програмного забезпечення, яке підтримується на мобільному пристрої. У нашій роботі безпека зосереджена на застосуванні надійних криптографічних основ всередині інфраструктури та зовні в API, таким чином надаючи перевагу підходу до безпеки зсередини.

КСБМП реалізовано на Java та виконується на платформі Java 2 Micro Edition із профілем мобільного інформаційного пристрою (J2ME/MIDP) [42].

Публікації та апробація ВКР.

1. Матеріали науково-практичного симпозіуму «Захист інформації», Тернопіль, 2023 — с. 188-192

2. Збірник матеріалів науково-практичної конференції молодих вчених, аспірантів та студентів «Кібербезпека та комп’ютерно-інтегровані технології» (КБКІТ – 2023), Тернопіль, 2023 — с. 117-120

1. Аналіз предметної області

1.1. Криптографія. Криптографічні системи. Методика оцінки дизайну архітектурних систем.

Криптографія, як інструмент передачі секретної інформації застосовувалась з давніх часів та сягає віків стародавнього Єгипту 4000 років тому [48]. Криптографія використовувалася під час обох світових війн і аж до 1960-х років, коли вона в основному використовувалася урядами та військовими для захисту національних секретів і стратегій. Бурхливий розвиток та застосування криптографії стався із розквітом Інтернету на початку 1990-х років, коли вона почала відігравати важливу роль у захисті Інтернету.

З розквітом мобільного Інтернету виникла необхідність в кращих механізмах безпеки. Трьома найважливішими аспектами мобільної безпеки є конфіденційність даних, контроль доступу та безпека даних на пристрої [49]. Конфіденційність даних означає захист повідомлень від підслуховування. Щоб запобігти несанкціонованим спробам отримати доступ до конфіденційних даних, що зберігаються на пристрої (наприклад, фінансових даних, паролів, особистих ключів тощо), має бути шифрування, щоб забезпечити захист цих даних. Протоколи безпеки на основі підключення, такі як HTTPS, WTLS і TLS, недостатні для покриття цих трьох аспектів мобільної безпеки. Ідея безпеки на основі з'єднання полягає в тому, щоб захистити все, що проходить через канал зв'язку. Цей підхід має кілька обмежень [18, 49]:

- Необхідно встановити пряме з'єднання між клієнтом і сервером: якщо програма має взаємодіяти із декількома посередниками, які надають численні додаткові послуги, кілька з'єднань HTTPS об'єднуються разом. Це може призвести до потенційної вразливості безпеки на з'єднувальних вузлах. Керування сертифікатами відкритих ключів для кожного з'єднання може бути виснажливим.
- Увесь вміст зашифровано: захист на основі з'єднання не потрібний у деяких прикладних сценаріях, таких як трансляція котирувань акцій або отримання багаторівневого схвалення транзакції, оскільки частини зв'язку мають бути відкритими. Натомість перевірка автентичності цих даних і

підписів важливіша. Шифрування всього вмісту без потреби, оскільки створює додаткові витрати на обробку.

- HTTPS є негнучким для програм, які мають особливі вимоги до безпеки та продуктивності: HTTPS не підтримує користувацькі механізми рукостискання або обміну ключами. Наприклад, він не вимагає від клієнтів автентифікації. Сильніша безпека, яка потребує 256-бітного симетричного шифрування з ключем, може бути не гарантованою.

Щоб охопити три аспекти мобільної безпеки, мобільним розробникам потрібен програмний доступ до криптографічних алгоритмів. Ці алгоритми загорнуті в набір програмних інтерфейсів прикладних програм (API), які надають розробнику доступ до їх функціональних можливостей. Якщо операційна система або платформа мобільного пристрою не підтримує криптографічні пакети, можна використовувати сторонній набір криптографічних інструментів. Приклади пакетів мобільної криптографії, які підтримуються на певній платформі/операційній системі, включають: Security and Trust Service API [43], Microsoft CryptoAPI для Windows CE [25], Bouncy Castle API [2].

1.1.1. Операції та символи

Існує ряд операцій і символів, які будуть використовуватися в цій роботі. Для операцій шифрування та дешифрування використовуються такі позначення:

- $\mathbf{CIPH_K(P)}$ – позначає операцію шифрування відкритого тексту *P* за допомогою секретного ключа *K*.
- $\mathbf{CIPH^{-1}_K(C)}$ – позначає операцію дешифрування зашифрованого тексту *C* за допомогою секретного ключа *K*.
- $\mathbf{CIPH_{PUB\_A}(P)}$ – позначає операцію шифрування повідомлення *M* з використанням відкритого ключа, що належить об’єкту *A*.
- $\mathbf{CIPH^{-1}_{PRI\_A}(C)}$ – позначає операцію дешифрування зашифрованого тексту *C* за допомогою закритого ключа, що належить об’єкту *A*.

Найбільш часто використовувані символи:

- **IV** – Вектор ініціалізації.
- $\mathbf{O_i}$ – і-й блок виведення.

- **NC** – одноразовий.
- **T** – лічильник.
- $\oplus$ – операція XOR.
- || – операція конкатенація. Наприклад, якщо X || Y, то результат XY.

1.1.2. Криптографія з симетричним ключем

Шифрування з симетричним ключем було єдиним типом схеми шифрування аж до кінця 1970-х років [39]. Процес шифрування та дешифрування в шифрі з симетричним ключем залежить лише від секретного ключа, який спільно використовують відправник і одержувач. Симетричні ключові шифри можна розділити на дві категорії, а саме потокові та блочні шифри. Потокові шифри працюють з відкритим текстом, 1 біт за раз. Блочні шифри працюють із групою бітів фіксованого розміру у відкритому тексті та є найбільш поширеним типом симетричного шифру.

Ми розглянемо блочний шифр Advanced Encryption Standard (AES), оскільки це останній стандарт шифрування, встановлений урядом США для шифрування та дешифрування несекретної конфіденційної інформації. Також розглянемо питання безпеки для використання криптографії з симетричним ключем. NIST визначає, що AES має бути симетричним блоковим шифром із розміром блоку 128 біт (16 байт) і підтримкою довжини ключа 128, 192 та 256 біт. AES не використовує конструкцію Feistel, але все ще використовує математичні операції, такі як підстановки, перестановки, XOR, і має кілька раундів [7]. Подальші раунди аналогічні, а кількість раундів (від 10 до 14 раундів) залежить від розміру ключа. AES працює на байтовому рівні, що дозволяє ефективно впроваджувати як апаратне, так і програмне середовище.

Для однієї циклічної операції AES першою операцією є XOR вхідного відкритого тексту 16 байтів за допомогою раундового ключа. Наступним процесом є байтовий підетап, де кожен із 16 байтів потім використовується як індекс у таблиці S-box, яка відображає 8-бітні входи на 8-бітні виходи. Усі S-блоки ідентичні. Третій і четвертий процеси — це рядок зсуву та стовпець змішування, де байти потім переставляються в 4 групи по 4 байти кожна для змішування в межах лінійної функції змішування (рисунок 1.1). Термін

лінійний означає, що кожен вихідний біт функції змішування є XOR кількох вхідних бітів [7].

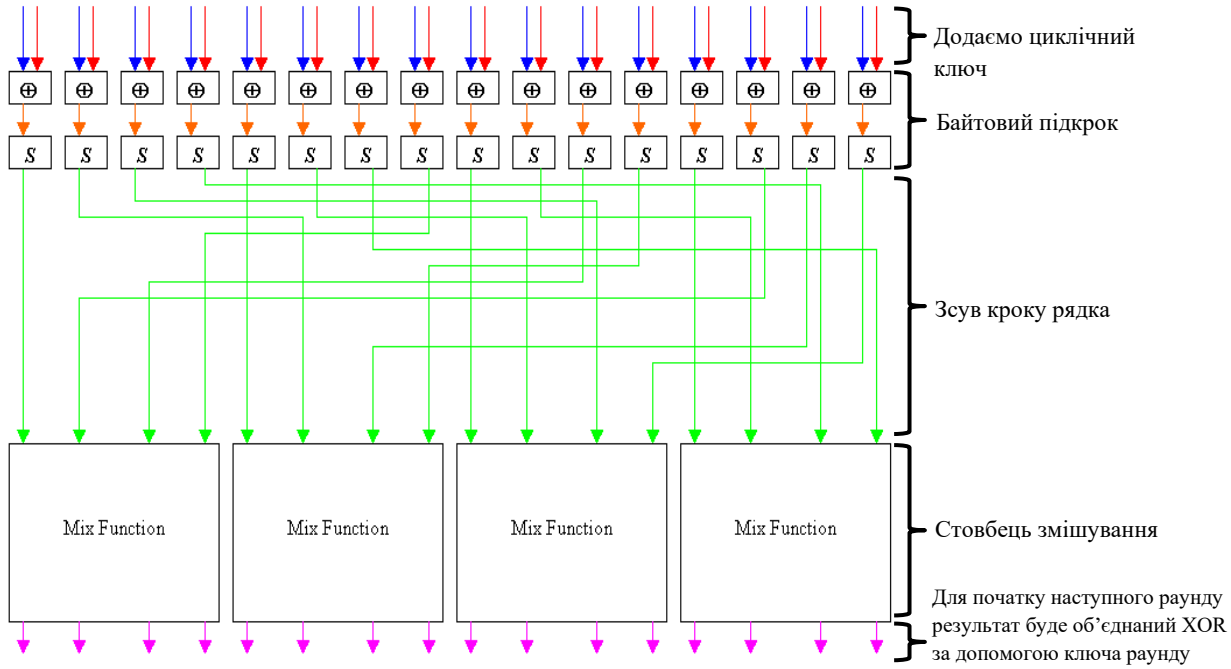


Рисунок 1.1 — Структура одного циклу AES

Зараз ми розглянемо деякі аспекти безпеки під час використання криптографії з симетричним ключем. Ці міркування містять важливі висновки, які будуть критеріями для безпечного впровадження розроблювальної нами криптографічної системи безпеки для мобільних пристроїв (КСБМП).

З нашого дослідження літератури було встановлено, що криптоаналітики використовують різні методи атак, щоб спробувати знайти вразливі місця в блокових шифрах. Проблеми колізії є більш помітними для 64-бітних блокових шифрів порівняно з 128-бітними блоковими шифрами.

Ефективне керування ключами охоплює створення, поширення і зберігання ключів. Хороший симетричний ключ повинен містити випадковий рядок бітів, який не викликає жодних математичних атак [38]. Щоб генерувати випадкові байти, слід використовувати криптографічно стійкий генератор псевдовипадкових чисел (PRNG). PRNG може бути корисним для генерації симетричних ключів [3]. PRNG містить детермінований алгоритм, який приймає випадкове початкове число для генерації виходу, який неможливо легко визначити за допомогою будь-якого математичного аналізу. Це досягається шляхом застосування криптографічних методів, таких як хеш-

функції та/або блочні шифри з відповідним режимом, який змішує ці біти. PRNG дає той самий результат, якщо зерно, яке використовується для отримання цього результату, повторно використовується. Вихідні дані завжди повинні бути недетермінованими, якщо насіння є передбачуваним, атака на отримання внутрішнього стану PRNG стає більш можливою. Після отримання внутрішнього стану зловмисник може визначити вихід PRNG.

Насіння можна зібрати з різних джерел подій [14]. Найкращим джерелом вхідних даних для змішування буде апаратна випадковість, наприклад, аудіовхід із тепловим шумом або радіоактивний розпад. Якщо апаратні випадкові джерела недоступні, можна використовувати програмні випадкові джерела. До них належать системні годинники, системні буфери або буфери вводу-виводу, мережеві серійні номери та/або адреси та дані користувача.

Якщо генератор не збирає достатньо ентропії для вихідного коду, випадковість виходу не може бути гарантована. Однак оцінити кількість зібраної ентропії важко, якщо не неможливо, зробити це правильно, тому PRNG під назвою Fortuna [7] вирішує цю проблему, збираючи випадкові джерела з різних подій у 32 групи рядків необмеженої довжини. Кожне джерело циклічно розподіляє свої випадкові події по пулах, щоб забезпечити більш-менш рівномірний розподіл ентропії з кожного джерела по пулах і гарантувати, що зловмисник не зможе визначити об’єднані дані. Ефективну стратегію розподілу ключів можна зробити за допомогою центру розподілу ключів [39] і асиметричної криптографії ключів. Однак з останнім слід звернути увагу на те, як буде отримано протокол.

Не рекомендується постійно зберігати симетричні ключі на мобільних пристроях, де існує більша ймовірність їх викрадення. Залежно від ситуації, якщо ключ використовується для зв’язку між двома сторонами через мережу, рекомендується використовувати ключ сеансу, оскільки він обмежує шанс зловмисника отримати правильний ключ.

Поширеним способом атаки на симетричний ключ є застосування грубої сили для всіх можливих комбінацій, які призведуть до отримання ключа. У таблиці 2.1 представлено середній час, необхідний для повного пошуку ключів. Одиниці вимірюються в мікросекундах (мкс). Результати показані з припущенням, що для виконання одного шифрування на процесорі з

використанням 1 ключа потрібно 1 мкс. Використовуючи паралельну організацію мікропроцесорів, можна дешифрувати з більшими величинами.

Таблиця 2.1 — Середній час, необхідний для вичерпного пошуку ключів

Розмір ключа, біт	**К-сть альтернативних ключів**	**Час, необхідний для 1 дешифрування/мкс**	**Необхідний час при 10^6 дешифрування/мкс**
32	$2^{32} = 4{,}3 * 10^9$	35,8 хвилин	2,15 мілісекунди
56	$2^{56} = 7{,}2 * 10^{16}$	1142 роки	10 годин
128	$2^{128} = 3{,}4 * 10^{38}$	$5{,}4 * 10^{24}$ років	$5{,}4 * 10^{18}$ років
168	$2^{168} = 3{,}7 * 10^{50}$	$5{,}9 * 10^{36}$ років	$5{,}9 * 10^{30}$ років
192	$2^{192} = 6{,}3 * 10^{57}$	$9{,}95 * 10^{43}$ років	$9{,}95 * 10^{37}$ років
256	$2^{256} = 1{,}2 * 10^{77}$	$1{,}84 * 10^{63}$ років	$1{,}84 * 10^{57}$ років

Сьогодні існує багато програм, які використовують 128-бітні ключі. Однак 128-бітний ключ може призвести до колізійних атак, таких як зустріч посередині та день народження. Відповідно до [7] більшість блочних шифрів допускають атаки типу зустрічі в тій чи іншій формі. Щоб досягти 128-бітного дизайну безпеки, потрібен 256-бітний ключ. Іншими словами, система повинна протистояти зловмисникам, які можуть виконати 2^{128} операцій у своїй атаці.

1.1.3. Режим блочного шифрування

Блочний шифр шифрує та розшифровує лише фіксований блок даних. Щоб зашифрувати та розшифрувати відкритий текст, розмір якого перевищує розмір блоку, блочний шифр повинен працювати в поєднанні з режимом блочного шифру. Ми розглянемо два режими блочного шифрування, а саме: режими лічильника (CTR) і ланцюжка блоків шифру (CBC). Ці два режими визначені в [28].

У режимі лічильника (CTR) блок відкритого тексту *j* обробляється XOR із вихідним блоком O_i, який генерується шляхом шифрування лічильника *T* ключем *K*, де *j* — номер блоку із загальної кількості *n* блоків в повідомленні. *O* формує значення потоку ключів. Для останнього блоку повідомлення O_n обробляється XOR із старшими бітами P_n, а решта бітів відкидається. Режим CTR перетворює базовий блоковий шифр на потоковий (рисунок 1.2). Цей процес визначається наступним [28]:

Для шифрування:

$O_j = CIPH_K(T_j)$ for j = 1….*n* (де *n* – кількість блоків)

$C_i = P_j \oplus O_j$

Для розшифровки:

$O_j = CIPH_K(T_j)$ for j = 1….*n* (де *n* – кількість блоків)

$P_j = C_j \oplus O_j$

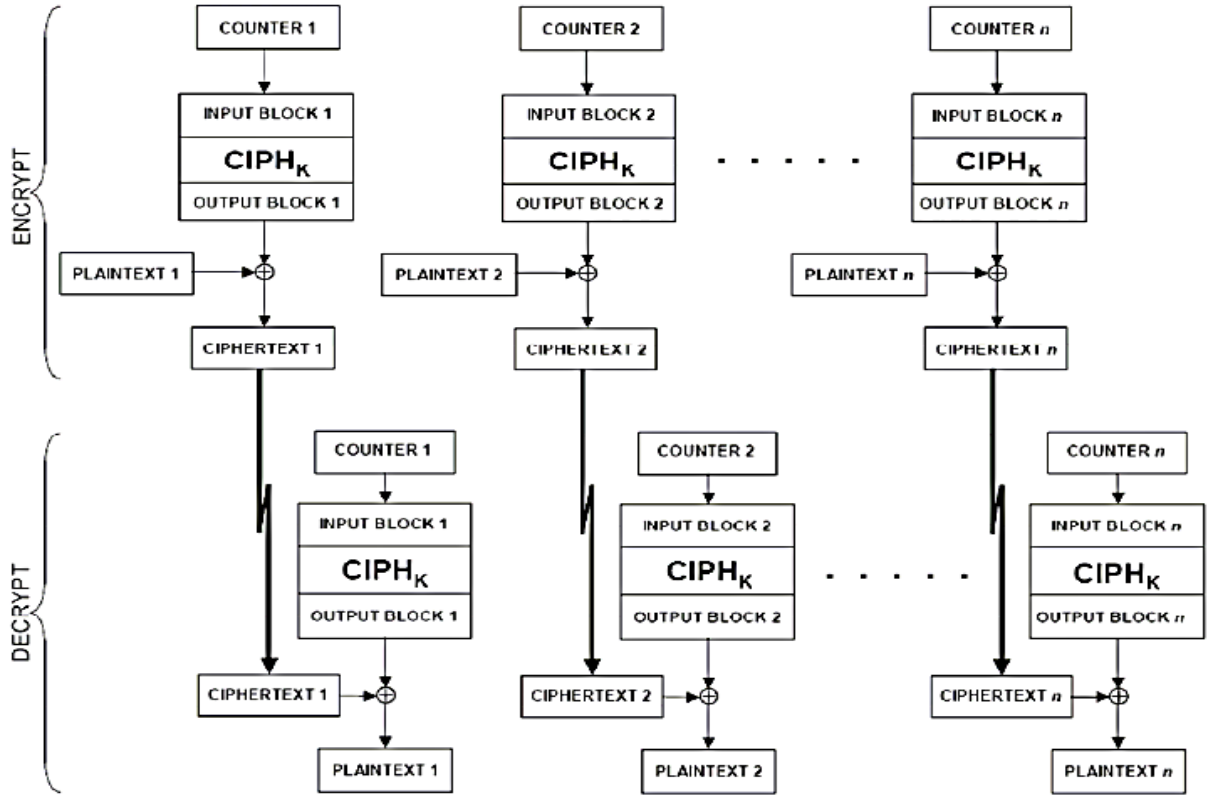


Рисунок 1.2 — Режим лічильника

Найважливішим аспектом, пов’язаним з *T*, є те, що його не можна повторно використовувати [7, 22]. Розмір *T* має бути таким же великим, як розмір блоку. Типове налаштування для 128-блочного шифру буде таким: $T_j = NC \,||\, j$, де *NC* може складатися з 48-бітного номера повідомлення, 16-бітів одноразових даних і 64-бітів для *j* [7]. Таким чином, система обмежена шифруванням 2^{48} різних повідомлень за допомогою одного ключа та обмежує кожне повідомлення 2^{68} байтами.

CTR може здійснювати попередню обробку та може використовувати переваги апаратного розпаралелювання через те, що процес шифрування не залежить від відкритого тексту [22]. Процеси шифрування та дешифрування залежать лише від функції шифрування, що сприяє простоті під час впровадження. Якщо в якомусь блоці зашифрованого тексту виникає помилка перевертання бітів, після дешифрування, помилка локалізується у відповідному

блоці відкритого тексту. Режим CTR не викликає проблем зіткнення, оскільки немає рівноцінних блоків зашифрованого тексту [7]. Одноразовий номер у лічильнику ніколи не можна використовувати повторно. Якщо його використовувати повторно, лічильник не буде унікальним, і це призведе до ідентичного значення потоку ключів (припускаючи, що ключ є однаковим для всього процесу шифрування). Такий інцидент призведе до витоку інформації про все повідомлення. Безпека режиму CTR залежить від міцності базового шифру. Якщо шифр слабкий (наприклад, DES), тоді легше виконати диференціальний криптоаналіз.

Для ланцюжка блоків шифру, щоб створити перший блок зашифрованого тексту, IV виконує XOR з першим блоком відкритого тексту. Ці вхідні дані вводяться в шифр шифрування та шифруються секретним ключем. Наступні блоки відкритого тексту об'єднуються XOR з попереднім блоком зашифрованого тексту, таким чином створюючи ланцюжок. Існує дві фундаментальні вимоги [7, 39] до режиму CBC. Перша вимога полягає в тому, що біти відкритого тексту повинні бути кратними розміру блоку; інакше цей відкритий текст потрібно відформатувати, додавши додаткові біти, щоб отримати бажану довжину. Друга вимога полягає в тому, що IV має бути унікальним для кожного процесу шифрування. Якщо друга вимога не виконується, між двома повідомленнями може бути виявлено шаблон ідентичних повідомлень першого блоку.

Функція дешифрування в режимі шифрування просто розшифровує дані, і хоча вона може створювати безглузді дані, вона все одно розшифровує модифікований зашифрований текст у деякий (модифікований і, можливо, безглуздий) відкритий текст. Майже в усіх ситуаціях шкода, яку можуть завдати модифіковані повідомлення, набагато більша, ніж шкода від витоку відкритого тексту [7]. У такій ситуації рекомендується створити MAC зашифрованого тексту та додати цей MAC до зашифрованого тексту.

Лічильник і IV мають однаковий принцип безпеки, і існують різні способи створення IV. Деякі з цих методів:

- **Фіксований IV**: у фіксованому IV, IV є тим самим значенням, яке використовується в усіх шифруваннях. Це призводить до згаданої раніше

проблеми шифрування ідентичних перших блоків повідомлень між двома повідомленнями.

- **Лічильник IV**: Ідея лічильника IV полягає в тому, що IV збільшується на 1 після кожного шифрування. Наприклад, IV = 0 використовується для першого повідомлення, IV = 1 для другого, IV = 2 для третього і так далі. Проблема тут полягає в тому, що значень кожного біта може бути недостатньо для зміни двох ідентичних перших блоків з двох повідомлень під час XOR.
- **Випадковий IV**: це випадковий рядок, згенерований генератором випадкових чисел. Є два недоліки. Перший полягає в тому, що алгоритм шифрування повинен мати доступ до джерела випадковості. Це вимагає реалізації хорошого генератора випадкових чисел. Друга проблема полягає в тому, що одержувач повинен знати IV, тому IV надсилається як перший блок перед рештою зашифрованого тексту. Результатом цього є те, що все повідомлення стає на один блок більшим, таким чином додаючись накладних витрат, якщо повідомлення невелике.
- **Негенерований IV**: найкращий спосіб генерації IV за допомогою одноразового номера [7]. Як правило, одноразовий номер – це певний номер повідомлення, можливо в поєднанні з іншою інформацією. Номери повідомлень використовуються для збереження повідомлень у правильному порядку та для виявлення дублікатів повідомлень, які використовуються для атак у відповідь. Одноразовий номер, що використовується, має бути унікальним для кожного шифрування, що використовується в двонаправленому трафіку між відправником і одержувачем. Коли одноразовий номер складений, він шифрується за допомогою блочного шифру для створення IV.

1.1.4 Шифр асиметричного ключа

На відміну від криптографії з симетричним ключем, криптографія з асиметричним ключем передбачає використання двох окремих ключів, а саме відкритого та закритого ключів, а алгоритми базуються на математичних функціях, таких як модульна та логарифмічна арифметика, замість операцій із бітовими шаблонами. Використання двох ключів має значні наслідки в трьох сферах застосування, а саме: конфіденційність, автентифікація та розподіл

ключів. Однак не всі алгоритми асиметричного ключа можуть виконувати всі три вимоги. Таблиця 2.3 підсумовує застосування різних криптографічних алгоритмів з асиметричним ключем.

Таблиця 1.3 — Застосування криптосистеми з асиметричним ключем

Алгоритм	**Шифрування/дешифрування**	**Цифровий підпис**	**Обмін ключами**
RSA	Так	Так	Так
Diffie-Hellman	Ні	Ні	Так
DSA	Ні	Так	Ні
Elliptic Curve	Так	Так	Так

З таблиці 1.3 ми бачимо, що шифри RSA та еліптичної кривої є найбільш універсальними.

Відповідно до [4] криптосистема з асиметричним ключем повинна відповідати наступним умовам щодо шифрування та дешифрування:

1. Суб'єкту *B* легко згенерувати пару відкритих і закритих ключів (PUB_B і PRI_B).

2. Для відправника *A*, знаючи відкритий ключ і повідомлення, яке потрібно зашифрувати, *P* легко згенерувати відповідний зашифрований текст: $\mathbf{C = CIPH_{PUB\_B}(P)}$.

3. Одержувачу *B* легко розшифрувати отриманий зашифрований текст *C* за допомогою закритого ключа для відновлення вихідного повідомлення: $\mathbf{P = CIPH^{-1}_{PRI\_B}(C)}$.

4. Для зловмисника, знаючи відкритий ключ PUB_B, обчислювально неможливо визначити закритий ключ PRI_B.

5. Для зловмисника, знаючи PUB_B і *C*, неможливо відновити *M*.

Система RSA [34] є, ймовірно, найпоширенішою криптосистемою з відкритим ключем у світі. Вона також дуже універсальна, оскільки забезпечує конфіденційність даних, цифрові підписи та обмін ключами. Сила RSA випливає з розкладання великих чисел [38]. Алгоритм RSA – це блочний шифр, у якому відкритий текст і зашифрований текст є цілими числами від 0 до n – 1 для деякого n. Більш простий опис алгоритму RSA виглядає наступним чином [38, 39]:

Компоненти відкритого ключа:

n = добуток двох великих простих чисел, а саме *p* і *q*, де *p* і *q* залишаються секретними

e = випадкове число, відносно просте до $(p-1)(q-1)$ і менше за нього.

Компоненти закритого ключа:

$d = e^{-1}\ mod\ \big((p-1)(q-1)\big)$ мультиплікативне обернення

$e\ mod\ \big((p-1)(q-1)\big)$

Схема шифрування (RSAEP): $C = P^e\ mod\ n$

Схема дешифрування (RSADP): $P = C^d mod\ n$

Цифровий підпис: $S = P^d\ mod\ n$

$P = S^e\ mod\ n = P^{ed}\ mod\ n$ (для перевірки підпису)

Щоб RSA був задовільним для шифрування з відкритим ключем, необхідно виконати такі вимоги:

1. Два великих простих числа p і q повинні залишатися таємними.

2. Можна знайти такі значення e, d, n, що $M^{ed} = M\ mod\ n$ для всіх $M < n$.

3. Відносно легко обчислити M^e і C для всіх значень $M < n$.

4. Неможливо визначити d за e і n.

Криптосистема з асиметричним ключем включає математичні функції, які вимагають більше обчислень порівняно з криптографією з симетричним ключем. Шифрування RSA та перевірка підпису відбуваються швидше, якщо використовується низьке значення для e; однак це було б небезпечно [7].

Використовуючи рекомендації щодо використання китайської теореми про залишки (CRT) для структури закритого ключа може прискорити швидкість дешифрування на одному процесорі. Використовуючи CRT, можна заощадити час обчислення в 3-4 рази в типовій реалізації, тоді як недоліком є додаткова складність програмного забезпечення та необхідні перетворення [7]. Це підвищення швидкості буде корисним, якщо RSA використовується на мобільному пристрої, де швидкість ЦП обмежена. Специфікацію RSA з використанням CRT можна знайти в [35].

Розглянемо деякі питання безпеки RSA.

Розкладання n на множники: найкращий спосіб розв’язати приватний ключ — це розкласти n на множники для отримання p і q, щоб відновити решту компонентів приватного ключа. Іншою альтернативою розкладу n є визначення $(p-1)(q-1)$, однак обсяг роботи такий самий, як і розкладання n.

Незахищений IND-CCA: схема шифрування, яка є семантично захищеною під адаптованою атакою зашифрованого тексту, називається безпечною IND-CCA [35]. Проблема з примітивами шифрування RSA полягає в тому, що вони є детермінованими, тому не підтримують захист IND-CCA. Щоб полегшити цю проблему, необхідна функція кодування для руйнування будь-якої структури повідомлення [32]. Для цього ми розглянемо функцію кодування RSAES-OAEP [35], яка поєднує в собі примітиви RSAEP і RSADP з методом кодування EME-OAEP. EME–OAEP базується на схемі оптимального асиметричного шифрування Bellare та Rogaway [1] і сумісна зі схемою IFES, визначеною в IEEE Std 1363-2000, де примітивами шифрування та дешифрування є IFEP-RSA та IFDP-RSA та схема кодування повідомлень EME-OAEP (OAEP означає «оптимальне доповнення асиметричного шифрування»). RSAES-OAEP може працювати з повідомленнями довжиною до $k - 2hlen - 2$ байтів, де $hlen$ – це довжина результату основної хеш-функції, а k – довжина в байтах модуля RSA одержувача. Це обмеження розміру повідомлення, яке можна надіслати. Наприклад, якщо k дорівнює 1024 біт, а $hlen$ — 160 біт, тоді розмір повідомлення обмежено 87 байтами замість 128 байт.

Загальна атака модуля: можлива реалізація RSA дає всім однакове n, але різні значення для e і d. Проблема полягає в тому, що якщо одне й те саме повідомлення коли-небудь шифрується двома різними експонентами (обидва мають однаковий модуль), і ці два показники є відносно простими, тоді відкритий текст можна відновити без будь-якого з показників дешифрування [38].

Шифрування невеликого відкритого тексту: якщо P має бути зашифровано і P менше ніж n (з точки зору кількості бітів), то існує ймовірність, що скорочення за модулем ніколи не відбудиться [7]. Практичний сценарій: якщо користувач шифрує 256-бітний секретний ключ за допомогою RSA, тоді зашифрований ключ менше ніж $2^{256*5} = 2^{1280}$, що менше ніж 2^{2048}, якщо припустити, що n становить 2048 бітів.

Атака з низьким показником дешифрування: атака, запропонована Вінером [47], стверджує, що якщо d становить приблизно одну чверть розміру модуля n, а e менше n, то d можна відновити. Ця атака не становить загрози, якщо d має приблизно такий самий розмір, як n.

Існують інші добре відомі асиметричні алгоритми, які використовуються в комерційних продуктах у поєднанні з RSA, однак ці алгоритми мають свої обмеження. Такими алгоритмами є Діффі-Хеллман [4], Ель-Гамаль [5], Алгоритм цифрового підпису [29] і Еліптична криптографія [27].

Проблема з алгоритмом Діффі-Хеллмана (DH) полягає в тому, що він не такий універсальний, як RSA, і генерація ключів для протоколу DH може бути надто дорогою з точки зору обчислень для мобільного пристрою.

ElGamal настільки ж універсальний, як і RSA, однак створений зашифрований текст вдвічі більший за відкритий текст; тому він погано підходить у середовищі з високою затримкою та низькою пропускною здатністю.

Алгоритм цифрового підпису (DSA) не такий універсальний, як RSA, і повільніший за RSA з точки зору перевірки підпису [38]. Ще одна проблема з DSA полягає в тому, що розмір ключа варіюється від 512 до 1024 біт, тому вимагати потужніший розмір ключа понад 1024 біт неможливо.

Основна перевага криптографії з еліптичною кривою (ECC) порівняно з RSA полягає в тому, що вона, здається, забезпечує однакову безпеку для значно меншого розміру ключа, тим самим зменшуючи накладні витрати на обробку [45]. Проте різні аспекти ECC були запатентовані багатьма людьми та компаніями по всьому світу, особливо функція шифрування.

Розглянемо деякі аспекти безпеки, що стосуються криптографії з асиметричним ключем. Ці міркування містять важливі висновки, які можуть бути використанні для впровадження КСБМП.

Згідно з нашим оглядом літератури про криптосистеми з асиметричним ключем, RSA все ще є рекомендованою реалізацією. Слід пам’ятати, що звичайне шифрування RSA не є безпечним для IND-CCA. Якщо шифр RSA використовується окремо для шифрування повідомлення, той самий зашифрований текст створюватиметься кожного разу, коли шифруватимуться той самий відкритий ключ і відкритий текст. Схеми кодування, такі як RSAEP-OAEP, гарантують, що для однакових повідомлень не створюється однаковий зашифрований текст. Важливість такого кодування є надзвичайно важливою, оскільки ймовірність повідомлення, зашифрованого мобільним пристроєм, буде однаковою через малий розмір повідомлення. Для захисту даних до 2030 року

RSA Labs рекомендує 2048-бітний ключ [17]. Крім того, через тривалий процес оновлення ключів RSA для нових програм рекомендується 2048-біт.

Підходи до управління асиметричними ключами дещо відрізняються від симетричних. Для асиметричних ключів пари приватного та відкритого ключів можуть бути згенеровані тимчасово під час зв'язку між двома або більше об'єктами, але в практичних системах відкритий та приватний ключі використовуються протягом більш тривалого часу після їх генерації.

Криптографія з асиметричним ключем чутлива до атак типу "людина посередині". Таким чином, щоб криптографія з асиметричним ключем працювала безпечно, відкритий ключ, який використовується для шифрування, має належати законному власнику, який виконуватиме розшифровку.

Є три способи [38], якими можна отримати відкритий ключ. Перший — це отримання відкритого ключа особисто від одержувача. Другий спосіб — від публічного сертифікат ключа, який отриманий від центру сертифікації (CA) – організації, якій довірено видавати сертифікати відкритого ключа – через їхню централізовану базу даних. Цей метод найчастіше використовується в інфраструктурі відкритих ключів (PKI). Третій — відкритий ключ одержувача можна отримати за допомогою розподіленого керування ключами за допомогою інтродьюсерів, якщо сторони зв'язку не довіряють жодному CA.

У більш практичній ситуації криптографія з асиметричним ключем використовується PKI. PKI складається з протоколів, служб і стандартів, що підтримують програми криптографії з відкритим ключем. Серед послуг, які, ймовірно, можна знайти в PKI, є наступні [36]:

- Реєстрація ключа: видача нового сертифіката відкритого ключа.
- Відкликання сертифіката: анулювання раніше виданого сертифіката. Це часто необхідно, коли приватний ключ зламано до закінчення терміну дії відповідного сертифіката відкритого ключа.
- Вибір ключа: отримання відкритого ключа сторони.
- Оцінка довіри: визначення того, чи є сертифікат дійсним і які операції він дозволяє.

Існують певні ризики, пов'язані з PKI, а саме рівень довіри до CA та особи, яку CA сертифікував сертифікат, а також швидкість і надійність відкликання [6, 7].

Використання ключа в криптосистемах з асиметричним ключем відрізняється від використання ключа з симетричним ключем [38]. Одна пара відкритих і закритих ключів недостатньо хороша, оскільки для шифрування та підпису можуть знадобитися окремі пари ключів. Крім того, одна особа може володіти різними парами відкритих і приватних ключів, які будуть використовуватися для різних ролей, якими володіє ця особа. Завдання полягає в тому, щоб відстежувати ці кілька пар ключів і переконатися, що вони використовуються відповідно до призначення.

1.1.5 Хеш-функції

Хеш-функції можна використовувати для автентифікації повідомлень, отримання ключа, генерації псевдовипадкових чисел і цифрового підпису. Хеш-функція H повинна мати такі властивості [39]:

1. H можна застосувати до блоку даних будь-якого розміру.

2. H створює вихідні дані фіксованої довжини.

3. $H(x)$ відносно легко обчислити для будь-яких заданих даних x, що робить практичними як апаратні, так і програмні реалізації.

4. Для будь-якого дайджесту d обчислювально неможливо знайти такий x, щоб $H(x) = d$. Ця точка визначає односторонню властивість.

5. Для будь-якого даного блоку x обчислювально неможливо знайти $y \neq x$ з $H(y) \neq H(x)$.

6. Обчислювально неможливо знайти будь-яку пару повідомлень (x, y), щоб $H(x) = H(y)$.

Хеш-функція, яка задовольняє перші п’ять пунктів, називається слабкою хеш-функцією, а якщо задовольняється 6-та властивість, то вона називається сильною хеш-функцією.

Алгоритм безпечного хешування (SHA) був розроблений NSA і стандартизований NIST і опублікований як FIPS PUB 180. Списки алгоритмів SHA: SHA-0, SHA-1, SHA-2 (SHA-256/224 і SHA-512/384). SHA-256 приймає розмір вхідного повідомлення менше 2^{64}-бітів та створює дайджест 256 бітів. Можливість отримати два повідомлення з однаковим дайджестом становить близько 2^{128} операцій для SHA-256 із складність пошуку повідомлення з заданим дайджестом становить близько 2^{256} операцій.

Проблеми безпеки, які пов’язані з односторонніми хеш-функціями [7] є:

- Розширення довжини. Повідомлення m розбивається на блоки $m_1, \ldots, m_k$ і хешується до значення d. Інше повідомлення m' розбивається на блоки $m_1, \ldots, m_k, m_{k+1}$. Оскільки перші k блоків m' ідентичні k блокам повідомлення m, хеш-значення $H(m)$ є лише проміжним хеш-значенням після k блоків у обчисленні $H(m')$, що дає $H'(H(m), m_{k+1})$. Проблема розширення довжини існує, оскільки немає спеціальної обробки в кінці обчислення хеш-функції. Результатом є те, що $H(m)$ надає пряму інформацію про проміжний стан після перших k блоків m'. Зловмисник може побудувати кілька відповідних пар (m, m') і перевірити цей зв'язок. Як наслідок розширення довжини, зловмисник може додати текст до m і оновити хеш, щоб він відповідав новому повідомленню.
- Часткова колізія повідомлень. Припустимо, що система аутентифікує повідомлення за допомогою $H(m \parallel s)$, де s — ключ автентифікації. Зловмисник може вибрати повідомлення m, але система аутентифікує лише одне повідомлення. Ідеальна хеш-функція забезпечить рівень безпеки n-біт. Зловмисник може знайти дві пари повідомлень (m, m'), які призводять до зіткнення, коли їх хешує H. Це можна зробити за допомогою атаки на день народження приблизно з $2^{n/2}$ кроків. Зловмисник системи може аутентифікувати m і замінити повідомлення на m'. Через ітераційну природу односторонніх хеш-функцій хешування m і m' призводить до того самого значення $H(m \parallel s) = H(m' \parallel s)$ для кожного S. Це означає, що зловмиснику не потрібно, щоб виводити s і все повідомлення $(m \parallel s)$ містить часткове зіткнення, яка виникає в m і m'.

1.1.6 Код автентифікації повідомлення

Код автентифікації повідомлення (MAC) — це конструкція, яка запобігає підробці повідомлень. MAC має перші п’ять властивостей односторонніх хеш-функцій, за винятком того, що він передбачає використання секретного ключа як додаткового введення разом із повідомленням, яке потрібно аутентифікувати. Це означає, що замість звичайної процедури застосування хеш-функції для створення дайджесту d із вхідних даних x, позначених $H(x) = d$,

ми тепер маємо $M(s,\ x) = d$, де M — функція MAC, яка генерує d, беручи в якості вхідних даних секрет s і повідомлення x.

Односторонню хеш-функцію не можна використовувати як MAC, оскільки вона не залежить від секретного ключа. Односторонні хеш-функції можуть працювати подібно до MAC, як $H(s \parallel x)$, $H(x \parallel s)$ або $H(s \parallel x \parallel s)$, однак [7] зазначає, що вони не будуть безпечними, оскільки мають s у передній частині, що дозволяє атаки розширення довжини, а наявність s у кінці дозволяє розумну атаку відновлення ключа приблизно за $2^{n/2}$ кроки.

Було кілька пропозицій щодо включення секретного ключа в існуючий односторонній алгоритм хешування. Найбільшу підтримку отримав підхід HMAC [39]. HMAC був виданий як RFC 2104 [15] і був обраний як обов'язковий для впровадження MAC для IPSec, і використовується в інших інтернет протоколах, таких як TLS і безпечні електронні транзакції (SET) [39].

RFC 2104 перераховує такі цілі для HMAC:

- використовувати без модифікацій доступні односторонні хеш-функції (зокрема, функції, які добре працюють у програмному забезпеченні та для яких код вільно та широко доступний);
- щоб забезпечити можливість заміни вбудованої хеш-функції у випадку, якщо знайдуться або знадобляться швидші або безпечніші хеш-функції;
- щоб зберегти початкову продуктивність хеш-функції без значного погіршення;
- використовувати та поводитися з ключами простим способом;
- мати добре зрозумілий криптографічний аналіз надійності механізму автентифікації на основі розумних припущень щодо вбудованої функції.

Перші дві цілі розглядають HMAC як програмний компонент. HMAC розглядається як «чорна скринька», яка приймає односторонню хеш-функцію, яка буде вбудована як частина його реалізації. Що ще важливіше, якщо односторонню хеш-функцію «зламано», її можна замінити більш безпечною. Це означає, що міцність HMAC може бути доведена безпечною, якщо вбудована хеш-функція безпечна. HMAC також забезпечує підтримку, щоб протистояти розширенню довжини та колізіям часткових повідомлень [7, 15]. Значення секретного ключа, що використовується як один із вхідних даних у HMAC,

також захищено від атак відновлення ключа, які можуть бути запущені зловмисником, який не взаємодіє з системою (офлайн-атаки).

1.1.7 Сертифікати відкритого ключа та цифрові підписи

Існує багато різних форматів сертифікатів, але найпоширенішим є сертифікат X.509 версії 3. Стандарт ISO для структури цифрових сертифікатів базується на стандарті X.509 [16]. X.509 визначає структуру для надання послуг автентифікації через каталог X.500, який складається зі сховища сертифікатів відкритих ключів. Кожен сертифікат складається з відкритого ключа користувача та підписується закритим ключем CA. X.509 також містить стандарти для списку відкликаних сертифікатів (CRL), який надає засоби для відкликання сертифікатів. Зараз X.509 використовується в SSL/TLS, IPSec, захищених/багатоцільових розширеннях Інтернет-пошти (S/MIME) і SET.

Сертифіковані сертифікати відкритого ключа X.509 створюються довіреним CA і розміщуються в каталозі CA або користувачем. Сервер каталогів лише надає користувачам легкодоступне місце для отримання сертифікатів.

Сертифікати користувача, створені CA, мають такі характеристики [39]:

• будь-який користувач, який має доступ до відкритого ключа CA, може відновити відкритий ключ користувача, який був сертифікований;

• жодна сторона, окрім центру сертифікації, не може змінити сертифікат без виявлення цього.

Наступні елементи сертифіката X.509 узагальнено з [39]:

• Версія: це поле описує версію закодованого сертифіката. Версії включають 1, 2 або 3.

• Серійний номер: серійний номер має бути додатним цілим числом, призначеним CA кожному сертифікату. Він має бути унікальним для кожного сертифіката, виданого даним CA (тобто ім’я видавця та серійний номер ідентифікують унікальний сертифікат).

• Ідентифікатор алгоритму підпису: це поле містить ідентифікатор алгоритму, який використовується CA для підпису сертифіката. Вміст поля додаткових параметрів змінюватиметься залежно від визначеного алгоритму.

• Ім’я видавця: поле видавця визначає організацію, яка підписала та видала сертифікат. Поле емітента має містити непорожнє розпізнаване ім’я (DN).

• Період дії: Період дії сертифіката – це інтервал часу, протягом якого CA гарантує, що він зберігатиме інформацію про статус сертифіката. Поле представлено у вигляді послідовності двох дат: дати початку терміну дії сертифіката (не раніше) та дати закінчення терміну дії сертифіката (не пізніше).

• Ім’я суб’єкта: поле суб’єкта ідентифікує об’єкт, пов’язаний із відкритим ключем, який зберігається в полі відкритого ключа суб’єкта.

• Інформація про відкритий ключ суб’єкта: це поле використовується для розміщення відкритого ключа та ідентифікації алгоритму, з яким використовується ключ (наприклад, RSA, DSA або Diffie-Hellman).

• Унікальні ідентифікатори видавця та суб’єкта: ці поля мають з’являтися лише у випадку версії 2 або 3. Унікальні ідентифікатори суб’єкта та видавця присутні в сертифікаті, щоб врахувати можливість повторного використання імен суб’єктів та/або емітента з часом.

• Розширення: розширення, визначені для сертифікатів X.509 v3, надають методи для асоціювання додаткових атрибутів із користувачами або відкритими ключами та для керування ієрархією сертифікації. Кожне розширення в сертифікаті позначається як критичне або некритичне. Система, яка використовує сертифікат, повинна відхилити сертифікат, якщо вона натрапить на критичне розширення, яке вона не розпізнає; однак некритичне розширення може бути проігноровано, якщо воно не розпізнано.

• Підпис: поле підпису — це хешоване значення всіх полів у сертифікаті, зашифроване приватним ключем CA. Це поле містить ідентифікатор алгоритму підпису.

Стандарт X.509 є ключовим елементом PKI. Версія 3 дозволяє використовувати додаткові розширення. Цифрові підписи забезпечують автентифікацію, авторизацію та неспростовність. Стандарт PKCS# v2.1 [35] рекомендує RSASSA-PSS в інтересах підвищення надійності для нових програм. Підписання на основі RSA має ризик безпеки, що зловмисник може змусити легітимну сутність A підписати M_3, створивши два повідомлення M_1 і

M_2, щоб $M_3 = M_1 M_2 (mod\ n)$. Якщо зловмисник може змусити A підписати M_1 і M_2, M_3 можна обчислити шляхом множення $\left(M_1^d * M_2^d\right) mod\ n = M_3^d mod\ n$ [7].

1.2 Огляд пакетів мобільної криптографії

Щоб забезпечити достатню безпеку в середовищі мобільних додатків, потрібен криптографічний інструментарій. Ми дослідимо два криптографічні пакети, а саме Bouncy Castle API та Secure and Trust Service API.

1.2.1 Bouncy Castle API

API криптографії Bouncy Castle (BC) [2] — це безкоштовний постачальник JCE з відкритим вихідним кодом, який підтримується в Австралії. Розробники BC розробили власний легкий API, який буде включено в класи провайдерів BC JCE. Спрощена структура пакета BC продемонстрована на рисунку 1.3. Пакет BC має власну реалізацію java.math.BigInteger, java.security.SecureRandom і java.io через відсутність підтримки таких класів і пакета на MIDP. Базовим пакетом, який підтримує криптографічні алгоритми та схеми заповнення, є пакет org.bouncycastle.crypto. Пакет org.bouncycastle.asn1 підтримує розбір і запис об’єктів ASN.1, що корисно для обробки цифрових сертифікатів X.509.

Для впровадження програми Pretty Good Privacy (PGP) з відкритим кодом розробник може використовувати класи в пакеті org.bouncycastle.bcpg. Пакет org.bouncycastle.math.ec забезпечує додаткову математичну підтримку криптографії еліптичної кривої. Допоміжні класи в org.bouncycastle.util можна використовувати для створення та читання рядків Base64 і Hexadecimal. Ця утиліта корисна, якщо потрібно, щоб зашифрований текст відображався як шістнадцятковий рядок.

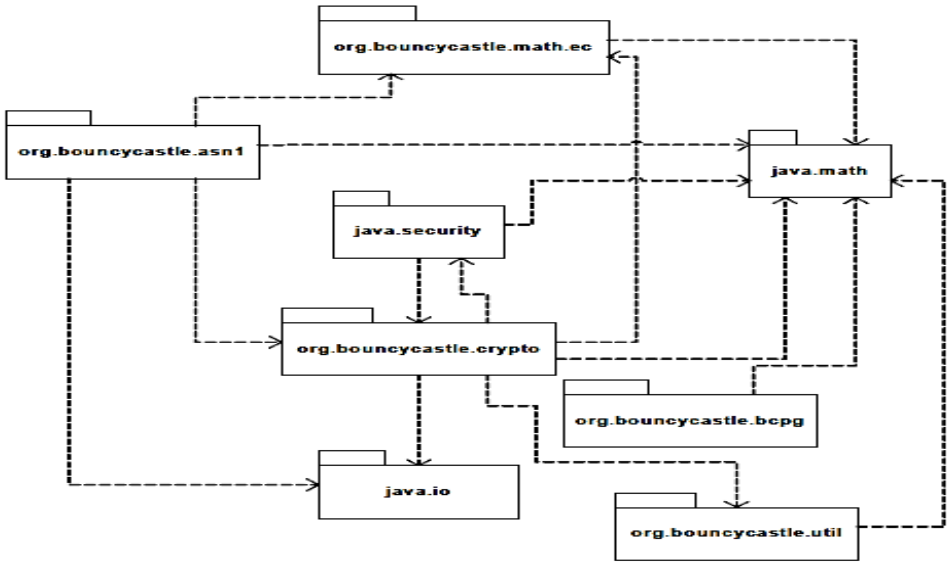


Рисунок 1.3 — Структура залежностей пакета API Bouncy Castle

BC підтримує криптографічні алгоритми вище описані однак він також містить слабкі алгоритми, такі як DES. Розмір легкого файлу jar API трохи менше 1 МБ. Однак більшість мобільних додатків потребуватимуть лише підмножини алгоритмів BC. Розробник може використовувати обфускатор, щоб зменшити непотрібні класи.

Спеціальна модель розробки (ad hoc development mode) має проблеми [49]:

• Підтримується забагато алгоритмів. Не має оптимізації, що призводить до відносно низької продуктивності, особливо для деяких асиметричних ключових алгоритмів.

• Дизайн BC API є гнучким, але досить складним, і початківцям важко освоїтися. Немає деяких зручних для розробників функцій API. Наприклад, йому бракує набору готових до використання API серіалізації загального ключа.

• Завдання управління ключами в BC покладено на розробника.

• Початкове число, яке використовується для ініціалізації класу `SecureRandom`, може бути невипадковим, оскільки початкове значення отримано із системного часу. Це означає, що шанси отримати секретний ключ, згенерований для шифрування з симетричним ключем, високі.

• Немає методів API для завантаження сертифіката та закритих ключів зі смарт-карти. Прикладом смарт-карти в мобільному телефоні є SIM-карта. BC має методи завантаження сертифікатів і приватних ключів із файлу, і розробник повинен забезпечити їх безпечне зберігання та автентичність.

1.2.2 Secure and Trust Service API

Secure and Trust Service API (SATSA) поширюється як додатковий пакет у MIDP на певних мобільних телефонах. Специфікація SATSA [43] стверджує, що і SATSA, і програма, яка використовує SATSA, повинні довіряти ОС. SATSA залежить від певних послуг, які надає операційна система мобільного пристрою. Структура залежностей пакетів SATSA представлена на рисунку 1.4. Мотивація для вивчення SATSA полягає в тому, що це поточний криптографічний пакет для MIDP 3.0.

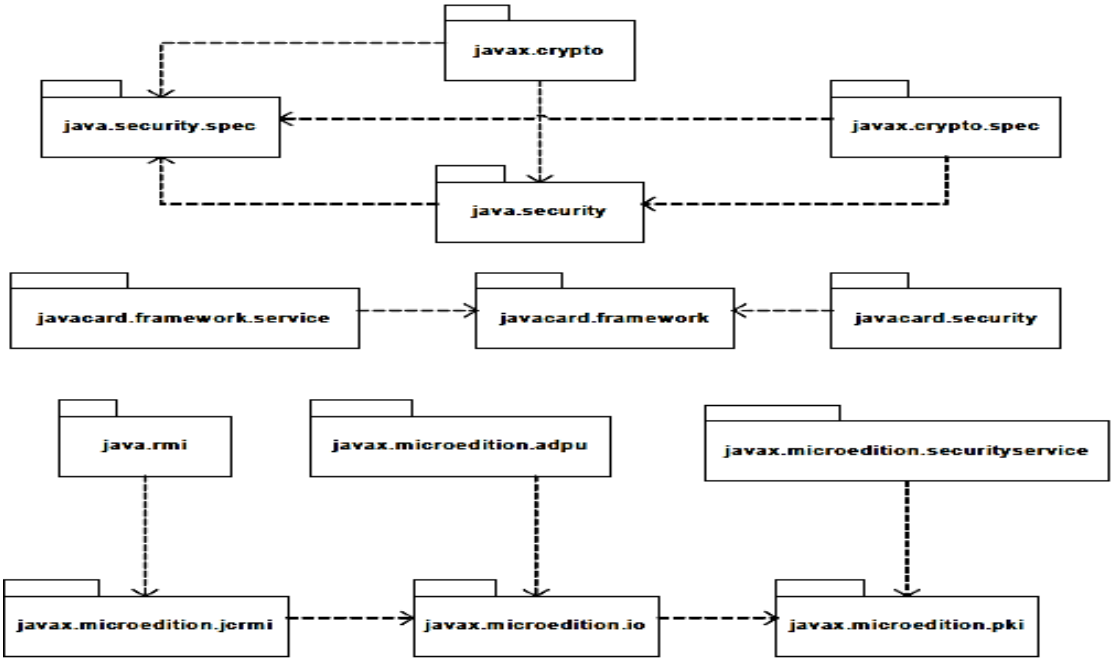


Рисунок 1.4 — Структура залежностей пакетів Secure and Trust Service API

Secure and Trust Service API (SATSA) — це додатковий пакет, який дозволяє мобільній програмі Java [31]:

• Робота з публічними цифровими сертифікатами, відкритими та закритими ключами, дайджестами повідомлень та цифровими підписами.

• Створюйте, зберігайте та використовуйте облікові дані користувача на основі цифрових сертифікатів X.509

• Шифруйте дані за допомогою криптографії з асиметричним і симетричним ключем.

SATSA покладається на елемент безпеки, яким може бути вбудована SIM-карта, смарт-карта або спеціальне обладнання всередині пристрою для виконання операцій безпеки. Точний тип елемента безпеки є прозорим для розробника. Елемент безпеки визначається реалізацією SATSA для конкретного пристрою.

Специфікація SATSA визначає чотири різні API:

• SATSA-APDU дозволяє програмам спілкуватися з програмами смарт-карт за допомогою протоколу низького рівня. Пакети, що належать до цього API: javax.microedition.apdu, який визначає обробник протоколу APDU (Application Protocol Data Unit) для зв'язку ISO7816-4 із пристроєм смарт-картки.

• SATSA-JCRMI надає альтернативний метод для зв'язку з програмами смарт-карт за допомогою Java Card Remote Method Invocation (JCRMI). До цього API належать пакети java.rmi, javacard.framework, javacard.security та javax.microedition.jcrmi.

• SATSA-PKI дозволяє програмам використовувати смарт-картки для цифрового підпису даних і керування сертифікатами користувачів. Пакети в цьому API: javax.microedition.pki та javax.microedition.securityservice. Пакет javax.microedition.pki визначає класи для підтримки базового керування сертифікатами користувачів, тоді як javax.microedition.securityservice визначає класи для генерації цифрових підписів на рівні програми, які відповідають формату Cryptographic Message Syntax (CMS).

• SATSA-CRYPTO — це криптографічний API загального призначення, який підтримує дайджести повідомлень, цифрові підписи та шифри. До цього API належать такі пакети: java.security та javax.crypto. Пакети java.security та javax.crypto надають класи та інтерфейси для системи безпеки та криптографічних операцій відповідно. Їхня специфікація підпакетів надає класи та інтерфейси для ключових специфікацій і специфікацій параметрів алгоритму.

Generic Connectivity Framework (GCF), визначену в javax.microedition.io, змінено, щоб включити нові підключення APDU та JCRMI для смарт-карт.

SATSA API має такі недоліки:

• Процес перевірки підпису не такий простий, як створення підпису. SATSA генерує підписані повідомлення у форматі CMS. Однак, щоб перевірити підпис за допомогою відкритого ключа, програмі потрібно проаналізувати вхідні дані на частини даних і підпису відповідно.

• Генерація підпису та перевірка підпису не обробляються однаково з точки зору того, як інформація подається користувачеві. Коли дані підписуються користувачем, базова реалізація SATSA бере на себе контроль

над інтерфейсом користувача (UI) і надає користувачеві сертифікат, який буде використовуватися для підпису, разом із даними, які потрібно підписати.

• SATSA не може перевірити сертифікати. Розробник повинен реалізувати процес перевірки сертифіката та вилучення відкритого ключа із сертифіката, а також надати користувачеві сертифікат і підписані дані.

• Приватні ключі, що зберігаються на смарт-картах, не можна використовувати для підпису, оскільки немає доступних методів для активації цього процесу.

• Секретний ключ, згенерований SATSA для шифрування з симетричним ключем, базується на матеріалі ключа. Цей матеріал зазвичай є PIN-кодом користувача або якимось паролем. Проблема полягає в тому, що ця схема підходить лише для шифрування даних на пристрої, оскільки створений ключ не є ключем сеансу.

• Як згадувалося раніше, SATSA підтримується лише на певних моделях телефонів, а телефони без SATSA повинні використовувати сторонні криптографічні інструменти, такі як BC.

• SATSA підтримує обмежені криптографічні алгоритми. Наприклад, він не підтримує SHA-256 або будь-які обчислювальні алгоритми MAC.

1.3. Критерії та методи кількісної оцінки дизайну архітектури системи

Існують різні способи оцінки такої системи, як КСБМП, і деякі концепції, взяті з [21, 33, 40], були об’єднані, щоб задовольнити нашу оцінку. Розглянемо критерії оцінки дизайну архітектури системи, а саме:

Динамічна розширюваність: здатність фреймворку полегшувати легке включення нових алгоритмів.

Багаторазове використання: здатність фреймворку забезпечувати повторне використання на прикладному рівні.

Гнучкість: розробка для гнучкості (адаптивність) означає активне передбачення змін, яких проект може зазнати в майбутньому.

Зв'язок: міра ступеня взаємозалежності між програмними модулями. Важливим принципом проектування є зменшення кількості зв’язків.

Ефективний дизайн: КСБМП — це програмний компонент, і він має відокремлювати проблеми застосування криптографії від програми. Таким чином додаток стає легшим у реалізації.

Ремонтопридатність: важлива внутрішня якість програмного забезпечення, яка вимірює ступінь, до якого програмне забезпечення можна модифікувати з найменшими можливими витратами.

Передбачення старіння: передбачення старіння означає планування еволюції технології чи середовища, щоб програмне забезпечення продовжувало працювати або його можна було легко змінити.

Замінюваність: можливість розробника взаємозамінювати внутрішні компоненти фреймворку без будь-якого дорогого рефакторингу чи реінжинірингу.

Непорушність: це стосується впливу зміни алгоритмів на систему.

Зрозумілість: ступінь розуміння розробником функціональності будь-якого конкретного криптографічного алгоритму.

Портативність: портативність гарантує, що програмне забезпечення може працювати на якомога більшій кількості платформ.

Простота тестування: здатність інфраструктури спрощувати тестування криптографічних алгоритмів.

Метрики оцінки дизайну архітектури системи: на основі [24] розглянемо декілька корисних показників для оцінки архітектури та дизайну КСБМП. Ці показники працюють на рівні пакету, а не на рівні класу, і КСБМП найкраще оцінюється за кількістю тісно пов’язаних класів, які об’єднані в один пакет і повторно використовуються разом.

Коли багато компонентів залежать від одного конкретного пакета *X*, важко внести зміни в *X*, не вносячи змін в інші компоненти. Таким чином, *X* є незалежним пакетом, оскільки на нього немає зовнішнього впливу. Такий пакет вважається стабільним. З іншого боку, якщо пакет *Y* залежить від багатьох компонентів, зміна будь-якого з цих компонентів вплине на *Y*. Тому *Y* є залежним, що призводить до нестабільності пакета.

Метрика нестабільності описує стабільність пакета з точки зору залежностей, які виходять з пакета або входять до нього. Щоб обчислити її, нам потрібно визначити два терміни для різних типів залежностей:

- C_a або аферентні зв’язки — це кількість класів поза цим пакетом, які залежать від класів усередині цього пакета.
- C_e або еферентні зв’язки – це кількість класів у цьому пакеті, які залежать від класів за межами цього пакета.

Використовуючи ці показники, нестабільність I можна обчислити за допомогою наступного рівняння:

$$I = \frac{C_e}{c_a - C_e} \quad (6.1)$$

Цей показник має діапазон [0, 1]. Значення 0 для I вказує на максимально стабільний пакет, тоді як 1 вказує на максимально нестабільний пакет.

Стабільний пакет має бути абстрактним (складається з абстрактиних класів), щоб його стабільність не перешкоджала його розширенню. З іншого боку, нестабільний пакет має бути конкретним, що дозволяє легко змінювати конкретний код у ньому. Метрика абстракції є мірою абстрактності пакета шляхом обчислення співвідношення абстрактних класів у пакеті до загальної кількості класів у пакеті.

Рівняння абстрактності пакета складається з:

- N_c – кількість класів у пакеті.
- N_a – кількість абстрактних класів у пакеті.
- A – абстрактність.

Таким чином задано рівняння:

$$A = \frac{N_a}{N_C} \quad (6.2)$$

Метрика A коливається від 0 до 1. Нуль означає, що пакет не має абстрактних класів. Значення 1 означає, що пакет містить лише абстрактні класи.

Зв’язок між абстрактністю та нестабільністю можна описати за допомогою двовимірного графіка, який має абстрактність на вертикальній осі та нестабільність на горизонтальній осі, що представлено на рисунку 1.5. Пакети, які є максимально стабільними та абстрактними, знаходяться у верхньому лівому куті (0,1), а максимально нестабільні та конкретні – у нижньому правому куті (1,0). Оскільки не всі пакети розташовані в ідеальних точках (0,1) або (1,0), існує геометричне місце точок на графіку A/I, яке визначає області, де пакет не повинен бути (тобто зони відчуження).

Область навколо (0,0) називається зоною болю. Пакет в зоні болю — стійкий і конкретний, тому його неможливо розширити. Пакети можуть бути там за умови, що вони є незалежними. Область навколо (1,1) називається зоною непотрібності. Ця локація небажана, тому що вона максимально абстрактна і при цьому не має утриманців. Такі пакети марні.

Абстрактність і нестабільність повинні утворювати баланс один з одним у добре спроектованій архітектурі. Це означає, що пакети повинні знаходитися якомога далі від зон виключення. Геометричним місцем точок, максимально віддалених від кожної зони, є пряма, що сполучає (1,0) і (0,1). Ця лінія відома як головна послідовність (позначена зеленою лінією на рисунку 1.5).

Тому добре розроблений пакет повинен бути абсолютно нестабільним і конкретним, повністю стабільним і абстрактним або лежати десь дуже близько до основної послідовності.

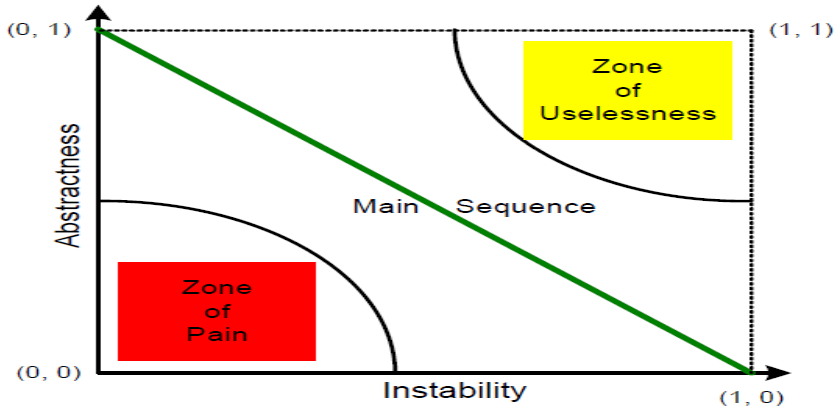


Рисунок 1.5 — Графік A-I із зонами відчуження

Тому бажано, щоб пакети були на головній послідовності або близько до неї, можна визначити метрику, щоб виміряти, наскільки далеко пакет знаходиться від цього ідеалу. Нормалізована відстань D' може бути розрахована як:

$$D' = |A + I - 1| \quad (6.3)$$

D' може коливатися від $[0, 1]$. Значення D' 0 вказує на те, що пакет знаходиться безпосередньо в головній послідовності, тоді як 1 вказує на те, що пакет знаходиться якомога далі від головної послідовності.

2. Опис розроблювального криптографічного пакету

Архітектура програмного забезпечення — це процес проектування глобальної організації програмної системи, включаючи поділ програмного забезпечення на підсистеми, вирішення того, як вони будуть взаємодіяти, і визначення їхніх інтерфейсів [21]. Ми представляємо високорівневий архітектурний огляд КСБМП у формі пакетних структур. Класи у цих пакетах відповідатимуть певним архітектурним стилям і шаблонам проектування. Більшість пакетів можна розглядати як програмні компоненти, а решта — службові модулі. Кожен компонент підтримує інтерфейс, який визначає операції, які повинні бути реалізовані для нього. Між цими компонентами можливі взаємодії через їх інтерфейси. На рисунку 2.1 показано, що КСБМП містить п’ять основних пакетів, а саме: `core`, `crypto`, `io`, `math` і `util`. Весь фреймворк упаковано в кореневий пакет `linca`. Пакет програми можна розглядати як будь-який фрагмент програмного забезпечення, що використовує КСБМП.

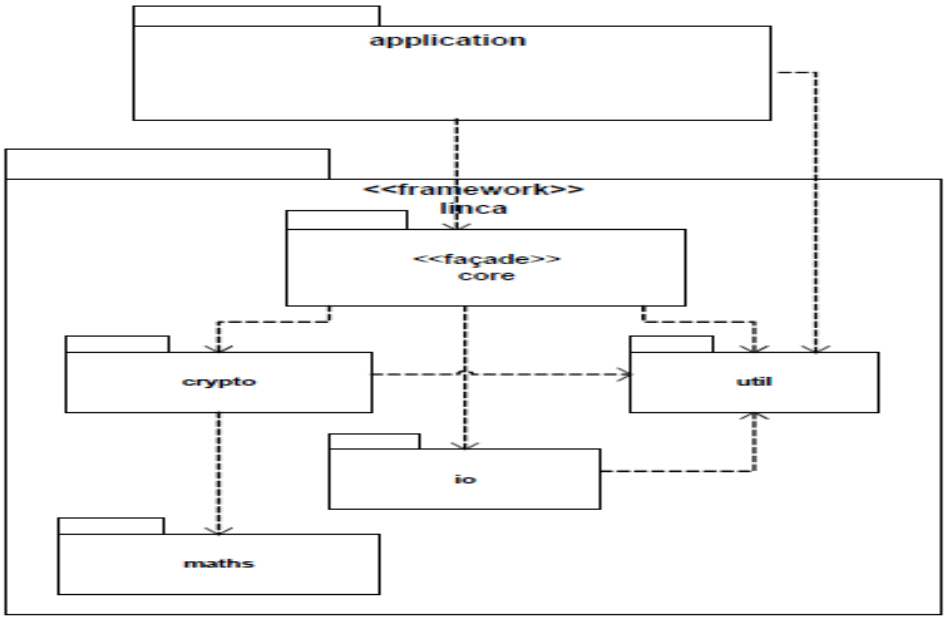


Рисунок 2.1 — Структура фреймворку КСБМП

2.1 Архітектурні моделі

Архітектурний шаблон дозволяє проектувати гнучкі системи з використанням незалежних один від одного компонентів [21]. Деякі компоненти КСБМП організовані в стилі об’єктно-орієнтованої та

багаторівневої архітектури. Об’єктно-орієнтована організація (рисунок 2.2) інкапсулює дані та їхні примітивні операції всередині абстрактного типу даних об’єкта, який взаємодіє з іншими об’єктами через виклики функцій або методів. Таким чином, об’єктно-орієнтовані системи можуть приховувати реалізацію об’єкта, щоб будь-які зміни, внесені до об’єкта, не впливали на користувача.

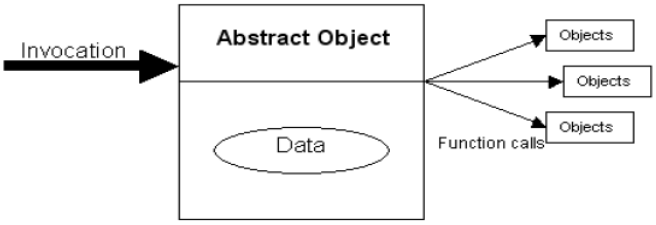


Рисунок 2.2 — Об’єктно-орієнтована організація

Компоненти поділено на рівні, де дані та/або послуги, що надаються на одному рівні, доступні лише для рівнів, що вище (рисунок 2.3). У разі заміни компонента під іншим компонентом це не вплине на верхній компонент.

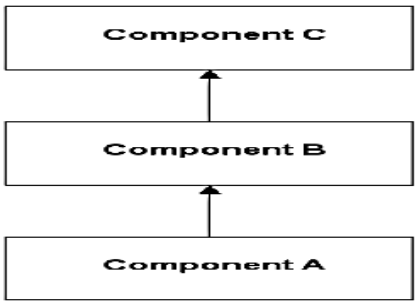


Рисунок 2.3 — Компонентна система

2.2 Патерни проектування

Патерн — це схема багаторазового рішення загальної проблеми, що виникає в конкретному контексті, а шаблон проектування — це шаблон, який корисний у розробці програмного забезпечення [21]. Шаблони проектування застосовуються в деяких компонентах, які забезпечують гнучкий дизайн для розгортання схеми шифрування, застосування ефективної криптографії та створення мережевого з’єднання. Використовуються три основні шаблони проектування: шаблони проектування Factory, Facade і Delegation.

Патерн проектування Factory дозволяє розробнику додати новий специфічний клас `ApplSpecificClass` до системи, яка містить багаторазово використовувану структуру, яка створює об'єкти як частину своєї роботи. Шаблон Factory дозволяє фреймворку створити екземпляр класу, не змінюючи фреймворк. Фреймворк делегує створення екземплярів `ApplSpecificClass` спеціалізованому класу ApplSpecificFactory. ApplSpecificFactory, який реалізує загальний інтерфейс Factory. Factory оголошує метод, метою якого є створення деякого підкласу `ApplSpecificClass` класу під назвою `GenericClass` (рисунок 2.4).

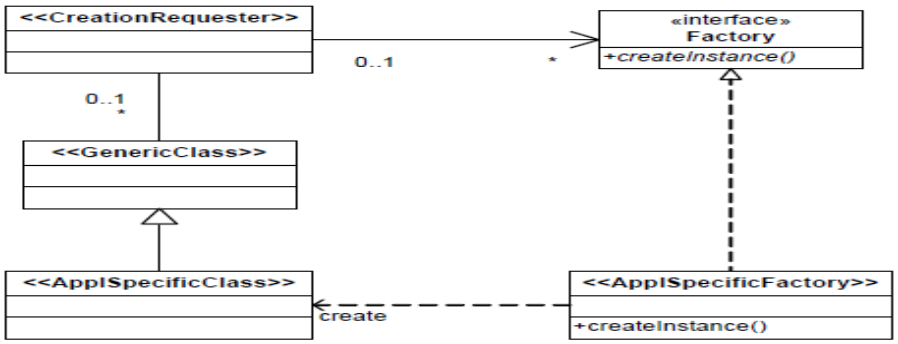


Рисунок 2.4 — Застосування шаблону проектування Factory

Шаблон делегування дозволяє одному класу отримувати доступ до методів іншого класу без використання успадкування. Зазвичай для використання делегування має існувати асоціація між Delegator і Delegate. Цей зв'язок може бути двонаправленим або однонаправленим від Delegator до Delegate. Однак інколи може бути доречним створити нову асоціацію лише для того, щоб можна було використовувати делегування – за умови, що це не збільшує загальну складність системи. На рисунку 2.5 клас Delegator викликає метод у класі Delegate, який має асоціацію з класом Delegator.

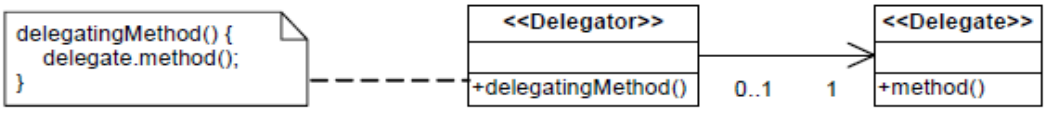


Рисунок 2.5 — Шаблон проектування Delegation

Патерн дизайну Facade спрощує погляд розробників на складний пакет. Клас Facade (рисунок 2.6) міститиме спрощений набір загальнодоступних

методів, щоб більшості інших підсистем не потрібен доступ до інших класів у пакеті. Кінцевим результатом є те, що пакет в цілому легше використовувати та має меншу кількість залежностей від інших пакетів.

Рисунок 2.6 — Шаблон проектування Facade

2.3 Позначення та визначення пакетів

Нотації UML використовуються для ілюстрації структурування пакетів у КСБМП. Щоб пояснити взаємозв'язок між цими пакетами, використовуються дві різні нотації пакетів рисунок 2.7.

Рисунок 2.7 — Нотація пакетів

Нотація компонента UML не підходить для архітектурних описів, оскільки вона моделює сутності рівня реалізації (виконувані файли, модулі вихідного коду). Підсистема UML розглядається як одиниця зі специфікацією, реалізацією та ідентичністю [37]. Тому пакет підсистеми можна розглядати як програмний компонент до його розгортання. На рисунку 2.7 ***нотація А*** позначає компонент, на якому зосереджується увага в термінах залежностей від інших компонентів, позначених ***нотацією В***. Назви пакетів часто використовуються лише замість них, коли описують їхній зв'язок один з одним для простоти. Наприклад, коли ми описуємо певний зв'язок між пакетом A і пакетом B, ми пишемо: «A вимагає B для генерації значення».

2.4 Пакет Crypto

Пакет `crypto` містить усі криптографічні примітиви, рекомендовані в розділі 1. Вони згруповані у чотири функції, а саме: автентифікація

повідомлень, криптосистеми з симетричним ключем, криптосистема з асиметричним ключем і цифрові підписи. Окремі пакети в `crypto` представлені в таблиці 2.1.

Таблиця 2.1 — Криптографічні примітиви, що використовуються в КСБМП

Назва	Опис
authentication	Складається з односторонньої хеш-функції та алгоритму MAC, який забезпечує автентифікацію повідомлення.
skcipher	Складається з симетричного шифру, який шифрує та розшифровує по одному блоку даних. Розмір блоку визначається специфікацією шифру.
mode	Складається з режиму блочного шифрування, який можна використовувати разом із симетричним шифром, який шифрує та розшифровує принаймні один блок даних.
symmetrickey	Складається з генератора симетричних ключів, який генерує два типи ключів, а саме: ключ на основі пароля (PBK) і ключ сеансу.
akcrypto	Складається з компонента, який визначає схему асиметричного шифрування.
akcipher	Складається з асиметричного шифру, який шифрує та розшифровує один блок даних за раз. Розмір блоку визначається специфікацією шифру.
encoder	Складається з функції кодування, яка відповідає за руйнування математичних структур у певній асиметричній криптосистемі.
asymmetrickey	asymmetrickey складається з компонентів, які дозволяють завантажувати відкриті та приватні ключі з апаратного чи програмного забезпечення, залежно від базової машини, наприклад, внутрішнього сервера або мобільного пристрою.
signature	Містить генератор і верифікатор цифрового підпису на основі відкритих і закритих ключів.

Компоненти всередині `crypto` керують власними внутрішніми даними та забезпечують чітко визначені методи інтерфейсу, які дозволяють взаємодіяти з іншими компонентами.

2.4.1 Автентифікація повідомлення

Хеш-функції не лише забезпечують функціональність для автентифікації повідомлень, їх також можна використовувати під час генерації псевдовипадкових чисел і цифрових підписів. Пакет `authentication` складається з хеш-функції та алгоритму MAC, який забезпечує автентифікацію повідомлення. Хеш-функція та алгоритм MAC мають інтерфейси, які визначають їх реалізацію. Хоча `authentication` є незалежним пакетом у `crypto` (рисунок 2.8), пакети `encoder`, `signature` та `symmetrickey` залежать від односторонньої хеш-функції.

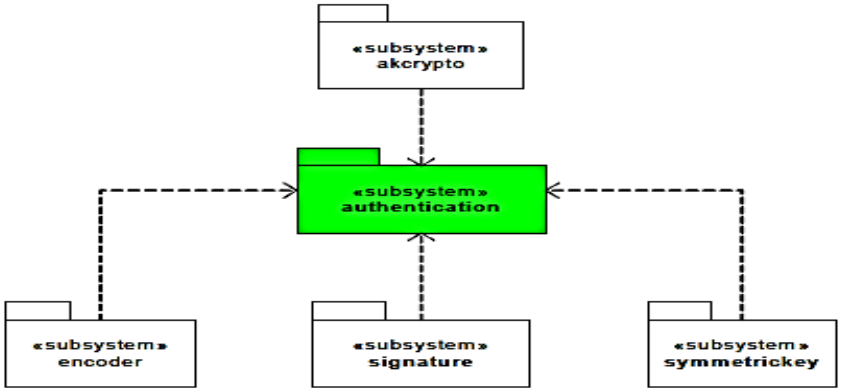


Рисунок 2.8 — Структура пакету authentication

Пакет `akcrypto` не викликає методи `authentication`, а передає екземпляр реалізованої хеш-функції базовій схемі асиметричного шифрування.

2.4.2 Криптосистема з симетричним ключем

Криптосистема з симетричним ключем представлена на рисунку 2.9 у КСБМП складається з: симетричного шифру, режиму блокового шифрування і механізму генерації ключів.

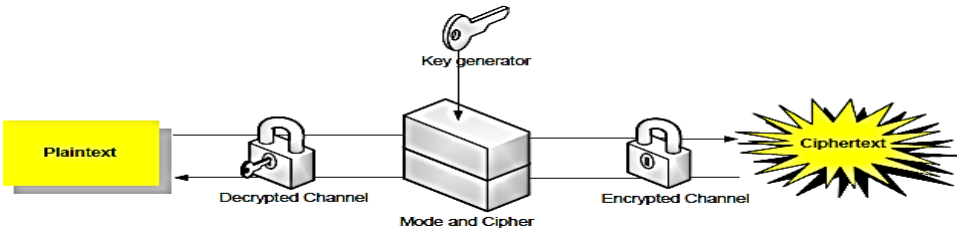


Рисунок 2.9 — Криптосистема з симетричним ключем у КСБМП

На рисунку 2.10 екземпляр симетричного шифру включено в `mode`, щоб увімкнути шифрування та дешифрування принаймні одного блоку даних. Тому виклики методів викликаються в класі інтерфейсу в `mode` замість `skcipher`.

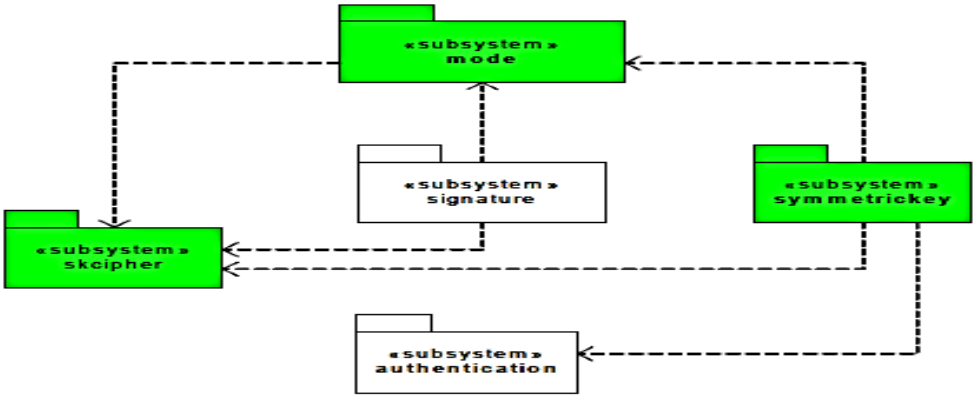


Рисунок 2.10 — Структура пакету симетричного ключа cryptosystem

Для `symmetrickey` потрібен `mode` для генерації або PBK, або сеансового ключа n-біт за допомогою процесу шифрування. Ключ для `mode`, генерується за допомогою односторонньої хеш-функції в `authentication` з використанням випадкових бітів, зібраних із джерела ентропії. Класи всередині `symmetrickey` не викликають жодних методів із `skcipher`, але існує залежність через співпрацю `skcipher` із `mode`. Клас усередині `symmetrickey` маршалюватиме фрагменти даних (параметри генерації ключа), які використовуватимуться для створення секретного ключа.

2.4.3 Криптосистема з асиметричним ключем

Криптосистема з асиметричним ключем (рисунок 2.11) складається з: асиметричного шифру, додаткової функція кодування і механізму завантаження ключів.

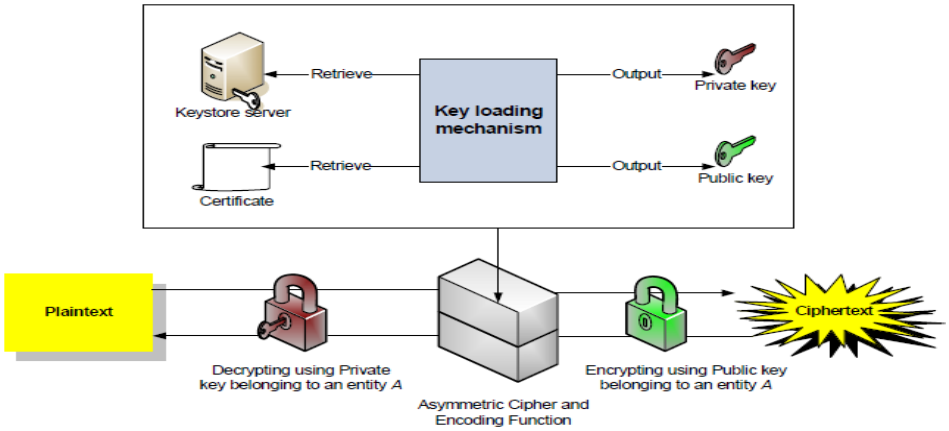


Рисунок 2.11 — Асиметричний ключ cryptosystem в КСБМП

Важливість функції кодування вже пояснюється. Однак не всі асиметричні шифри вимагають функції кодування, наприклад, ElGamal або Elliptic Curve. Мета `akcrypto` — надати КСБМП гнучкість для оновлення до найбезпечнішої доступної схеми асиметричного шифрування без впливу на інші компоненти. Щоб досягти цього, клас реалізації в `akcrypto` слідує шаблону делегування.

На рисунку 2.12 клас `AsymmetricKeyCryptoImpl` делегує виклики методу або схемі шифрування, що використовується з функцією кодування, або просто асиметричному шифру, який є достатньо безпечним для використання

без функції кодування. Таким чином, компоненти всередині `core` не постраждають.

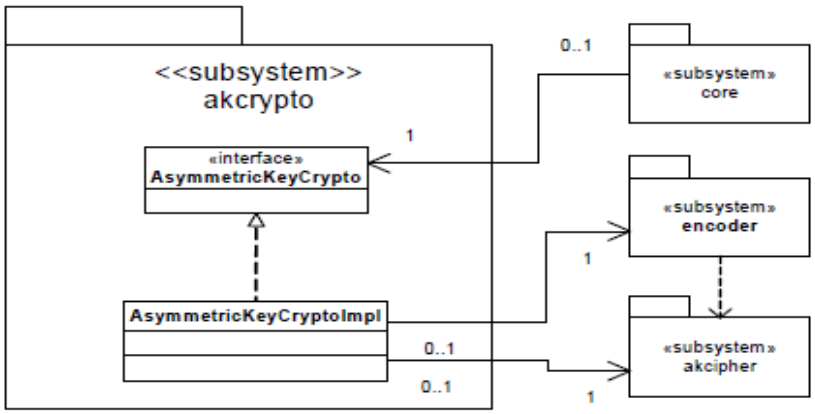


Рисунок 2.12 — Структура завантажувача внутрішньої схеми шифрування

Оскільки `akcrypto` делегує виклики методів базовій схемі шифрування, це здебільшого залежить від компонентів, задіяних в асиметричній криптосистемі. Однак цікаво відзначити, що існує залежність від `authentication`. Відповідно до [35] `encoder` потребує основного асиметричного шифру та односторонньої хеш-функції, щоб функціонувати, і, як наслідок, `encoder` залежатиме від `authentication`.

Пакет `akcipher` містить шифр, який надає послуги асиметричного шифрування та дешифрування. Такі пакети, як `akcrypto`, `encoder` і `signature`, залежатимуть від `akcipher` для виконання своїх відповідних криптографічних функцій. Пакет `akcipher` залежатиме від `asymmetrickey` і `certificate` для завантаження відкритих і приватних ключів із конкретного середовища зберігання.

Механізм асиметричного завантаження ключа запозичений з об'єктно-орієнтованої моделі. Метою механізму завантаження ключів є керування відкритими та закритими ключами. Використовуючи об’єктно-орієнтовану модель, він дає змогу `asymmetrickey` отримувати відкритий і закритий ключі з будь-якого базового типу машини та розташування (рисунок 2.13).

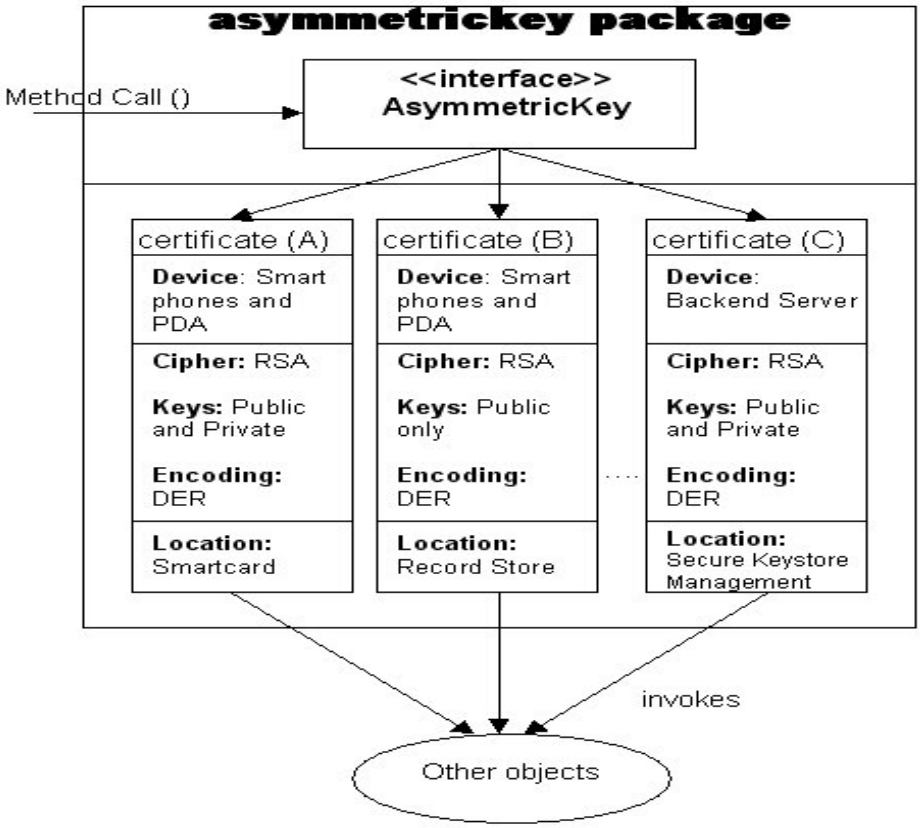


Рисунок 2.13 — Архітектура асиметричного завантаження ключів

КСБМП може бути реалізована як для мобільного пристрою, так і для настільного пристрою, щоб досягти сумісності, коли він використовується для корпоративних мобільних додатків; тому для кожного середовища будуть різні реалізації. Ці різні реалізації повинні зберігати однакову структуру пакета (рисунок 2.14), щоб забезпечити переваги архітектурних принципів.

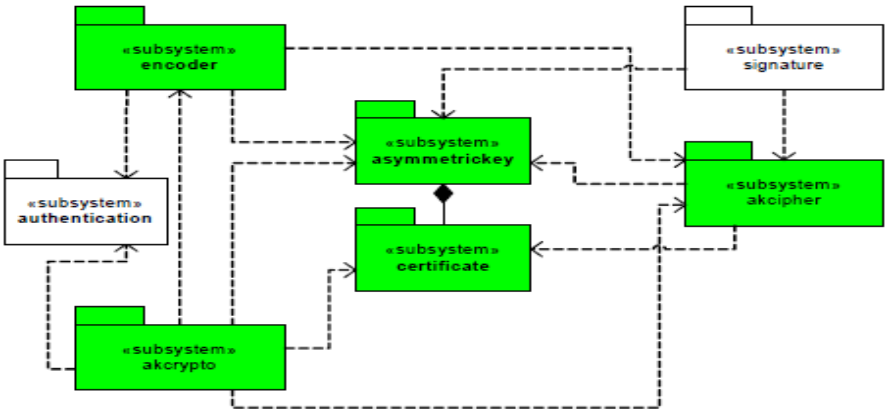


Рисунок 2.14 — Структура пакету cryptosystem асиметричного ключа

Розташування відкритих і закритих ключів може залежати від апаратного чи програмного забезпечення та може бути пов’язане з основною машиною.

Прикладами розташування апаратного забезпечення є смарт-карти або вбудовані мікрочіпи, а програмне забезпечення може бути із захищеної бази даних. Класи завантаження ключів у КСБМП знаходяться в пакеті `certificate`. `Certificate` завантажує потоки сертифіката та приватного ключа з певного місця та аналізує потоки на математичні елементи, які складають відкритий та приватний ключі для конкретного асиметричного шифру. Припустимо, що в майбутньому стандарт X.509 і шифр RSA буде замінено на більш безпечну та ефективну реалізацію, тоді `asymmetrickey` можна буде оновити лише заміною компонентів у `certificate`, не впливаючи на інші компоненти.

2.4.4 Цифрові підписи

Механізм цифрового підпису в КСБМП забезпечується підписом і має включати об'єкт *A* для обчислення сигнатури *SA* з повідомлення *m* та сутність *B* для перевірки *SA* обчислюється сутністю *A*. Пакет `signature` підпису представлено на рисунку 2.15.

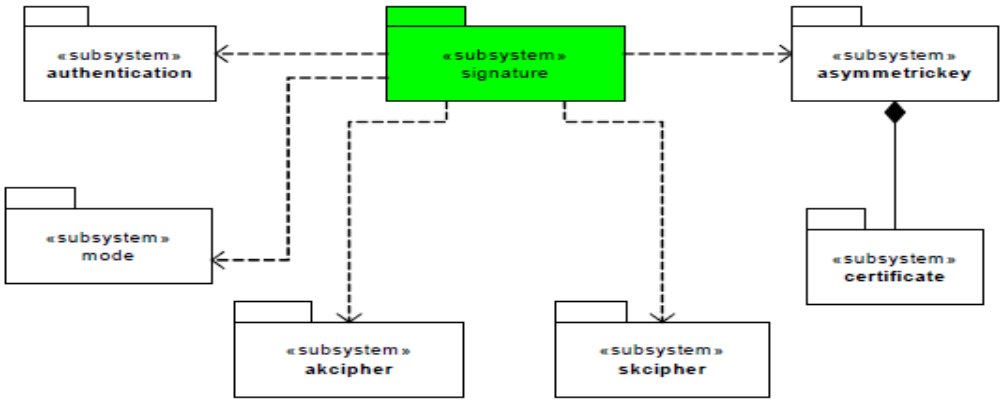


Рисунок 2.15 — Структура пакету signature

Генерація та перевірка цифрового підпису в КСБМП передбачає використання криптографії з асиметричним ключем. Шифрування повідомлення за допомогою закритого ключа створює цифровий підпис повідомлення. Алгоритм цифрового підпису вимагатиме співпраці з `akcipher`, `authentication`, `asymmetrickey` і `mode`. Причина необхідності для `mode` пакету `signature` полягає в тому, щоб зберегти дизайн якомога більш узагальненим. Наприклад, алгоритм генерації цифрового підпису вимагає використання PRNG, а деякі з PRNG можуть вимагати

використання режиму блочного шифрування. Хоча КСБМП не містить компонент PRNG, для `signature` потрібен `mode`, щоб поєднати функції, які пропонує PRNG.

2.5 Математичний модуль

Математичний модуль надає можливість для арифметичних обчислень великих цілих чисел, які використовуються в криптографії з асиметричним ключем. Велике ціле число містить сотні десяткових цифр, які придатні для представлення математичних елементів у криптосистемі асиметричного шифру.

Модуль КСБМП `math` надає засоби для asymmetrickey.certificate і akcipher для виконання великих цілочисельних арифметичних операцій спеціально для шифрів, які підтримуються в рамках проекту (рисунок 2.16). Ці операції включають додавання, віднімання, множення, модуль і ділення великих цілих чисел.

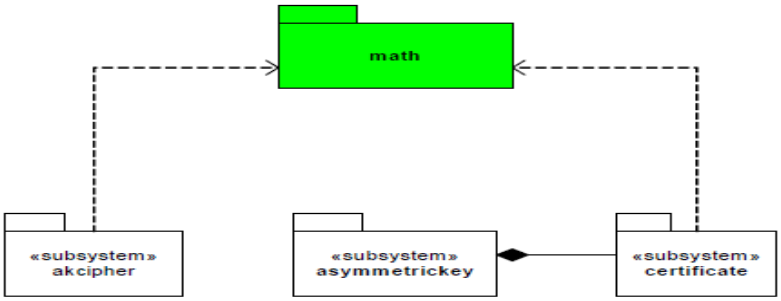


Рисунок 2.16 — Структура пакету math

2.6 Службовий модуль

У пакеті `util` є клас, який відповідає за перетворення integer і long типи даних у масиви байтів і навпаки та символи Unicode у шістнадцятковому представленні. Літеральні значення, такі як числа та рядки, повинні бути перетворені в байтові масиви, перш ніж вони будуть передані в криптографічну функцію. Символи Unicode, які надаються як вихідні дані після шифрування або операції MAC, можуть бути представлені рядком шістнадцяткових значень. Пакет `util` можна використовувати іншими компонентами в КСБМП або в програмі, оскільки він не залежить від інших пакетів (рисунок 2.17).

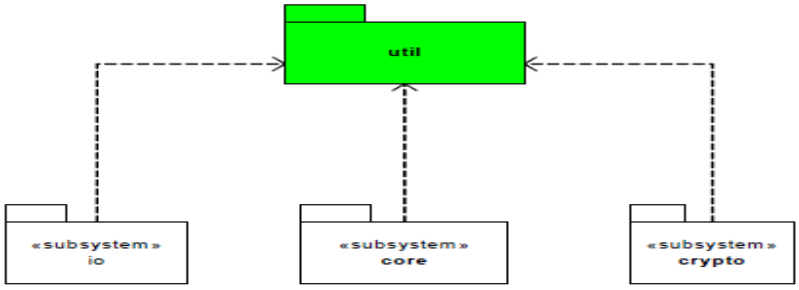


Рисунок 2.17 — Структура пакету math

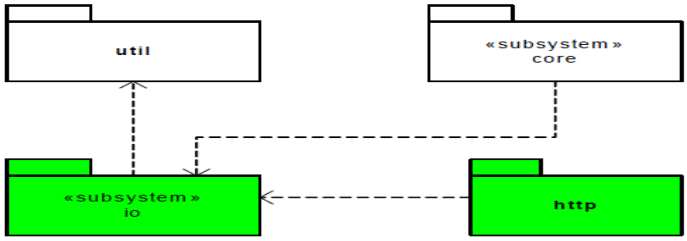
2.7 Компонент підключення КСБМП

Компонент підключення КСБМП (КП КСБМП) — це субфреймворк КСБМП, який надає програмі мережеві функції, коли повідомлення передаються бездротово. Основна перевага КП КСБМП полягає в тому, щоб надати розробникам гнучкість для реалізації належної практики кодування під час використання пакета вводу-виводу, який підтримується різними моделями мобільних телефонів. Одна конкретна вказівка щодо створення мережі проілюстрована наступник кодом. Правильний підхід полягає в тому, щоб зчитувати один байт у буфер.

```
1 // Визначення зв'язків
2 HttpConnection hc = null;
3 DataInputStream dIn = null;
4 byte [] source = byte[512];
5 hc = (HttpConnection) Connector.open(”http://localhost:8080”);
6 dIn = new DataInputStream(hc.openInputStream());
7 // Частина А: Неправильний підхід до читання потоку байтів
8 int bytes = dIn.read(source);
9 dIn.close();
10 hc.close();
11 // Частина Б: правильний підхід до читання потоку байтів
12 int offset = 0;
13 int bytes = 0
14 while(true) {
15   bytes += dIn.read(source, offset, source.length – offset);
16   offset += bytes;
17   if (bytes == -1 || offset >= source.length) break;
18 }
22 dIn.close();
23 hc.close();
```

Іншою перевагою КП КСБМП є відокремлення коду, пов’язаного з мережею, від основного додатка, таким чином сприяючи принципу відокремлення проблем. КП КСБМП не є критично важливим компонентом для

функціональності КСБМП, тому він необов'язковий, оскільки розробники можуть реалізувати власний код для роботи з мережею. Структуру пакету КП КСБМП показано на рисунку 2.19.



Рисунку 2.19 — Структура пакету io

Для кращої розширюваності КП КСБМП дотримується заводського шаблону дизайну, щоб інші типи з'єднань (наприклад, Bluetooth, SMS, FTP тощо) можна було включити, коли виникне у цьому потреба (рисунок 2.20).

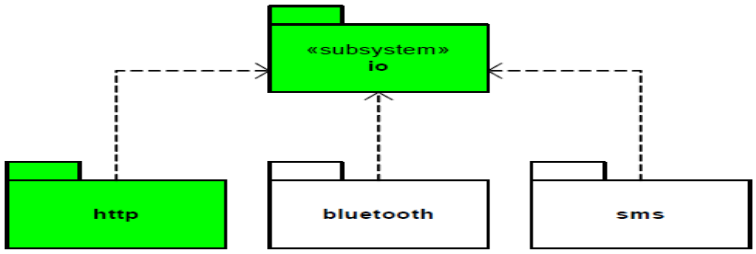


Рисунок 2.20 — Можливість розширення, запропонована КП КСБМП

2.8 Компонент Core

Основний компонент несе відповідальність за спрощення використання криптографічних алгоритмів у пакетах `crypto` та `io`. Це дозволяє реалізувати безпечніший спосіб застосування криптографії, оскільки можна ізолювати непотрібні операції, які можуть послабити застосування криптографії. Компонент `core` розроблено з використанням шаблону проектування Facade. Пакет КСБМП `crypto` можна ефективно інтегрувати у компонент фасаду завдяки логічній організації криптографічних алгоритмів (з точки зору їх функціональності) та простішому визначенню інтерфейсу.

Після реалізації класів у базовому пакеті їх не потрібно замінювати так часто, як класи в `crypto` або `io`. Таким чином розробники можуть зосередитися на підтримці алгоритмів у `crypto` та `io`. Внутрішню структуру компонента `core` показано на рисунку 2.21.

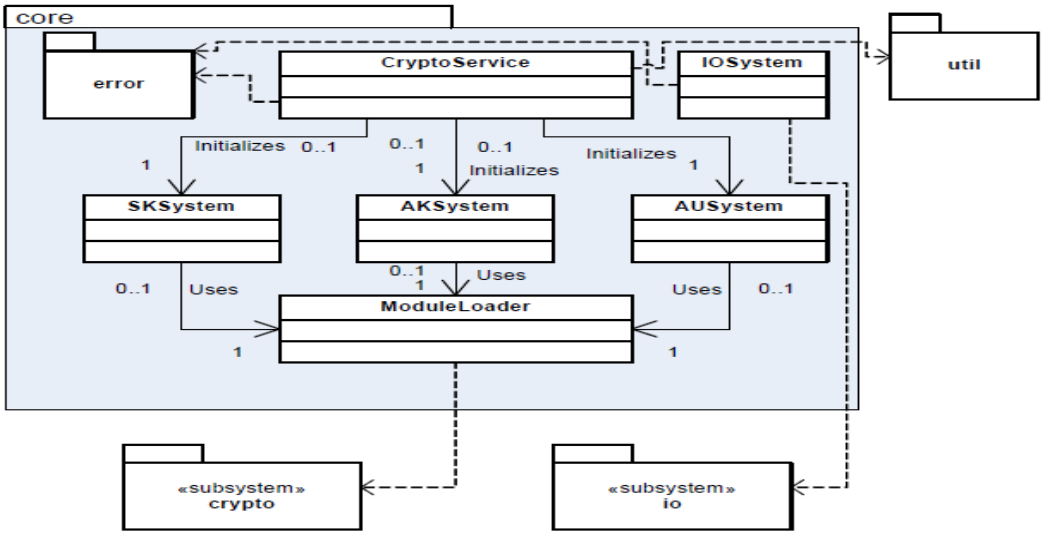


Рисунок 2.21 — Внутрішня структура пакета core

Клас `ModuleLoader` відповідає за створення екземплярів криптографічних примітивів, які будуть використовуватися трьома фасадними класами, а саме: `SKSystem`, `AKSystem` і `AUSystem`. Для простоти ми будемо називати ці три класи криптофасадами. Криптофасади відповідають за:

- Логічне впорядкування криптографічних операцій пов'язаних із з симетричним та асиметричним ключем, автентифікацією повідомлень відповідно.
- Створення криптографічних примітивів, необхідних у криптосистемі.
- Забезпечення методів безпечного використання та ініціалізації криптосистем.

Програма взаємодіє з класами `CryptoService` та `IOSystem`, які функціонують як головні класи фасадів над криптофасадами та пакетом `io`. `CryptoService` несе основну відповідальність за забезпечення безпечного застосування криптосистем КСБМП, а `IOSystem` забезпечує простіше використання пакета io. Клас `CryptoService` має інші функції, де він:

- Спрощує виправлення помилок і обробку винятків. Усі винятки, створені криптофасадами, поширюються в `CryptoService`, а винятки операцій повторно створюються як: `CryptoException` або `MessageNumberException`.
- Забезпечує логічний доступ до криптографічного сервісу. `CryptoService` дозволяє розробникам керувати криптографічними функціями, необхідними для системи. Наприклад, якщо потрібно створити MAC, розробнику потрібно викликати метод для ініціалізації `MDSystem`.

3. Розробка, впровадження безпеки, аналіз дизайну та розготання системи

3.1. Впровадження безпеки у системі КСБМП

Ми розглянемо питання безпеки та впровадження для кожного компонента в КСБМП. Для ефективного розгортання КСБМП її потрібно впроваджувати окремо для мобільного клієнта та настільного сервера, що зумовлено двома різними середовищами, які впливають на керування симетричними та асиметричними ключами. Продемонтуємо необхідні відмінності для різних реалізацій. Інтерфейс для обох реалізацій залишається незмінним.

3.1.1 Вибір алгоритму та розмір ключа

Алгоритми в КСБМП мають підвищувати безпеку та ефективність системи, причому безпеці надається вищий пріоритет, ніж ефективності. Таблиця 3.1 ілюструє симетричні, асиметричні алгоритми та алгоритми MAC, які підтримуються в КСБМП. Ці алгоритми базуються на рекомендаціях, представлених у розділі 1.

Таблиця 3.1. Криптографічні алгоритми, які використовуються в КСБМП

Алгоритм	**Розмір ключа**	**Опис**
AES_CTR (Rijndael)	256-bit	Криптографія з симетричним ключем з використанням шифру AES (Rijndael) і режиму лічильника
RSAES-OAEP	2048-bit	Криптосистема RSA використовує схему кодування OAEP
HMAC_SHA-25	256-bit секретне значення	Автентифікація повідомлень за допомогою HMAC і базової хеш-функції SHA-256

3.1.2 Аналіз продуктивності

Ми провели аналіз продуктивності, порівнюючи час і швидкість шифрування та дешифрування з використанням різних розмірів ключів на конкретній моделі телефону. Спосіб проведення аналізу продуктивності полягає в порівнянні тривалості шифрування/дешифрування та швидкості шифрування алгоритмів шифрування. Використовуються такі шифри:

- AES128: простий шифр AES для 128-бітного ключа.
- AES256: простий шифр AES для 256-бітного ключа.
- AES256_CTR: режим лічильника з шифром AES для 256-бітного ключа.

• RSAES-OAEP_2048: шифр RSA зі схемою OAEP для 2048-бітного ключа.

Усі шифри реалізовані на Java. AES реалізовано відповідно до оптимізації в [8], де загалом 2 Кбайт статичних таблиць використовуються для циклічного попереднього обчислення. Шифр RSA реалізований за допомогою CRT.

Тривалість шифрування/дешифрування вимірюється часом, витраченим на шифрування/дешифрування 512 КБ даних для AES і 191 байт для RSAES-OAEP протягом у середньому трьох спроб. AES128 і AES256 шифрують і розшифровують блоки розміром до 16 байт, поки не буде досягнуто 512 КБ. Швидкість шифрування вимірюється шляхом ділення всіх зашифрованих даних у кілобайтах на час, витрачений на шифрування/дешифрування. Їх результати представлено в таблиці 3.2.

Таблиця 3.2. Аналіз продуктивності шифрів шифрування

Алгоритм/ розмір ключа	**Тривалість шифрування, мс**	**Тривалість дешифрування, мс**	**Швидкість шифрування, КБ/с**
AES128	3203	3062	200КБ/с
AES256	4214	4129	121КБ/с
AES256_CTR	5104	4791	100КБ/с
RSAES-OAEP-2048	693	5411	0,36КБ/с

3.1.3 Режим блочного шифрування

Режим CTR вимагає, щоб лічильник був унікальним. Щоб забезпечити цю властивість, КСБМП дотримується концепції негенерованого IV. Режим блочного шифру є компонентом КСБМП, тому створення лічильника має здійснюватися в пакеті `mode`, щоб не створювати жодних залежностей. Ми рекомендуємо, щоб лічильник, який використовується в КСБМП, складався з таких елементів: 8 байт додаткової інформації, 4 байти, виділені для номера повідомлення, 4 байти, виділені для номера блоку.

Додатковою інформацією може бути системний час або просте випадкове число, яка формує параметр режиму. Її метою є забезпечення унікальності одноразового номеру у всій системі шифрування. Номер повідомлення, гарантує, що лічильник використовується один раз для кожного повідомлення між двома сторонами. Зазвичай номер повідомлення починається з 0, однак [21] припускає, що він повинен починатися з 1, щоб уникнути додаткових реалізацій коду для підтримки стану повідомлення. Весь лічильник шифрується в AES за допомогою

256-бітного ключа, перш ніж обробляється XOR з відкритим текстом, що гарантує знищення будь-яких структур в одноразовому номері.

3.1.4 Керування ключами

Засіб генерації ключів повинен мати можливість генерувати ключ, що складається з n-бітових випадкових байтів. Причиною того, що значення не є фіксованим, є адаптивність. Наразі рекомендований розмір ключа становить 256 біт (32 байти). Основними вимогами до симетричний генератор ключів є:

• Генератор повинен мати можливість генерувати криптографічно стійкий псевдовипадковий бітовий рядок без обмеження на вихідну довжину, за умови, що він буде достатнім для генерації принаймні 256-бітного ключа.

• Ключ можна використовувати для шифрування каналу зв'язку або даних на пристрої.

• Якщо від генератора вимагається використання криптографічних алгоритмів для його реалізації, то це будуть ті, що підтримуються в КСБМП.

• Вихідні дані згенерованого ключа не повинні надаватися пакету `core`, щоб фактичний ключ не обмінювався через мережу, оскільки це погіршить надійність системи безпеки [10].

• Генератор повинен бути ефективним і простим у впровадженні.

Генератор ключів КСБМП розділений на дві частини, а саме функцію накопичувача та функцію генерації ключів. Внутрішню структуру симетричного компонента генерації ключів показано на рисунок 3.1.

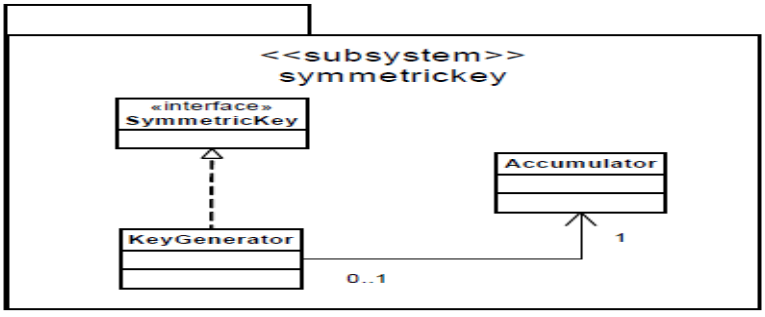


Рисунок 3.1. Генератор симетричних ключів

На мобільних пристроях відсутні справжні випадкові джерела, але є допустимі: мікрофон, розмір доступної пам'яті, системний час. Для вимірювання випадковості ми використали інструмент під назвою ENT, написаний Джоном Вокером [46]. Проведені цим інструментом типи тестів є наступні:

• **Ентропія та оптимальне стиснення**: інформаційна щільність вмісту файлу, виражена кількістю бітів на символ. Значення ентропії має бути близьке до 8 біт. Оптимальне стиснення – показник щільності інформації, який вказує, наскільки файл, що містить послідовність, може бути стиснутий. Для випадкового файлу відсоток стиснення має бути якомога нижчим.

• **Розподіл хі-квадрат**: розподіл хі-квадрат обчислюється для потоку байтів у файлі та виражається як абсолютне число та відсоток, що вказує на те, як часто справді випадкова послідовність перевищуватиме обчислене значення. Кнут [19] рекомендує вважати що значення 10% – 95% є прийнятним.

• **Середнє арифметичне**: це просто результат підсумовування всіх байтів у файлі та ділення на довжину файлу. Дані близькі до випадкових, якщо значення дорівнює 127,5.

• **Значення PI за методом Монте-Карло**: результат алгоритму Монте-Карло для визначення значення PI. Дійсно випадкова послідовність повинна мати високий ступінь точності значення PI.

• **Послідовна кореляція**: послідовна кореляція вимірює ступінь, до якого кожен байт у файлі залежить від попереднього байта. Значення може бути додатним або від'ємним. Значення повино бути близьке до 0 [19].

Ми провели експеримент, щоб визначити випадковість звукового потоку при записі через мікрофон мобільного пристрою. Записаний звуковий потік надсилається на сервер для статистичного аналізу результатів. Формат кодування та параметри якості запису (підтримувані на пристрої) наведені в таблиці 3.3. Якість налаштувань варіюється від 1 (найгірша) до 10 (найкраща).

Таблиця 3.3. Налаштування записів для експеременту

Налаштування	**Формат кодування**	**Частота дискретизації**	**Бітова глибина**	**Канали**
1	PCM	8000	8	1
2	PCM	8000	16	1
3	PCM	8000	16	2
4	PCM	11025	16	1
5	PCM	11025	16	2
6	PCM	16000	8	1
7	PCM	16000	16	1
8	PCM	16000	16	2
9	PCM	22050	16	1
10	PCM	22050	16	2

Ми записали 10 зразків звукового потоку (по 5 секунд кожен) для кожного параметра. Потім зразки були проаналізовані за допомогою інструменту ENT. Параметр 1 (найнижча якість запису) дав найкращий статистичний результат. Його статистичний аналіз має наступні результати:

- Ентропія = 7,813719 біт на байт.
- Оптимальне стиснення зменшило б розмір 1024-байтного файлу на 2 %.
- Розподіл хі-квадрат для 1024 вибірок дорівнює 245,5, і випадковим чином перевищить це значення в 50,00 відсотках випадків.
- Середнє арифметичне значення байтів даних — 128,8633.
- Значення PI за методом Монте-Карло — 3,105882353 (похибка 1,14 %).
- Серійний коефіцієнт кореляції — 0,057228 (некорельований = 0,0).

Клас `Accumulator` (рисунок 3.1) збирає випадкові дані (системний час, обсяг доступної пам'яті та звукові потоки). Джерела хешуються за допомогою SHA-256 для створення 32-байтового хешу (рисунок 3.2). Послідовність хешування джерел виконується випадковим чином. Щоб обмежити доступ до методу, який реалізує збір ентропій із випадкових джерел, у сигнатурі методу слід використовувати модифікатор protected synchronized, щоб клас `KeyGenerator` був єдиним класом, який може його викликати. Модифікатор synchronized забезпечує потокобезпечну властивість.

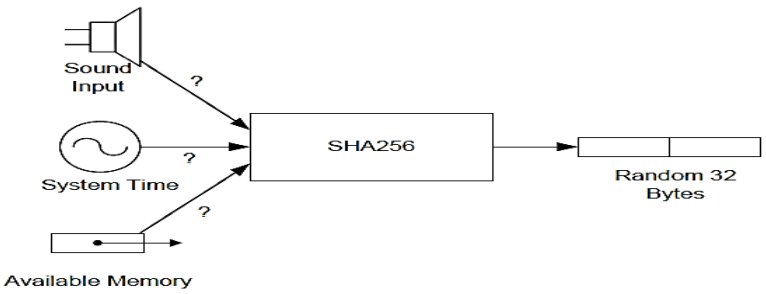


Рисунок 3.2. Вихід акумулятора

Клас `KeyGenerator` може використовувати випадкове джерело, зібране `Accumulator`. Генератор симетричних ключів обробляє генерацію ключів PBE та ключів сеансу. Клас `KeyGenerator` містить два методи для забезпечення цього: stretch() і generateKey(). Нижче представлено їх псевдокод.

```
void stretch
  input: byte [] b // Хешовані випадкові джерела з Accumulator
      StringBuffer t // Дані аутентифікації користувача
```

```
    integer q // Кількість ітерацій хешу
    output: byte [] s // Початкове число, яке використовується для створення блоків шифру
{ //початок void
  s0 ← 0 // переконайтеся, що s0 порожній
  r ← 0 // r — глобальна змінна цілого типу
  if q > 0 {
    r ← q
    for i = 1….r {
      si ← SHA-256(si-1 || t || b)
    }
    return sr
  else {
    do {
      si ← SHA-256(si-1 || t || b)
      r ← r + 1
    } until (1000 milliseconds) // тривалість розтягування
    return sr
  }
} //кінець void
```

Метою методу stretch() є підвищення безпеки пароля або фрази з обмеженою ентропією та створення тимчасового ключа для методу generateKey(). Вхід *t* використовується для автентифікації користувача (пароль, парольна фраза, маркер безпеки тощо). Коли використовується *t*, то важливо додати значення salt (*b*), щоб забезпечити збереження випадковості, якщо *t* не має достатньої ентропії. Випадковий *s* може бути згенерований за допомогою *b*, оскільки це свіжий випадковий рядок байтів. Розмір *r* вибрано, що обчислення *s* займає 200-1000 мс на обладнанні користувача [7] і тому значення ніколи не фіксоване. Тому *r* може збільшуватися відповідно до швидкості обчислень. Вихідні дані `stretch` є початковим числом розміром 256 біт, який використовується як тимчасовий ключ у generateKey(), що представлено нижче.

```
// Перед викликом функції переконайтеся в наступному: assert, що генератор засіяний
assert c == null OR c == getModeParam() // де c — параметр режиму
assert s.length >= 32 // початкове число s походить від stretch()
void generateKey
    output: byte [] k // Псевдовипадковий рядок у байтовому масиві розміром 32
{ //посаток void
    z ← є // ініціалізувати z як порожній байтовий масив відкритого тексту розміром 32
    // Застосуйте методи шифрування з симетричним ключем у КСБМП
    installMode(SC, c) // Примірник симетричного шифру позначається SC
    initMode(true, s, r)
    processBytes(z, k)
    return k
} //закінчення void
```

Метою методу generateKey() є генерація псевдовипадкового ключа за допомогою AES256_CTR. Перевага наявності `KeyGenerator` залежно від `mode` в тому, що якщо AES буде замінено безпечнішим стандартом, функція шифрування також буде оновлена. Методи installMode(), initMode() і processBytes() є методами в `mode`, які застосовують шифрування до початкового порожнього масиву байтів *k*. Задля додаткової безпеки вихідні дані *k* ніколи не використовуються в програмі, а лише в пакеті `core`. `KeyGenerator` має реалізувати метод для повернення набору параметрів. Це дасть можливіть повторно згенерувати секретний ключ легетівним об'єктом. Ці параметри називаються параметром генерації ключа та згруповані разом у такому конкретному порядку: лічильник ітераційного хешування *r*; початкове число, згенероване методом `stretch()`; лічильник *c*, який використовується в `KeyGenerator`.

Щоб повторно згенерувати секретний ключ, необхідно мати параметр генерації ключа та дані автентифікації. Слід зберегти той же порядок, щоб обробку цих параметрів можна було стандартизувати.

За замовчуванням КСБМП отримує відкритий ключ із цифрового сертифіката X.509 версії 3. Формат сертифіката є двійковим і базується на нотації ASN.1. Двійкове кодування сертифіката визначається за допомогою правил відмінного кодування (DER), які базуються на основних правилах кодування (BER). І BER, і DER забезпечують незалежний від платформи метод кодування об'єктів, таких як сертифікати та повідомлення, для передачі між пристроями та програмами. Реалізація асиметричного завантажувача ключів у КСБМП залежить від: підтримуваних асиметричних шифрів, структури сертифіката та метод кодування та розташування збереженого сертифіката.

Завантажувач у КСБМП реалізований для завантаження сертифікатів у кодуванні ASN.1 DER із відкритим ключем RSA. Сам шифр RSA реалізовано за допомогою CRT для більшої швидкості обробки. Завантажувач асиметричних ключів складається з основного інтерфейсу `AsymmetricKey`, реалізованого `KeyLoader`, який містить код, що дозволяє завантажувати пари асиметричних ключів. Класи `RSAPrivateKeyImpl` і `RSAPublicKeyImpl` містять математичні конструкції пар ключів RSA і повертаються `KeyLoader`.

`Other_Classes` позначає класи, необхідні `KeyLoader`, які дозволяють завантажувати пари ключів із певних місць (рисунок 3.4).

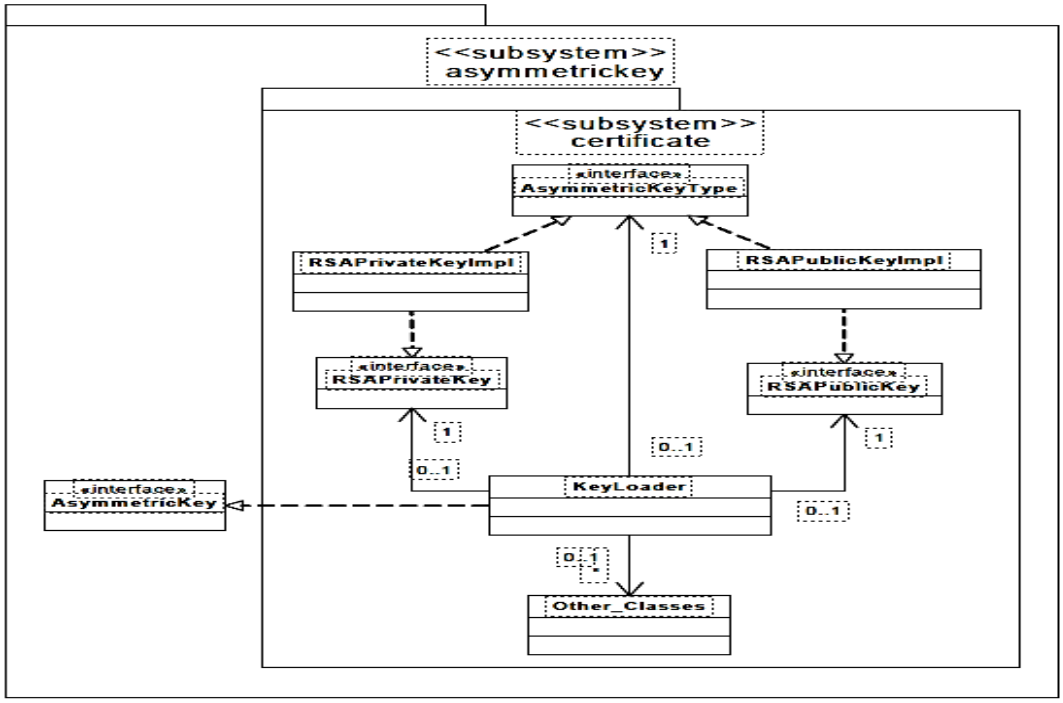


Рисунок 3.4. Завантажувач асиметричних ключів

Завантаження пар ключів із серверної машини суттєво відрізняється від портативного пристрою, оскільки місце зберігання ключів може відрізнятися. Вибір місця залежить від політики безпеки. КСБМП підтримує протоколи для завантаження пар ключів на мобільний пристрій:

1. **File**: Відкритий ключ зберігається в сертифікаті X.509 як файл формату DER. Відповідний закритий ключ відповідає ASN.1 і зашифрований як двійковий файл (рисунок 3.5).

2. **APDU**: Мобільні програми можуть обмінюватися даними зі смарт-карткою за допомогою протоколу на основі блоків даних протоколу додатків (APDU). Цей протокол визначено ISO 7816-4 і описано в документації Java Card Development Kit [44].

3. **JCRMI**: Мобільні програми можуть обмінюватися даними зі смарт-карткою за допомогою протоколу, заснованого на Java Card Remote Method Invocation (JCRMI). Для роботи з об’єктом на картці додатки використовують віддалений інтерфейс.

Реалізація завантажувача ключів КСБМП для робочого столу містить додатковий протокол для завантаження пари ключів:

4. **Keystore**: цей протокол дозволяє серверній програмі завантажувати пару ключів із безпечного сховища ключів. Наприклад, таким сховищем ключів може бути засіб Keystore, наданий J2SE.

У таблиці 3.5 наведено протоколи завантаження ключів із прикладами URI для завантаження пари ключів.

Таблиця 3.5. Протокол завантаження ключа з прикладом

Протокол	Формат URI	Приклад
File	**file:**<location>	**file:**/rsacert2048.cer
APDU	**apdu:**<slot>;<target>	**apdu:**0;target=SAT
JCRMI	**jcrmi:**<slot>;<AID>	**jcrmi:**0;AID=A0.0.0.67.4.7.1F.3.2C.3
Keystore (private key)	**keystore:**<location>;<alias>;<storepass>;<keypass>	**keystore:**.keystore;mystor;yI8T4;Se4&1t
Keystore (public key)	**keystore:**<location>;<alias>;<storepass>	**keystore:**.keystore;johnnystor;yI8T4;

Для файлового протоколу приватний ключ має бути зашифрований за допомогою КСБМП перед розгортанням на мобільному пристрої відповідно до формату, зображеного на рисунок 3.5.

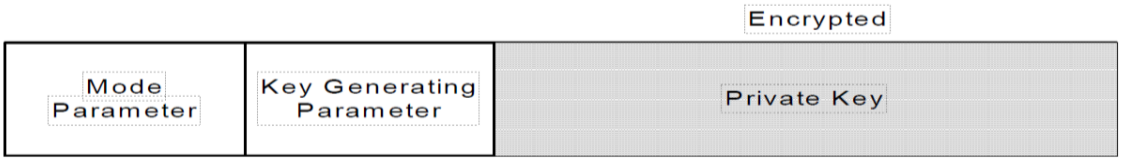


Рисунок 3.5. Зашифрований формат закритого ключа

Під час додавання цих значень до зашифрованого приватного ключа розробник повинен звернути увагу на порядок параметрів режиму та параметрів генерації ключа. Це необхідно для забезпечення правильності розбору полів для дешифрування.

Під час завантаження ключа файлу необхідно, щоб відкритий ключ завантажувався з автентичних джерел, щоб уникнути атак типу "людина посередині". Найефективніший спосіб гарантувати, що сертифікат не буде підроблено, це підписання набору MIDlet, який містить сертифікат, видавцем програми. Таким чином весь пакет перевіряється менеджером додатків Java щодо кореневого сертифіката перед установкою.

Під час роботи APDU <slot> вказує номер слота, куди вставлено смарт-карту. <target> вказує на ідентифікаційний номер картки. <target> може бути або ідентифікатором додатка (AID), або (U)SIM Application Toolkit ((U)SAT). AID унікально ідентифікує програму смарт-картки, використовуючи від 5 до 16 шістнадцяткових байтів, де кожне значення байта розділене символом «.».

Під час роботи JCRMI <slot> використовується точно так само, як і для APDU. Єдина відмінність полягає в тому, що JCRMI підтримує лише AID. Якщо URI є однаковим для доступу до сертифіката та закритого ключа за допомогою APDU або JCRMI, тоді той самий URI для відповідного протоколу можна передати в параметри ініціалізації криптосистеми з асиметричним ключем для завантаження відкритого та закритого ключів [42].

Коли використовується протокол Keystore, формат відповідає стандарту механізму keystore, наданого J2SE. <alias> - це логічне ім'я сховища ключів. <storepass> і <keypass> — це паролі для захисту сховища ключів і закритого ключа в сховищі ключів відповідно.

3.1.5 Цифровий підпис

Алгоритм цифрового підпису в КСБМП має такі критерії:

- Сила алгоритму залежить від асиметричного шифру.
- Алгоритм не запатентований і має бути ефективним у реалізації.
- Згенерований підпис матиме випадкове відображення для кожного повідомлення m.

Існує хороша техніка, реалізована для цифрової перевірки за допомогою шифру RSA [7]. Ми змінили алгоритм відповідно до операцій у КСБМП. Ідея цього алгоритму полягає у використанні псевдовипадкового відображення для розширення $H(m)$ (де H — хеш-функція, а m — повідомлення) до випадкового числа s у діапазоні 0, …, n-1. Сигнатура s тоді обчислюється як $s^{1/e}(mod\ n)$. Цей алгоритм нижче представлено псевдокодом алгоритму генерації підпису за допомогою RSA.

void MsgToRSACipher
input: PubKey *v <n> // Відкритий ключ RSA v з n як модуль*
byte [] *m // Повідомлення, яке потрібно перетворити на значення s*
output: byte [] *s*

```
{ //початок void
    byte [] x ← є // length — це максимальний розмір вхідних даних для RSA
    byte [] z ← є // length — це максимальний розмір вхідних даних для RSA
    byte [] t ← SHA-256(m) // Початковий код із хешем повідомлення
    byte [] c ← є // переконайтеся, що лічильник є порожнім масивом
    Integer k ← ⌊ log2n ⌋ // підлогова функція
    // Створіть випадкове зіставлення повідомлення
    installMode(SC, c) // Примірник симетричного шифру позначається SC
    initMode(true, t, 1) // встановіть номер повідомлення 1 за замовчуванням
    processBytes(z, x) // шифрування з симетричним ключем
    initAsymmetricKeyCipher(true, v) // ініціалізація асиметричного шифру
    s ← processBytes(x) // Операція RSA відображена в x mod 2k
    return s
} // закінчення void

void generateSignature
    nput: PriKey w<n, d, e>// Компоненти закритого ключа RSA
    byte [] m // Повідомлення для підпису
    output: byte [] σ // Підпис для м
{// початок void
    byte [] s ← MsgToRSACipher(n, m)
    initAsymmetricKeyCipher(true, w) // ініціалізація асиметричного шифру
    σ ← processBytes(x) // Операція RSA відображена в s^1/e mod n
    return σ
} // закінчення void

void VerifyRSASignature
    input: PubKey v<n, e> // Відкритий ключ RSA з показником е
    byte [] m // Повідомлення, яке передбачається підписати
    byte [] σ // Підпис на повідомленні
    output: true якщо підпис відповідає false інакше
{// початок void
    s ← MsgToRSANumber(n, m)
    initAsymmetricKeyCipher(true, v) // ініціалізація асиметричного шифру
    byte [] b ← processBytes(σ) // Операція RSA відображається на σ^e mod n
    if(b == s) {return true} else {return false}
} //закінчення void
```

Алгоритм генеруватиме випадкові пари $\left(s, s^{1/e}\right)$ для набору повідомлень m_1, m_2, …, m_i. Це тому, що m хешується за допомогою хеш-функції H, щоб надати ключ для ініціалізації режиму блокового шифрування. Параметр режиму можна ініціалізувати до 0 для простішої реалізації. Процес шифрування забезпечує випадкове зіставлення. Поки H є безпечним, на $H(m)$ можна впливати лише методом проб і помилок. Кожен може створити пари у формі $\left(s, s^{1/e}\right)$ для випадкових значень s, тому це не надає нової інформації, яка допоможе зловмиснику підробити підпис. Однак для будь-якого конкретного повідомлення m тільки хтось (наприклад, *сутність A*), хто знає приватний

ключ, може обчислити відповідну пару $\left(s, s^{1/e}\right)$, тому що s має бути обчислено з $H(m)$, а потім $s^{1/e}$ має бути обчислено з s, для якого потрібен закритий ключ. Таким чином, будь-хто, хто перевіряє підпис, знає, що *сутність A* має підписати його.

3.1.7 Безпека, яку забезпечує базовий пакет

Методи в класі `CryptoService` забезпечують більш ефективний спосіб застосування криптографії на прикладному рівні, запобігаючи доступ розробника до складних інтерфейсів нижчого рівня в пакеті шифрування. Таким чином, розробник отримує доступ до більш простих методів у `CryptoService`, які сприяють більш безпечній і простішій техніці застосування криптографії. Для того, щоб використовувати клас `CryptoService`, можна використовувати статичний метод у `CryptoService` для повернення його екземпляра, який представлений нижче.

```
public static CryptoService getCryptoService(boolean isRemote)
```

Параметр `isRemote` вказує, чи КСБМП застосовується в середовищі мобільної мережі. Один екземпляр `CryptoService` може ініціалізувати симетричні та асиметричні криптосистеми лише один раз. Кожен екземпляр `CryptoService` може керувати своїми секретними та особистими ключами окремо, що робить їх потокобезпечними.

Важко передбачити, яку інформацію використовуватиме розробник, щоб зробити повідомлення однозначним. Розробник повинен надати цю інформацію через параметр `extras` разом із `secret` під час ініціалізації. Вхідні дані для параметра `extras` є універсальними, оскільки він приймає масив Object, що складається з таких типів, як масиви байтів або рядки. Значення безпеки в параметрі `extras` можна використовувати лише один раз інакше створюється `CryptoException`. Щоб змінити значення повідомлення в одному екземплярі, використовується метод resetMessageMeaning(Object []). Розробник може використовувати той самий екземпляр `CryptoService` для створення MAC-адрес для кількох повідомлень. Методи хеш-функції представлені нижче.

public void initMACComponent(**byte**[] secret, **Object**[] extras) **throws** CryptoException{..}
public void generateMAC(**Object**[] data, **byte**[] mac_out) **throws** CryptoException{..}
public void generateMAC(**byte**[] data, **byte**[] mac_out, **int** count) **throws** CryptoException{..}
public void resetMessageMeaning(**Object**[] extras) **throws** CryptoException{..}
public void compareMAC(**Object**[] data, **byte**[] mac_in) **throws** CryptoException{..}
public void compareMAC(**byte**[] data, **byte**[] mac_in) **throws** CryptoException{..}
public int getMACOutputSize() **throws** CryptoException{..}
public String getMACSystemName() **throws** CryptoException{..}

Рекомендована довжина `secret` має становити принаймні 32 байти, однак це може бути довільна довжина, оскільки `secret` хешується 10 разів за допомогою SHA-256, перш ніж значення буде використано функцією HMAC. Функція MAC переважно використовуватиметься в мобільній програмі, оскільки одностороннє хешування застосовується у внутрішніх компонентах КСБМП, які обробляють симетричний ключ і генерацію цифрового підпису. Тому стає непотрібним використовувати односторонні хеш-функції на прикладному рівні.

Криптосистема з симетричним ключем ініціалізується ідентифікацією користувача, параметром генерації ключа та параметром режиму. `Usr_id` — це інформація, яку користувач використовує для автентифікації. Якщо криптосистема з симетричним ключем ініціалізується вперше об'єктом-ініціатором, параметри `key_param` і `mode_param` можуть бути ініціалізовані до null. Це інформує генератор симетричних ключів і компоненти режиму самостійно генерувати необхідні параметри. Параметри генерації ключа та параметри режиму можна отримати за допомогою методів getSymmetricKeyParam(byte[]) і getModeParam(byte[]). Методи криптосистеми з симетричним ключем представлені нижче.

public void initSymmetricKeyCipherComponent(**StringBuffer** usr_id, **byte** [] key_pram, **byte** [] mode_param) **throws** CryptoException{..}
public void symmetricKeyEncryption(**int** msgnum, **byte** [] plaintext, **byte** [] ciphertext) **throws** CryptoException, MessageNumberException{..}
public void symmetricKeyDecryption(**int** msgnum, **byte** [] ciphertext, **byte** [] plaintext) **throws** CryptoException, MessageNumberException{..}
public void getModeParam(**byte** [] mode_param_out) **throws** CryptoException{..}
public void getSymmetricKeyParam(**byte** [] key_param_out) **throws** CryptoException{..}
public void getModeParamSize() **throws** CryptoException{..}
public void getSymmetricKeyParamSize() **throws** CryptoException{..}
public int getSymmetricKeyCipherOutputSize (**int** len, **boolean** forEncryption) **throws** CryptoException{..}
public String getSymmetricKeyCryptosystemName() **throws** CryptoException{..}

Під час шифрування та дешифрування важливо зберігати номер повідомлення унікальним для кожної операції. Номер повідомлення передається в метод initMode(boolean, byte[], int), визначений інтерфейсом `BlockCipherMode` у `mode`. Якщо номер повідомлення повертається до 0 або виходить з послідовності, буде викинуто `MessageNumberException`.

Коли `CryptoService` ініціалізовано, логічне значення передається через `isRemote`. Якщо значення true, то КСБМП застосовано в програмі, яка потребує безпечного зв'язку через мобільну мережу. Якщо значення false, КСБМП застосовується для захисту даних локально на мобільному пристрої. Коли КСБМП використовується для захисту даних локально на пристрої, послідовність повідомлень керується лише під час шифрування. Розробник повинен постійно зберігати відповідні номери повідомлень для кожного зашифрованого повідомлення, параметр генерації ключа та параметр режиму на пристрої, щоб повідомлення можна було розшифрувати на пізнішому етапі. Код для керування номером повідомлення через мережу та локальним керування номером повідомлення на пристрої представлено нижче.

```
// Екземпляр CryptoService, ініціалізований ініціатором
...
StringBuffer pswd = new StringBuffer(auth);
CryptoService cs1 = CryptoService.getCryptoService(true);
cs1.initSymmetricKeyCipherComponent(pswd, null, null);
// ініціатор шифрує два повідомлення для надсилання через мережу
...
cs1.symmetricKeyEncryption(1, plaintext1, ciphertext1);
cs1.symmetricKeyEncryption(2, plaintext2, ciphertext2);
...
cs1.symmetricKeyDecryption(1, ciphertext1, output1); //Illegal!
/* Лише отримувач, який ініціалізував свій CryptoService отриманим ключем і параметром mode, може розшифрувати зашифрований текст*/
……
StringBuffer pswd = new StringBuffer(auth); // одержувач знає пароль
CryptoService cs2 = CryptoService.getCryptoService(true);
cs2.initSymmetricKeyCipherComponent(pswd, keyparam, modeparam); // за допомогою отриманих параметрів ключа та режиму
cs2.symmetricKeyDecryption(1, ciphertext1, output1);
cs2.symmetricKeyDecryption(2, ciphertext2, output2);
...
// якщо одержувач вирішить надіслати зашифроване повідомлення, воно продовжиться з повідомлення номер 3
cs2.symmetricKeyEncryption(3, plaintext3, ciphertext3);
```

```
// CryptoService ініціалізовано для постійного шифрування даних на пристрої
...
StringBuffer pswd = new StringBuffer(auth);
CryptoService cs1 = CryptoService.getCryptoService(false);
cs1.initSymmetricKeyCipherComponent(pswd, null, null);
// користувач шифрує два повідомлення для постійного зберігання
...
cs1.symmetricKeyEncryption(1, plaintext1, ciphertext1);
cs1.symmetricKeyEncryption(2, plaintext2, ciphertext2);
// на більш пізньому етапі повідомлення витягуються
...
cs1.symmetricKeyDecryption(1, ciphertext1, output1);
cs1.symmetricKeyDecryption(2, ciphertext2, output2);
```

Криптосистему з асиметричним ключем можна ініціалізувати двома способами з допомогою відкритого ключа та пари ключів. Ініціалізація відкритого ключа використовує метод initAsymmetricKeyCipherComponent(String, boolean), який приймає розташування сертифіката та логічну умову, щоб вказати, чи слід використовувати відкритий ключ у завантаженому сертифікаті для перевірки цифрових підписів. Таким чином, якщо для параметра `processSignature` встановлено значення true, він автоматично ініціалізує компоненти обробки підпису в КСБМП. Формат розташування сертифіката та розташування закритого ключа залежить від протоколу завантаження ключа (таблиця 3.5).

Ініціалізація пари ключів має додаткові параметри, які вимагають введення місцезнаходження закритого ключа, а також ідентифікаційного пароля, необхідного для розшифровки закритого ключа. Цей метод корисний, якщо завантажувач може отримати доступ до закритого ключа на SIM-карті мобільного клієнта. Номер повідомлення універсально збільшується під час шифрування та дешифрування повідомлень між криптосистемами з асиметричним і симетричним ключами. Зашифрований/розшифрований результат повертається як посилання на масив байтів замість копії через складність попереднього визначення розміру виводу. Методи криптосистеми з асиметричним ключем представлені нижче.

public void initAsymmetricKeyCipherComponent(**String** cert_loc, **boolean** processSignature) **throws** CryptoException{..}

public void initAsymmetricKeyCipherComponent(**String** cert_loc, **String** prikey_loc, **StringBuffer** usr_id,

boolean processSignature) **throws** CryptoException {..}

public [] **byte** asymmetricKeyEncryption(**int** msgnum, **byte** [] plaintext) **throws** CryptoException, MessageNumberException{..}

public [] **byte** asymmetricKeyDecryption(**int** msgnum, **byte** [] ciphertext) **throws** CryptoException, MessageNumberException{..}
public int getAsymmetricKeyInputSize(**boolean** forEncryption) **throws** CryptoException{..}
public String getAsymmetricKeyCryptosystemName() **throws** CryptoException{..}
public String getCertificateInfo(**int** item) **throws** CryptoException{..}

Структуру сертифіката можна надрукувати за допомогою getCertificateInfo(int). Для сертифіката X.509 значення, яке повертається, є версією розпізнаного імені (DN) із сертифіката для друку. DN X.509 — це набір атрибутів, де кожен атрибут є послідовністю ідентифікатора об'єкта та значення. Порядок інформації [42], у якій друкується сертифікат, відповідає формату X.509.

Найфундаментальнішим аспектом роботи з парами асиметричних ключів у КСБМП є можливість розробнику вибирати, які пари використовуватимуться для обробки цифрового підпису чи криптографії з асиметричним ключем, або для обох. Методи обробки підпису представлені нижче.

public void initSignerComponent(**String** cert_loc) **throws** CryptoException{..}
public void initSignerComponent(**String** cert_loc, **String** prikey_loc, **StringBuffer** usr_id) **throws** CryptoException{..}
public void generateSignature(**byte** [] data, **byte** [] signature_out) **throws** CryptoException{..}
public boolean verifySignature(**byte** [] data, **byte** [] signature_in) **throws** CryptoException{..}
public int getSignatureOutputSize() **throws** CryptoException{..}

Якщо `processSignature` має значення false під час ініціалізації криптосистеми з асиметричним ключем, тоді асиметричні ключі використовуються лише для шифрування та/або дешифрування. Виняток `CryptoException` буде створено, якщо криптосистему з асиметричним ключем і компонент підписувача ініціалізовано для використання пар ключів. Формат протоколу завантаження ключа точно такий же, як в таблиці 3.5.

Класи всередині `core` дозволяють розробнику легше використовувати криптокомпоненти, і якщо метод використовується неправильно, викидаються винятки `CryptoSystemException` або `MessageNumberException`. Подальша перевірка параметрів компонентами `crypto` та `io` забезпечує більш надійне відстеження помилок, коли їхні винятки поширюються на

CryptoService. Натспуний фрагмент коду описує формат повідомлення про помилку, яке видає система КСБМП.

Error at <Class Name><Method Name>: <Error Message>

Компоненти в класах crypto та io відповідають за перевірку вхідних параметрів відповідно до визначених ними методів інтерфейсу. Наприклад, компонент у режимі mode виконає перевірку довжини параметрів режиму під час ініціалізації. Винятки, викликані компонентами crypto та io: InvalidArgumentException, RuntimeException і IOException. Ці винятки повторно створюються та перехоплюються в CryptoService, які в кінцевому підсумку створюються як CryptoException.

3.1.8 Мережа

Клас IOSystem у пакеті core пропонує симплексний мережевий засіб, який пропонує більш універсальний спосіб створення мережевих з'єднань у середовищі мобільних програм. Нове мережеве підключення можна створити, викликавши метод newIOConnection(String string). URI має форму типу протоколу, який пропонує поточна реалізація пакета io. Назви протоколів можна запитати за допомогою методу getProtocolTypes(). URI протоколу залежить від того, що підтримується платформою чи операційною системою. Методи класу IOSystem представлені нижче.

public void newIOConnection(**String** URI, **String** protocolType) **throws** CryptoException{..}
public void send(**int** msgnum, byte [] message **StringBuffer** usr_id) **throws** IOException{..}
public [] **byte** receive() **throws** CryptoException{..}
public void closeIOConnection() **throws** CryptoException{..}
public String getProtocolTypes() **throws** CryptoException{..}

Коли створюється зашифроване повідомлення, воно надсилається за допомогою методу send(int, message) через вибраний протокол зв'язку. Метод send() повинен увімкнути надсилання номера повідомлення, довжини повідомлення та зашифрованого повідомлення у трьох послідовних потоках байтів через протокол зв'язку. Метод receive() повинен виконувати перевірку номерів і довжини отриманих повідомлень. Якщо превірка пройдена, то номер

повідомлення та зашифроване повідомлення додаються разом і повертаються абоненту. Одержувач може виконувати подальшу обробку повернутого повідомлення, порівнюючи номер отриманого повідомлення з очікуваним номером повідомлення в іншому випадку може відхилити отримане повідомлення.

3.2. Аналіз архітектурного дизайну системи КСБМП

На основі описаних критеріїв оцінки в розділі 1.3 дизайну архутетури програмного забезпеченням можна зробити наступні висновки:

Динамічне розширення: Компонент підключення КСБМП може включати кілька типів мережевих підключень. Це стало можливим завдяки загальному інтерфейсу та фабричному шаблону дизайну.

Повторне використання: ми ілюстрували залежність різних криптокомпонентів, наприклад, генератор симетричних ключів повторно використовує `mode` і `skcipher` для генерації випадкових ключів. Оскільки КСБМП є фреймворком, її можна використовувати як програмний компонент.

Гнучкість: КСБМП прагне бути достатньо гнучким, щоб бути адаптованим до потреб розробника та системи. Пакет `akcrypto` дозволяє розробнику адаптуватися до змін, внесених до асиметричної криптосистеми, не впливаючи на інші пакети. Однак розробник не має гнучкості, щоб змінити тип алгоритму або розмір ключа для використання в програмі.

Зв'язок: КСБМП зменшує зв'язок вмісту [21], інкапсулюючи всі змінні екземпляра в компонентах `crypto` як приватні та дозволяючи лише компоненту `core` ініціалізувати їх. Компонент `core` також зменшує зв'язок між компонентами `crypto` та `io` та прикладним рівнем.

Ефективний дизайн: використовуючи метрики, наведені в розділі 1.3, КСБМП продемонструвала хороший результат дизайну, оскільки більшість пакетів розташовані близько до основної послідовності (таблиця 3.8). Пакет `core` має нестабільність 1, оскільки пакет `core` є фасадом над пакетами `crypto` та `io` і залежатиме від них. Він не має жодних абстрактних або інтерфейсних класів, що дозволяє легко змінювати конкретний код у ньому. Пакети `authentication` і `skcipher` мають значення нестабільності 0,

оскільки вони не залежать від жодних пакетів. Пакети `math` і `util` мають нестабільність 1, що є низькою якість дизайну, але в них є класи, які служать утилітами тому така нестабільність прийнятна.

Дизайн КСБМП також порівнюється з дизайном BC і SATSA, і результати метрик представлені в таблицях 3.6 і 3.7 відповідно. Значення в N_C, N_A, C_A, C_E, I, A і D' визначаються інструментом під назвою Structural Analysis for Java [13]. Значення в таблиці 3.7 базуються на SATSA RI 1.0 від Sun і можуть відрізнятися залежно від реалізацій.

Таблиця 3.6. Показники для дизайну Lightweight Bouncy Castle API

Назва пакета	N_C	N_A	C_A	C_E	I	A	D'
org.bouncycastle.crypto	26	15	452	6	0,01	0,58	0,41
org.bouncycastle.crypto.agreement	4	0	0	60	1	0	0
org.bouncycastle.crypto.digests	17	0	21	13	0,38	0	0,62
org.bouncycastle.crypto.encodings	3	0	3	46	0,94	0	0,06
org.bouncycastle.crypto.engines	27	0	8	214	0,96	0	0,04
org.bouncycastle.crypto.generators	19	0	3	201	0,99	0	0,01
org.bouncycastle.crypto.io	4	0	0	16	1	0	0
org.bouncycastle.crypto.macs	8	0	4	74	0,95	0	0,05
org.bouncycastle.crypto.modes	9	0	10	80	0,89	0	0,11
org.bouncycastle.crypto.paddings	8	1	10	33	0,77	0,13	0,1
org.bouncycastle.crypto.params	45	0	296	99	0,25	0	0,75
org.bouncycastle.crypto.signers	8	0	3	168	0,98	0	0,02
org.bouncycastle.asn1	53	4	2389	11	0,005	0,08	0,92
org.bouncycastle.asn1.cmp	4	1	3	32	0,91	0,25	0,16
org.bouncycastle.asn1.cms	26	2	11	406	0,97	0,08	0,05
org.bouncycastle.asn1.cryptopro	7	1	0	74	1	0,14	0,14
org.bouncycastle.asn1.esf	5	2	0	47	1	0,4	0,4
org.bouncycastle.asn1.ess	5	0	0	69	1	0	0
org.bouncycastle.asn1.gnu	1	1	0	2	1	1	1
org.bouncycastle.asn1.icao	3	1	0	33	1	0,33	0,33
org.bouncycastle.asn1.misc	6	1	0	45	1	0,17	0,17
org.bouncycastle.asn1.mozilla	1	0	0	12	1	0	0
org.bouncycastle.asn1.nist	2	1	0	8	1	0,5	0,5
org.bouncycastle.asn1.ocsp	17	1	0	258	1	0,06	0,06
org.bouncycastle.asn1.oiw	2	1	2	15	0,88	0,5	0,38
org.bouncycastle.asn1.pkcs	27	1	8	441	0,98	0,04	0,02
org.bouncycastle.asn1.sec	3	1	2	27	0,93	0,33	0,26
org.bouncycastle.asn1.smime	6	1	0	52	1	0,17	0,17
org.bouncycastle.asn1.teletrust	1	1	0	2	1	1	1
org.bouncycastle.asn1.tsp	5	0	0	116	1	0	0
org.bouncycastle.asn1.util	2	0	0	53	1	0	0
org.bouncycastle.asn1.x9	11	1	8	147	0,95	0,09	0,04
org.bouncycastle.asn1.x509	62	1	141	713	0,83	0,02	0,15
org.bouncycastle.asn1.x509.qualified	8	2	0	89	1	0,25	0,25
org.bouncycastle.bcpg	49	8	23	31	0,57	0,16	0,27

org.bouncycastle.bcpg.attr	1	0	1	1	0,5	0	0,5
org.bouncycastle.bcpg.sig	11	0	15	22	0,59	0	0,41
org.bouncycastle.math.ec	10	1	73	29	0,28	0,1	0,62
org.bouncycastle.util	2	0	1	0	0	0	1
org.bouncycastle.util.encoders	11	2	3	0	0	0,18	0,82
java.io	2	0	9	0	0	0	1
java.math	1	0	247	2	0,008	0	0,99
java.security	1	0	78	3	0,04	0	0,96

Таблиця 3.7. Показники для дизайну Lightweight Bouncy Castle API

Назва пакета	**N_C**	**N_A**	**C_A**	**C_E**	**I**	**A**	**D'**
crypto	5	0	2	22	0,92	0	0,08
javax.crypto.spec	2	0	3	10	0,77	0	0,23
java.security	14	2	20	27	0,57	0,14	0,29
java.security.spec	5	2	7	8	0,53	0,4	0,07
javax.microedition.pki	4	1	2	17	0,89	0,25	0,14
javax.microedition.securityservice	2	0	1	10	0,91	0	0,09
java.rmi	2	1	2	8	0,8	0,5	0,3
javacard.framework	8	0	2	3	0,6	0	0,4
javacard.framework.service	1	0	0	1	1	0	0
javacard.security	1	0	0	1	1	0	0
javax.microedition.apdu	1	1	0	3	1	1	1
javax.microedition.io	19	16	3	69	0,96	0,84	0,8
javax.microedition.jcrmi	3	2	0	10	1	0,67	0,67

Таблиця 3.8. Метрики для дизайну КСБМП Mobile

Назва пакета	**N_C**	**N_A**	**C_A**	**C_E**	**I**	**A**	**D'**
linca.core	6	0	0	46	1	0	0
linca.crypto.akcipher	2	1	13	11	0,46	0,5	0,04
linca.crypto.akcrypto	2	1	4	10	0,71	0,5	0,21
linca.crypto.asymmetrickey	2	2	20	4	0,17	1	0,17
linca.crypto.asymmetrickey.certificate	59	5	6	31	0,84	0,08	0,08
linca.crypto.authentication	4	2	27	0	0	0,5	0,5
linca.crypto.encoder	2	1	6	10	0,63	0,5	0,13
linca.crypto.mode	2	1	16	6	0,27	0,5	0,23
linca.crypto.signature	2	1	4	18	0,82	0,5	0,32
linca.crypto.skcipher	2	1	17	0	0	0,5	0,5
linca.crypto.symmetrickey	3	1	8	17	0,68	0,3	0,02
linca.io	4	2	7	1	0,13	0,5	0,37
linca.io.http	1	0	1	2	0,67	0	0,33
linca.math	1	0	20	0	0	0	1
linca.util	3	0	7	0	0	0	1

BC і SATSA також продемонстрували хороший дизайн, більшість їхніх пакетів лежали близько до основної послідовності. Щоб порівняти дизайни КСБМП, BC і SATSA, ми взяли загальне середнє значення стовпців *D'* з кожної таблиці. Ці результати представлені в таблиці 3.9.

Таблиця 3.9. Порівняння середніх D'

Бібліотека/Фреймворк	Середнє D'
Lightweight Bouncy Castle API	0,34
Secure and Trust Service API	0,31
КСБМП	0,33

Результати з таблиці 3.9 показали, що середнє значення *D'* для всіх трьох API лежить близько до основної послідовності. Однак, оскільки BC і КСБМП містять пакети службових програм, які навряд чи зміняться, ми можемо виключити їх із розрахунку. Тому, якщо ми вилучимо пакет linca.util із дизайну КСБМП та пакети bouncycastle.util і bouncycastle.util.encoders із BC. КСБМП отримає найкращий дизайн загалом. Цей результат проілюстровано в таблиці 3.10.

Таблиця 3.9. Порівняння середніх D' без урахування пакетів службових програм

Бібліотека/Фреймворк	Середнє D'
Lightweight Bouncy Castle API	0,32
Secure and Trust Service API	0,31
КСБМП	0,28

Ремонтопридатність: Розділення незалежних функцій безпеки на різні компоненти дає можливість розподілити роботу з розробки між кількома розробниками. Крім того, алгоритми, які підтримуються в КСБМП, відповідають максимально безпечним стандартам. Обслуговування додатків має стати легшим, оскільки можна додавати нові переваги безпеки, не торкаючись існуючого коду. З тієї ж причини спрощується обслуговування криптографічного алгоритмічного коду.

Передбачте застарівання: КСБМП — це платформа з відкритим вихідним кодом, яку можна повторно використовувати для багатьох типів середовищ. Завдяки зусиллям із відкритим кодом спільнота розробників знайде способи передбачити застарівання.

Можливість заміни: криптокомпоненти КСБМП можна легко замінити безпечнішою реалізацією, не впливаючи на компонент `core`.

Непорушність: оскільки всі деталі криптографічного алгоритму містяться в КСБМП, програмі не потрібно буде вносити жодних змін, щоб врахувати зміну алгоритму.

Зрозумілість: розуміння розробником функціональності шаблону фасаду в пакеті `core`.

Портативність: КСБМП — це фреймворк, тому перенесення його на інші мови програмування не повинно бути проблемою. Наразі КСБМП реалізовано на Java. Таким чином, портативність не викликає занепокоєння для КСБМП на будь-яких мобільних пристроях, незалежно від їх марки чи моделі.

Простота тестування: компоненти пакета `crypto` можна легко протестувати через їхні інтерфейси. Наприклад, компоненти `skcipher`, `akcipher`, `mode` і `authentication` мають інтерфейси, які дозволяють перевіряти ці алгоритми на відповідних тестових векторах.

3.3. Вдосконалення системи безпеки в порівнянні з іншими криптологічними пакетами

Покращення безпеки продемонстровано порівнянням КСБМП з Bouncy Castle API (BC) і Secure and Trust Service API (SATSA).

3.3.1 Створення MAC

BC має реалізацію HMAC, однак інтерфейс не забезпечує автентифікацію додаткових даних у повідомленні. Єдиний спосіб додатково автентифікувати повідомлення за допомогою додаткових даних через виклик метода update і помістити ці дані в цей метод.

SATSA-CRYPTO також не вказує розробнику автентифікувати додаткові дані, і його використання складніше, оскільки розробник має виділити точний розмір буфера для зберігання дайджесту. Це означає, що розробник повинен знати характеристики хеш-функції. Щоб запобігти фальсифікації повідомлень у мережі зв’язку, розробник має використовувати CMS, яка обчислюється дорожче, ніж використання MAC.

КСБМП долає обмеження попередніх API, дозволяючи розробнику вказувати додаткові дані під час ініціалізації MAC (рядок 7).

```
1 // Нами визначений pin
2 byte [] pin = "34265".getBytes();
3 // Повідомлення для MAC
4 byte [] toMAC = “Balance: $345,467,66”.getBytes();
```

```
5 // Ініціалізуйте MAC
6 CryptoSystem cipher = CryptoSystem.getCryptoSystem(false);
7 cipher.initMACComponent(pin, extras);
8 // Ініціалізуйте буфер і створіть MAC
9 byte [] mac_out = new byte[cipher.getMACOutputSize()];
10 cipher.generateMAC(toMAC, mac_out, 1);
11 return mac_out;
```

3.3.2 Криптографія з симетричним ключем

Шифрування з симетричним ключем у BC є дуже універсальним, оскільки пропонує широкий вибір шифрів і режимів блочного шифрування. Це може призвести до прийняття неправильних рішень про те, який розмір ключа та шифр шифрування використовувати, а також критерії їх ініціалізації.

SATSA-CRYPTO API містить лише кілька симетричних шифрів і режимів блочного шифрування [43], і він не підтримує режим CTR. BC і SATSA-CRYPTO мають обмеження, які скеровують розробника щодо безпечного та ефективного застосування шифрів.

Як згадувалося проблема з PRNG у BC полягає в тому, що він заповнюється за допомогою системного часу, а SATSA-CRYPTO не має PRNG. Обидва API покладають на розробника уравляти IV. Через те, що розробник повинен управляти IV, і існує можливість повторного використання цього IV.

BC і SATSA-CRYPTO дозволяють ініціалізувати шифр AES до 256-бітного розміру ключа. Цей байтовий масив може бути від 128 до 256 біт, однак пароль може бути не такої довжини. Для компесації решти байтів пароль може бути доповнений. Фабрика ключів створює лише статичний ключ, що є обмеженням, коли потрібен ключ сеансу для шифрування певного каналу мережі. SATSA-CRYPTO передбачає, що розробник має виділити правильний розмір буфера для зберігання зашифрованого та розшифрованого тексту. Симетричне шифрування та дешифрування в КСБМП представлено нижче.

```
1  // Відкритий текст для шифрування
2  byte[] toEncrypt = “PIN Request: OK”.getBytes();
3  // Припустимо, що користувач ввів пароль
4  StringBuffer pswd = new StringBuffer("de23ISS5");
5  // Ми припускаємо, що ми не працюємо через мережеве з’єднання
6  CryptoSystem cipher = CryptoSystem.getCryptoSystem(false);
7 // Ініціалізація системи симетричного шифру...
8 cipher.initSymmetricKeyCipherComponent(pswd, null, null);
9 // Створіть буфер для зберігання зашифрованого тексту та шифрування
```

```
10 byte [] ciphertext = new
byte[cipher.getSymmetricKeyCipherOutputSize(toEncrypt.length, true)];
11 cipher.symmetricKeyEncryption(1, toEncrypt, ciphertext);
12 // Створіть буфер для зберігання відкритого тексту та виконання дешифрування
13 byte [] plaintext = new byte[cipher.getSymmetricKeyCipherOu
```

Генератор ключів КСБМП не використовується як PRNG, а метод generateKey() викликається лише внутрішнім компонентом КСБМП. Розробнику залишається лише керувати послідовністю повідомлень, що є додатковою підтримкою автентифікації даних. Пароль, введений користувачем, може бути отриманий програмою реєстратора ключів (рядки 4 і 8). Однак це та сама проблема з SATSA-CRYPTO, коли користувач повинен ввести PIN-код для доступу до смарт-картки.

3.3.3 Криптографія з асиметричним ключем

BC має класи, які можуть читати сертифікат X.509 і закритий ключ у кодуванні ASN.1. Сертифікат і закритий ключ зберігаються у вигляді двійкового файлу на мобільному телефоні. BC підтримує широкий спектр асиметричних алгоритмів. Однак розробнику доведеться приймати рішення про те, який розмір ключа та схеми шифрування використовувати. Також розробнику необхідно виконати значну кількість викликів методів, щоб розібрати правильні математичні елементи з файлів сертифіката X.509 і закритого ключа, щоб побудувати пару ключів RSA.

Криптографія з асиметричним ключем SATSA-CRYPTO підходить до шифрування, але немає підтримки для розшифровки за допомогою закритого ключа. Перед шифруванням розробник має закодувати відкритий ключ, який використовується класом `X509EncodedKeySpec`, відповідно до інформації відкритого ключа суб'єкта, визначеної стандартом X.509. На жаль, SATSA-CRYPTO не надає жодних класів для включення такого кодування, тому розробник повинен вдатися до інших методів, визначених поза сферою SATSA.

Процес шифрування та дешифрування з асиметричним ключем за допомогою КСБМП представлений наступним кодом.

```
1  // Припустимо, що користувач надав пароль для розшифровки файлу закритого
ключа
2  StringBuffer keypin = new StringBuffer("jkLw23Bnme");
3  // Ми припускаємо, що ми працюємо через мережеве з'єднання.
```

```
4  CryptoSystem cipher = CryptoSystem.getCryptoSystem(true);
5  // Ініціалізація системи асиметричного шифрування та використання тієї самої пари ключів для підписання
6  cipher.initAsymmetricKeyCipherComponent("file:/rsa2048cert.cer",
7  "file:/privatekeyenc.ser", keypin, true);
8 return cipher.asymmetricKeyEncryption(1, toEncrypt);
// На приймальному кінці процес ініціалізації такий самий, як описано вище
..
n  return cipher.asymmetricKeyDecryption(1, toDecrypt);
```

КСБМП пропонує логічний спосіб застосування криптографії з асиметричним ключем. Розробнику не потрібно хвилюватися про виклики складних методів, пов’язаних із керуванням сертифікатами. КСБМП підтримує мінімальний розмір ключа RSA 2048 біт і може підтримувати сертифікати X.503 версії 3. Сам шифр RSA реалізовано за допомогою CRT для покращення швидкості дешифрування.

3.3.4 Створення та перевірка цифрового підпису

BC підтримує низку алгоритмів цифрового підпису. На жаль, щодо алгоритму PSS є патентні проблеми, тому дуже важко дотримуватися рекомендацій у PKCS#1 v2.1. Виведення пари ключів, що використовується для цифрового підпису, має таку ж складність. Цифровий підпис у SATSA-CRYPTO обробляється через клас `Signature` і `CMSMessageSignatureService`. Цифровий підпис можна зробити лише для текстового рядка за допомогою класу `CMSMessageSignatureService`. Це може бути обмеженням, якщо потрібно підписувати тип даних, відмінний від рядка. Послуга цифрового підпису в SATSA-CRYPTO обмежена лише перевіркою підпису, оскільки генерація підпису не підтримується. У КСБМП створення та перевірка цифрового підпису відбувається наступник чином.

```
1 // Повідомлення для підпису
2 byte [] msg = "Account 455355 transacted".getBytes();
3 // Щоб створити підпис
4 byte [] signature = cipher.generateSignature(msg);
5 // Для перевірки підпису
6 boolean isValid = cipher.verifySignature(msg, signature);
```

КСБМП пропонує простіший спосіб створення та перевірки підписів. Це дозволить розробникам керувати парами ключів, які відрізняються за критеріями використання. Даний код ілюструє використання тієї самої пари

ключів, яка використовується в шифруванні з асиметричним ключем для створення та перевірки цифрового підпису. Метод ініціалізації буде ініціалізовано значенням false, якщо для перевірки підпису використовується інша пара ключів. Таким чином окрему пару ключів, яка використовується для підпису, можна ініціалізувати в методі initSignerComponent, який представлений нижче.

```
1 // явно призначити пари ключів, які використовуються для цифрового підпису
2 cipher.initSignerComponent("file:/signingcert.cer", "file:/sgnprikey.ser", keypin2);
```

3.4. Розгортання продукту

У цьому пункті ми продемонструємо, як КСБМП можна використовувати для захисту даних, що передаються через мережу, і даних на пристрої. Наразі КСБМП розгортається на мобільних телефонах із підтримкою Mobile Information Device Profile (MIDP). Програма Java, написана для телефонів MIDP, називається MIDlet, і файли класів, а також будь-які двійкові файли, пов’язані з програмою, інкапсульовані у файл jar під назвою MIDlet suite.

3.4.1 Розгортання криптографічної системи

Існує два способи розгортання пакетів MIDlet, а саме: однорангове або віддалене розгортання (рисунок 3.6).

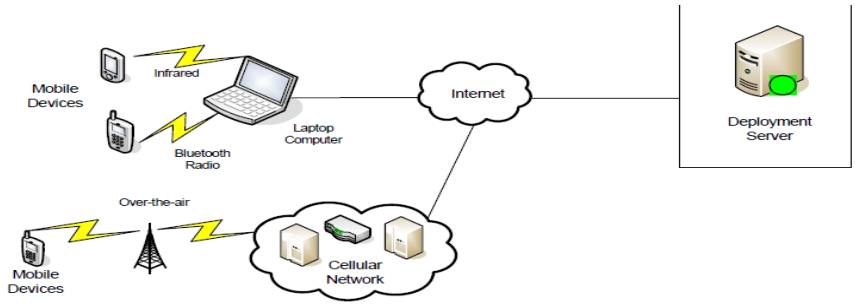


Рисунок 3.6. Різні способи розгортання програми

Однорангове розгортання мобільних програм можна здіснити через Bluetooth, Wi-Fi або кабель підключення, підключивши мобільний пристрій до настільного комп’ютера. Наприклад, комерційний пакет MIDlet можна спочатку завантажити на ноутбук через HTTPS, а потім передати на мобільний пристрій через Bluetooth. Віддалене розгортання можна здійснити через сервер,

на якому розміщено пакет MIDlet. Цей сервер діє як бездротовий портал, який дозволяє клієнтам завантажувати програму бездротовим способом (OTA).

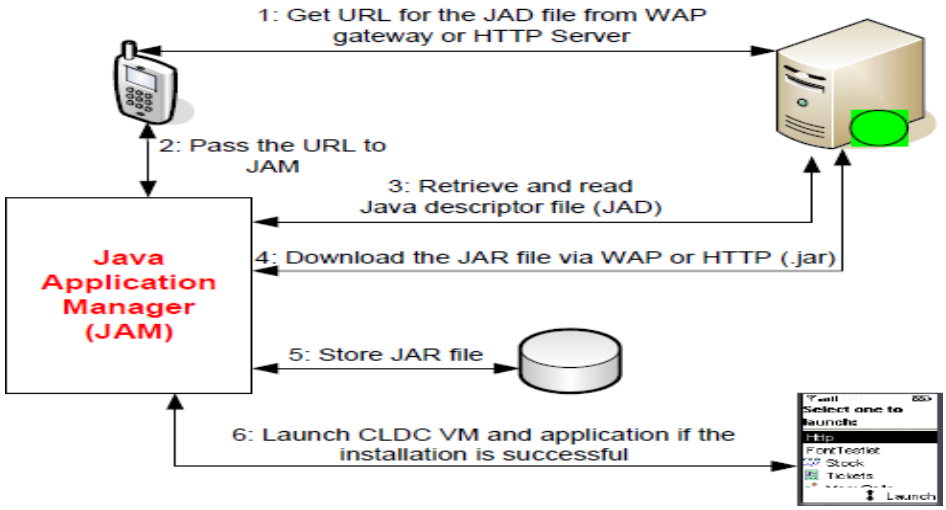


Рисунок 3.7. Бездротова підготовка MIDlets

Очікується, що пристрої забезпечать механізми, які дозволять користувачам відкривати пакети MIDlet, які можна завантажити на пристрій. В інших випадках це може бути резидентна програма, написана спеціально для ідентифікації наборів MIDlet для завантаження користувачем [23]. Процес надання OTA зображено на рисунок 3.7. Дескриптор програми Java (JAD) — це файл маніфесту з атрибутами, що ідентифікують назву пакета, версію, творця пакета MIDlet, опис профілю та версію конфігурації. Файл JAD корисний для розгортання OTA, оскільки менеджер програм на пристрої може перевірити властивості MIDlet, що містяться в JAD. Для розгортання КСБМП весь вихідний код мобільної реалізації обфускується разом із MIDlet і інкапсулюється в набір MIDlet. Потім пакет MIDlet поширюється на мобільний пристрій за допомогою методів однорангового або віддаленого розгортання. Для розгортання серверної реалізації ми рекомендуємо розгортати КСБМП як обфускований файл jar на настільному сервері, який захищений брандмауером.

3.4.2 Захист даних на пристрої

У цьому пункті ми продемонструємо безпеку даних на пристрої за допомогою програми, яка надійно зберігає паролі чи будь-які інші секрети на мобільному пристрої (рисунок 3.8).

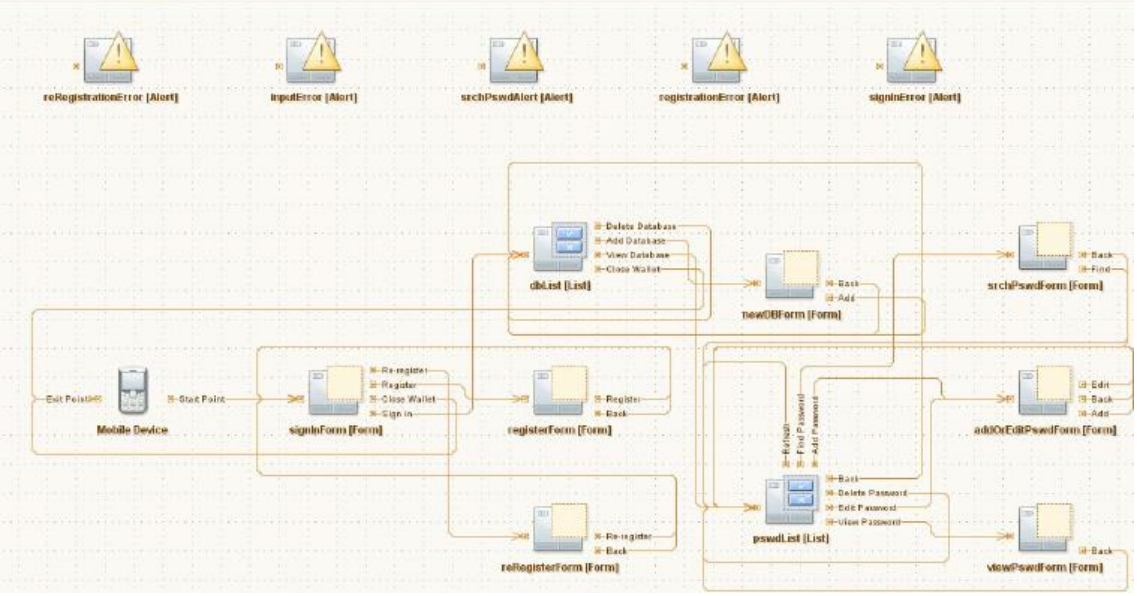


Рисунок 3-8. Перегляд потокового дизайну програми PasswordWallet

Додаток PasswordWallet має функції, які дозволяють користувачеві: зареєструвати/перереєструвати ім’я користувача, створити головний пароль та здійснювати операції з ним, керувати паролями на основі груп.

Для того, щоб керувати симетричним ключем як ключем PBE у КСБМП, розробник має зберегти на пристрої генеруючий ключ, параметри режиму та номер зашифрованого повідомлення. Ключ не можна повторно згенерувати без правильних даних автентифікації користувача, які надаються через параметр usr_id у методі initSymmetricKeyCipherComponent(), який повертає параметри генерації ключа та режиму, що продемонстровано нижче. Коли гаманець було вперше ініціалізовано, процес реєстрації обробляє генерацію параметрів генерації ключа та режиму шляхом виклику методу initNewSymmetricKeyComponent() у класі `SecurityManager`, який повертає ці параметри для збереження в сховищі записів. Зберігання параметрів обробляється в класі `UserManager`, який відповідає за вхід і реєстрацію користувача.

```
1  public Object [] initNewSymmetricKeyComponent(StringBuffer credentials) {
2  // Цей метод викликається, коли необхідно ініціалізувати новий симетричний ключовий компонент
3    CryptoService newService = CryptoService.getCryptoService(false);
4    try {
5      newService.initSymmetricKeyCipherComponent(credentials, null, null);
6      byte keyParams []= new byte[newService.getSymmetricKeyParamSize()];
7       byte modeParams[] = new byte [newService.getModeParamSize()];
8       newService.getSymmetricKeyParam(keyParams);
9       newService.getModeParam(modeParams);
```

```
10    Object params [] = new Object[2];
11    params[0] = keyParams;
12    params[1] = modeParams;
13    return params;
14   }catch(CryptoException ce){}
15   return null;
16 }
```

Ініціалізація криптосистеми з симетричним ключем для шифрування та дешифрування здійснюється наступним кодом.

```
1 public void initSecurityManager(StringBuffer credentials, byte [] keyParam, byte [] modeParam, boolean isReRegister) {
2   try {
3     if(isReRegister) {
4       if(regService != null) regService = null;
5       regService = CryptoService.getCryptoService(false);
6       regService.initSymmetricKeyCipherComponent(credentials, keyParam, modeParam);
7       hasReRegistered = true;
8     } else {
9       if(hasReRegistered && mainService != null) {
10          mainService = null;
11          hasReRegistered = false;
12      }
13      mainService = CryptoService.getCryptoService(false);
14      mainService.initSymmetricKeyCipherComponent(credentials, keyParam, modeParam);
15      mainService.setMessageNumber(trackNum);
16    }
17  }catch(CryptoException ce) {
18    ce.printStackTrace();
19  }
20 }
```

У вище вказаному коді збережені параметри генерації ключа та режиму передаються через `keyparam` і `modeparam` під час входу. Ми використали два екземпляри CryptoService, де один обробляє вхід, шифрування та дешифрування (mainService), а інший — повторну реєстрацію (regService). Під час повторної реєстрації програма повинна розшифрувати збережені паролі за допомогою старого головного пароля та повторно зашифрувати їх за допомогою нового головного пароля. Для звичайного входу номер повідомлення встановлюється на останній використаний номер. Номер повідомлення зберігається щоразу, коли пароль зберігається або редагується та отримується під час шифрування. Номер повідомлення ніколи не використовується повторно, навіть якщо пароль видалено. Паролі

розшифровуються лише в пам'яті. Код для шифрування та дешифрування можна посилатися на клас `Container`.

Функція MAC використовується для перевірки під час реєстрації користувача. MAC шенерується використовуючи параметри генерації ключа та режиму. Пароль хешується за допомогою SHA-256 у КСБМП, щоб використовувати його як секретне значення для MAC. Метод генерації MAC міститься в методі generateMAC() у класі `SecurityManager`. Щоб гарантувати цілісність MAC-адреси, використовується версія сховища записів. Тому значення MAC завжди буде відрізнятися відповідно до останньої модифікації сховища записів. Під час входу значення MAC перевіряється, щоб останній хеш був правильним і щоб запис не було підроблено. Код створення MAC для перевірки користувача представлений нижче.

```
1  public boolean register(String userName, String password) {
2   if(!isRegistered()) {
3    try {
4     StringBuffer cr = new StringBuffer();
5     cr.append(userName);
6     cr.append(password);
7    // ініціалізувати новий криптокомпонент симетричного ключа
8 Object params [] =
MainManager.getSecurityManager().initNewSymmetricKeyComponent(cr);
9     byte temp [] = null;
10     // Зберігання облікових даних користувача в такому порядку...
11     // 1: Параметр симетричного ключа
12     // 2: параметр режиму
13     // 3: Хеш-дані пароля
14     userRecord.addRecord(temp = (byte [])params[0], 0, temp.length);
15     userRecord.addRecord(temp = (byte [])params[1], 0, temp.length);
16     temp = null;
17     // тепер ми знаємо додаткову цінність оновлення...
18     Object authData [] = {Integer.toString(userRecord.getVersion() + 1)};
19     Object toMAC [] = ((byte [])params[0], (byte [])params[1]};
20    byte [] mac = MainManager.getSecurityManager().generateMAC(password.getBytes(),
toMAC, authData);
21     userRecord.addRecord(mac, 0, mac.length);
22     return true;
23    } catch(RecordStoreException rse) {}
24   }
25  return false;
26 }
```

ВИСНОВКИ

1. Розроблено структуру пакетів криптографічної сиситеми у вигляді окремих компонентів в стилі об’єктно-орієнтованої та багаторівневої архітектури. Під час розробки компонентів було затосовано шаблони проектування Factory, Façade і Delegation, які забезпечують гнучкий дизайн для розгортання схеми шифрування, застосування ефективної криптографії та створення мережевого з’єднання.

2. Проведено якісний аналіз створеної криптографічної системи. На основі метрик нестабільності та абстрації здійснено кількісну оцінку архітектурного дизайну системи. Здійснено порівняння цих показників із аналогічними програмними продуктами такими, як Bouncy Castle API (BC) та Secure and Trust Service API (SATSA).

3. Проведено аналіз продуктивності шифрування і дешифрування ширфів, які використовуються у продукті. Аналіз проводився для визначення часу необхідного для цих вище вказаних операції для наступних ширфів: простий AES із 128-бітним і 256-бітним ключем, AES з режимом лічильника із 256-бітним ключем та RSA зі схемою кодування OAEP із 2048-бітним ключем

4. Продемонтровано покращення системи безпеки криптографічного пакету у порівняні із BC і SATSA під час створення MAC та цифрового підпису. Поліпшено процес використання криптографічних алгоритмів з симетиричним і асиметричним ключем.

5. Використовуючи мову програмування Java, було реалізовано криптографічну систему у які використовуються запропоновані методи шифрування та створено фреймворк, який надає доступ до оперування цими функціями.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1) M. Bellare and P. Rogaway 1995. Optimal Asymmetric Encryption - How to Encrypt with RSA. *Advances in Cryptology - Eurocrypt '94*, vol. 950 of Lecture Notes in Computer Science, 92-111, Springer Verlag.

2) The Legion of BouncyCastle 2005. BouncyCastle API. [Online]. Available: www.bouncycastle.org. Last accessed on: 10 November 2005.

3) CellularOnline 2006. Mobile User Statistics. [Online]. Available: http://www.cellular.co.za. Last accessed on: 10 November 2006.

4) W. Diffie and M. Hellman November 1976. New Directions in Cryptography. *IEEE Transactions on Information Theory*.

5) T. ElGamal 1985. A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. *IEEE Transactions on Information Theory*, vol. IT-31, n. 4, 469-472.

6) C Ellison and B Schneier 2000. Ten Risks of PKI. *Computer Security Journal*, vol. 16, n.1, 1-7.

7) Niels Ferguson and Bruce Schneier 2003. *Practical Cryptography*. Indianapolis, Indiana: Wiley Publishing, Inc.

8) Brian Gladman 2005. Code for AES and Combined Encryption/Authentication Modes. [Online]. Available: http://fp.gladman.plus.com/cryptography_technology/index.htm. Last accessed on: 18 November 2005.

9) GNU Crypto 2006. The GNU Crypto Project. [Online]. Available: http://www.gnu.org/software/gnu-crypto/. Last accessed on: 02 October 2006.

10) Constatinos F. Grecas, Sotirios I. Maniatis, and Iakovos S. Venieris 2003. Introduction of the Asymmetric Cryptography in GSM, GPRS,UMTS, and Its Public Key Infrastructure Integration. *Mobile Networks and Applications*, vol. 8, 145-150.

11) Peter Gutmann 2001. The Design and Verification of a Cryptographic Security Architecture, PhD Thesis, Department of Computer Science, University of Auckland, New Zealand.

12) IAIK 2004. IAIK JCE-ME library. [Online]. Available: http://jce.iaik.tugraz.at/products/10_mejce/index/php. Last accessed on: 12 June 2004.

13) IBM 2004. Structural Analysis for Java. [Online]. Available: https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?source=AW-

0FF&S_PKG=0FF&S_TACT=105AGX30&S_CMP=DEVX. Last accessed on: 27 November 2006.

14) Internet Engineering Task Force 1994. "RFC 1750 - Randomness Recommendations for Security."

15) Internet Engineering Task Force 1997. "RFC 2104 - HMAC: Keyed-Hashing for Message Authentication."

16) Internet Engineering Task Force 2002. "RFC 3280 - Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile."

17) Burt Kaliski 2003. TWIRL and RSA Key Size. [Online]. Available: http://www.rsasecurity.com/rsalabs/node.asp?id=2004. Last accessed on: 18 April 2006.

18) Jonathan Knudsen 2003. *Wireless Java: Developing with J2ME*. New York: Apress.

19) Donald E. Knuth 1998. *The Art of Computer Programming*. vol. 2, 3rd ed: Addison-Wesley.

20) T. Kohno 2004. Analysis of the WinZip encryption standard. IACR ePrint Archive, University of California at San Diego, vol. 2004/078.

21) Timothy C. Lethbridge and Robert Laganière 2005. *Object-Oriented Software Engineering Practical Software Development using UML and Java*. 2nd ed. Glasgow: McGraw Hill.

22) Helger Lipmaa, Phillip Rogaway, and David Wagner January 2003. Comments to NIST concerning AES Modes of Operations: CTR-Mode Encryption. [Online]. Available: http://csrc.nist.gov/CryptoToolkit/modes/workshop1/papers/lipmaa-ctr.pdf. Last accessed on: 12 September 2005.

23) Qusay H. Mahmoud 2002. Deploying Wireless Java Applications. [Online]. Available: http://developers.sun.com/techtopics/mobility/midp/articles/deploy/. Last accessed on: 18 August 2004.

24) Robert C. Martin 2002. *Agile Software Development*. 2nd ed: Addison-Wesley.

25) Microsoft Corporation 2003. Windows CE and Pocket PC. [Online]. Available: http://www.microsoft.com/windowsmobile/pocketpc/ppc/default.mspx. Last accessed on: 10 August 2004.

26) Microsoft Corporation 2005. Whats New in the .NET Compact Framework 2.0. [Online]. Available: http://msdn2.microsoft.com/en-us/library/aa446574.aspx. Last accessed on: 23 May 2006.

27) V.S. Miller 1986. Use of Elliptic Curves in Cryptography. *Advances in Cryptology - Crypto 85 Proceedings, Springer Verlag*, 417-426.

28) National Institute of Standards and Technology 2002. Secure Hash Standard. *Federal Information Processing Standards Publication 180-2*.

29) National Institute of Standards and Technology 1991. "Digital Signature Standard - FIPS PUB 186."

30) NTRU 2004. NTRU Toolkits. [Online]. Available: http://www.ntru.com/cryptolab/algorithms.htm. Last accessed on: 10 March 2004.

31) C. Enrique Ortiz 2005. The Security and Trust API (SATSA) for J2ME: The Security APIs. [Online]. Available: http://developers.sun.com/techtopics/mobility/apis/articles/satsa2/. Last accessed on: 05 May 2006.

32) David Pointcheval 2002. How to Encrypt Properly with RSA. *RSA Laboratories Cryptobytes*, vol. 5, 10-19.

33) Karen Renaud, Judith Bishop, Johnny Lo, and Basil Worrall 2005. Algon: from interchangeable distributed algorithms to interchangeable middleware. *Software Composition 2004, Electronic. Notes Theory. Computer Science*, 114:65-85.

34) R. Rivest, A. Shamir, and L. Adleman February 1978. A Method for Obtaining Digital Signatures and Public Key Cryptosystems. *Communications of the ACM*.

35) RSA Laboratories 2002. PKCS#1 v2.1: RSA Cryptography Standard. *PCKS Standards*.

36) RSA Laboratories 2006. What is PKI? [Online]. Available: http://www.rsasecurity.com/rsalabs/node.asp?id=2268. Last accessed on: 9 August 2006.

37) James Rumbaugh, Ivar Jacobson, and Grady Booch 1999. *The Unified Modeling Language Reference Manual*. Addison Wesley Longman, Inc.

38) Bruce Schneier 1996. *Applied Cryptography*. 2nd ed. New York: John Wiley & Sons, Inc.

39) William Stallings 2000. *Network Security Essentials, application and standards*. New Jersey: Prentice Hall.

40) Jari Sukanen 2004. *Extension framework for Symbian OS application*, Master of Science Thesis, Department of Computer Science, University of Helsinki, Finland.

41) Sun Microsystems 2002. Java Cryptography Architecture Reference. [Online]. Available: http://java.sun.com/products/jdk1.2/docs/guide/security/cryptospec.html. Last accessed on.

42) Sun Microsystems 2002. JSR 118: MIDP 2.0 Specification.

43) Sun Microsystems 2004. Security and Trust Service API. [Online]. Available: http://java.sun.com/products/satsa/. Last accessed on: 12 September 2005.

44) Sun Microsystems 2006. Java Card Development Kit version 2.2.2. [Online]. Available: http://java.sun.com/products/javacard/. Last accessed on: 10 April 2006.

45) S.A. Vanstone 2003. Next generation security for wireless: elliptic curve cryptography. *Computers and Security*, vol. 22, 12-44.

46) John Walker 1998. ENT Tool. [Online]. Available: http://www.fourmilab.ch. Last accessed on: 20 September 2006.

47) Michael J. Wiener 1990. Cryptanalysis of short RSA secret exponents. In *IEEE Transactions ofInformation Theory*, vol. 36, 553-558.

48) Wikipedia 2005. History of Cryptography. [Online]. Available: http://en.wikipedia.org/wiki/History_of_cryptography Last accessed on: 07 July 2005.

49) Micheal Juntao Yuan 2004. *Enterprise J2ME - Developing Mobile Java Applications.* Upper Saddle River, New Jersey: Prentice Hall.