Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”

КОМП’ЮТЕРНИЙ ПРАКТИКУМ З ДИСЦИПЛІНИ «МЕТОДИ РЕАЛІЗАЦІЇ КРИПТОГРАФІЧНИХ МЕХАНІЗМІВ» №2

Реалізація алгоритмів генерації ключів гібридних криптосистем

Виконала:
студентка групи ФІ-12мн
Звичайна Анастасія Олександрівна

Перевірила:
Селюх Поліна Валентинівна

Київ 2021

Мета роботи: Дослідження алгоритмів генерації псевдовипадкових послідовностей, тестування простоти чисел та генерації простих чисел з точки зору їх ефективності за часом та можливості використання для гененерації ключів асиметричних криптосистем.

Умова задачі: Запропонувати схему генератора ПВЧ для інтелектуальної картки, токена чи смартфону. Розглянути особливості побудови генератора простих чисел в умовах обмеженної пам’яті та часу генерації.

Хід роботи та необхідна теорія:

Для початку треба зрозуміти що таке токен та інтелектуальна картка. Токен є невеликим пристроєм, яке інколи за зовнішнім виглядом нагадує флешку, але в середині замість області для зберігання файлів має чіп (або мікропроцесор), що складається з таких складових:

- операційна система;
- APDU інтерфейс для зв'язку з зовнішніми операційними системами та пристроями (логічний рівень);
- захищена пам'ять, де зберігається ключова інформація, доступ до якої можливий лише при вводі PIN-кода (з фіксованою кількістю спроб);
- кріптоядро (копроцесор, NPU, numeric processing unit, cryptoprocessor), що дозволяє виконувати криптографічні операції (цифровий підпис, шифрування).

Токени зустрічаються у різних представленнях, які виробники називають форм-факторами. Найрозповсюдженішими форм-факторами є USB-токени та інтелектуальні картки (смарт-картки). Окрім цього, зустрічаються мікротокени для ноутбуків та microSD-токени для смартфонів й планшетів.

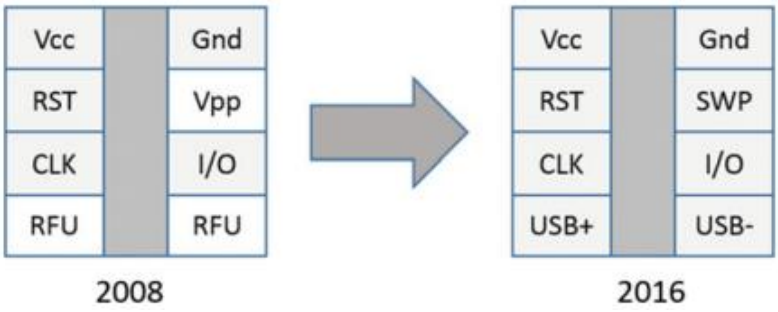
Токени використовують для ідентифікації та автентифікації користувачів у корпоративних і державних установах, адже вони є безпечними носіями цифрових підписів, ключів та сертифікатів. Також існують токени з апаратною криптографією. Для контролю доступу використовують токени з RFID-мітками (обсяг пам'яті яких 150-700 Байт), що можуть бути контактними та безконтактними – так інтелектуальна картка (токен) стає аналогом пропуску на роботу. Основною відмінністю між смарт-карткою та токеном є те, що токен може бути безпосередньо підключений до комп'ютера за допомогою інтерфейсу USB, коли для першого потрібен пристрій для читання карт.

Існують різні стандарти, які описують вимоги та рекомендації щодо реалізації та впровадження токенів/інтелектуальних карток. Так, для контактних смарт-карток існує ISO/IEC 7816, для безконтактних – ISO/IEC 14443 (інколи використовують ISO/IEC 15693, що призначено для карт з більшим радіусом дії).

Картки бувають з магнітними смугами, з чіпами, з мікропроцесорами, з RFID-мітками і їх всі часто називають «смарт», але необхідно з'ясувати що ж таке «смарт» насправді. Картки на магнітних смугах використовують смуги, що є аналогічними смугам на касетних магнітофонах, де магнітне поле вириває диполі (у них зберігається інформація, а метод зберігання відомий як формат Wiegend) по довжині смуги. Даний процес можна перевіряти та відновлювати за допомогою головки стрічки, що зчитує інформацію. Однак, обсяги пам'яті у картках на магнітних смугах дуже обмежені, тому використовують декілька смуг. Даний тип карток не належить до типу «смарт», адже їх дуже просто підробити.

Чіп-картки схожі на магнітні, але мають спеціальний пристрій, який називають чіпом (чи мікросхемою). Контакти VCC/GND, лінія введення-виведення зв'язку, а також лінії синхронізації (CLK) та скидання (RST) забезпечують живлення мікросхеми. VPP колись використовувався для перепрограмування EEPROM (Electrically Erasable Programmable Read-Only Memory) пам'яті старого стилю, але тепер він застарів. Оскільки два нижні контакти історично не використовувалися і зарезервовані для майбутнього використання (RFU), деякі карти та зчитувачі використовують лише

шість верхніх контактів. Однак у деяких галузях стандартизовано використання старих контактів RFU та VPP для нових послуг.

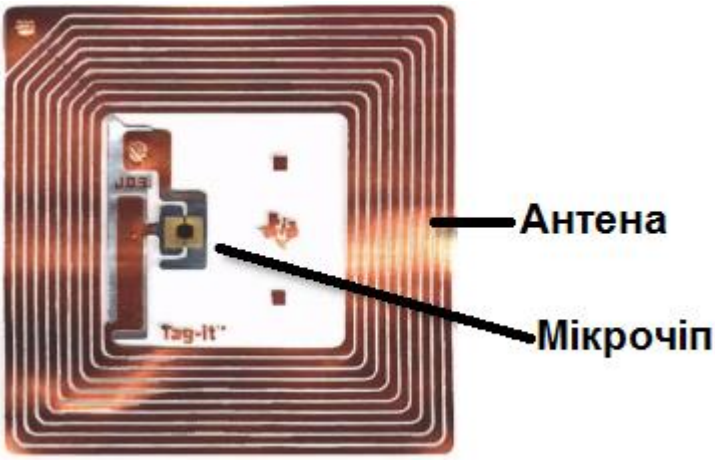


Доступ до контактної чіп-карти можна отримати, помістивши її у пристрій для читання карт, який дозволяє чіпу отримувати живлення, синхронізувати та підтримувати зв'язок. Електричний інтерфейс низького рівня описаний у стандартах ISO 7816. Зв'язок є напівдуплексним, при цьому зчитувач (хост) виконує роль провідного пристрою, а карта – веденого пристрою. Фактично існують два низькорівневі протоколи, які можуть використовуватися для обміну інформацією, відомі як T = 0 і T = 1, і обидва вони визначені в стандарті ISO 7816. Протокол T = 0 користується популярністю в індустрії мобільного зв'язку (тепер стає зрозуміло чому у завданні пропонується також смартфон), а протоколу T = 1 віддають перевагу у фінансових установах. Можна припустити, що такі картки є смарт-картками, адже вони використовуються сьогодні скрізь, але такі припущення є неправильними, адже є практичні атаки на EVM чіпи (наприклад атаки липня 2020 року на Visa картки).

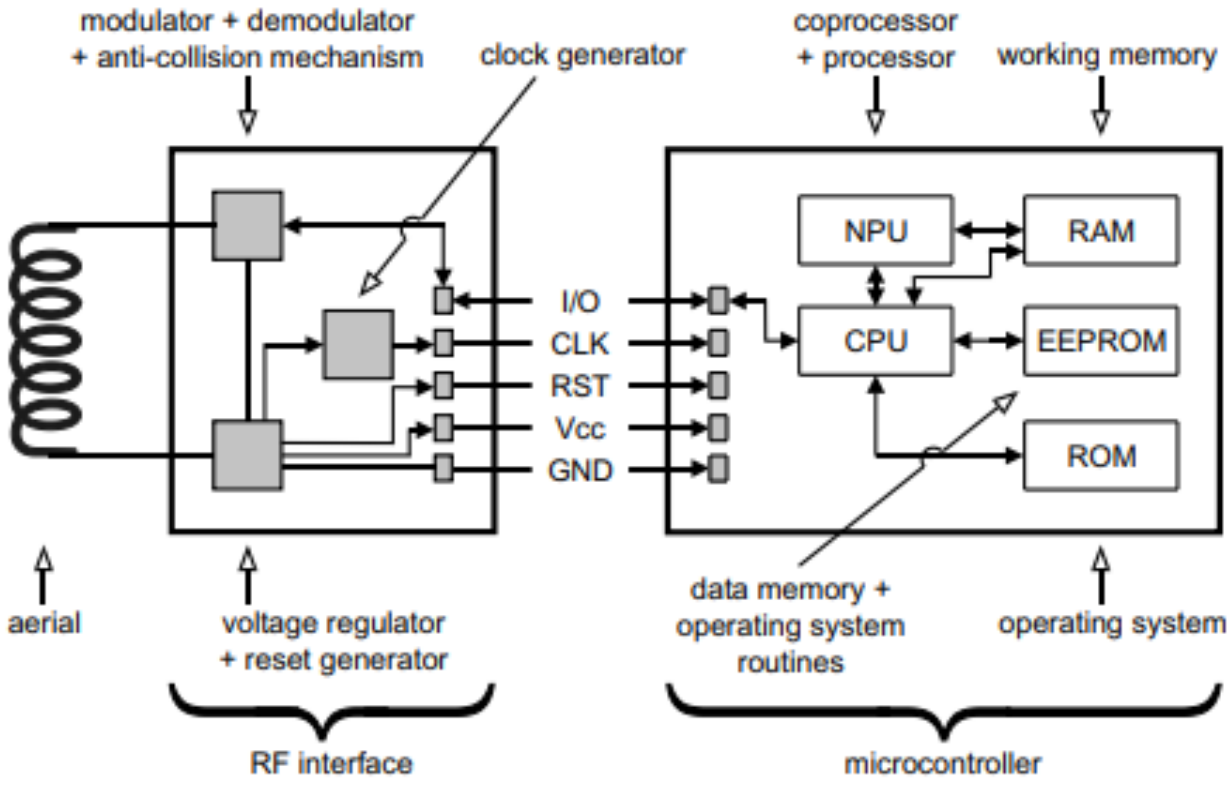
Інтерфейс протоколу мікропроцесорних карт визначений так, щоб було неможливо отримати інформацію або перепрограмувати вміст картки без відповідних дозволів, які перевіряються та застосовуються криптографічними функціями. Однак, вважати таку картку смарт-карткою також не можна, адже зловмисників не зупиняє захист на логічному рівні, вони можуть виконувати атаки, які направлені безпосередньо на мікросхему, використовуючи витік інформації з пристрою, який працює, тому дуже важливим є те, щоб мікропроцесор був спроектован на захист від несанкціонованого доступу.

Окрім цього, зростає інтерес та використання карт, які не мають фізичних контактів, але натомість використовують радіотехніку – RFID-картки, які вважають смарт-картками через їх безконтактність. Більш того, якщо говорити про смартфони, то там використовують подібну технологію, а саме NFC, яка дозволяє виконувати платежі,

але така технологія не так розповсюджена як QR-коди через необхідність звуження ринку потенційних користувачів того чи іншого банку при роботі або з GooglePay, або з ApplePay, або з іншою системою платежів (а свою власну створювати дорого).



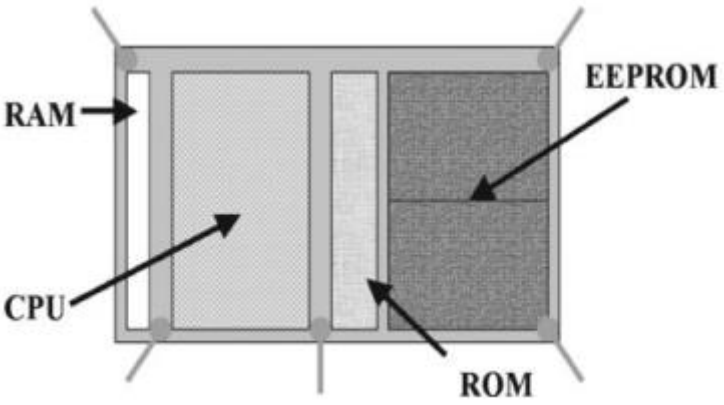
На мою думку, найкращою смарт-карткою є мікропроцесорні RFID-картки, тобто:

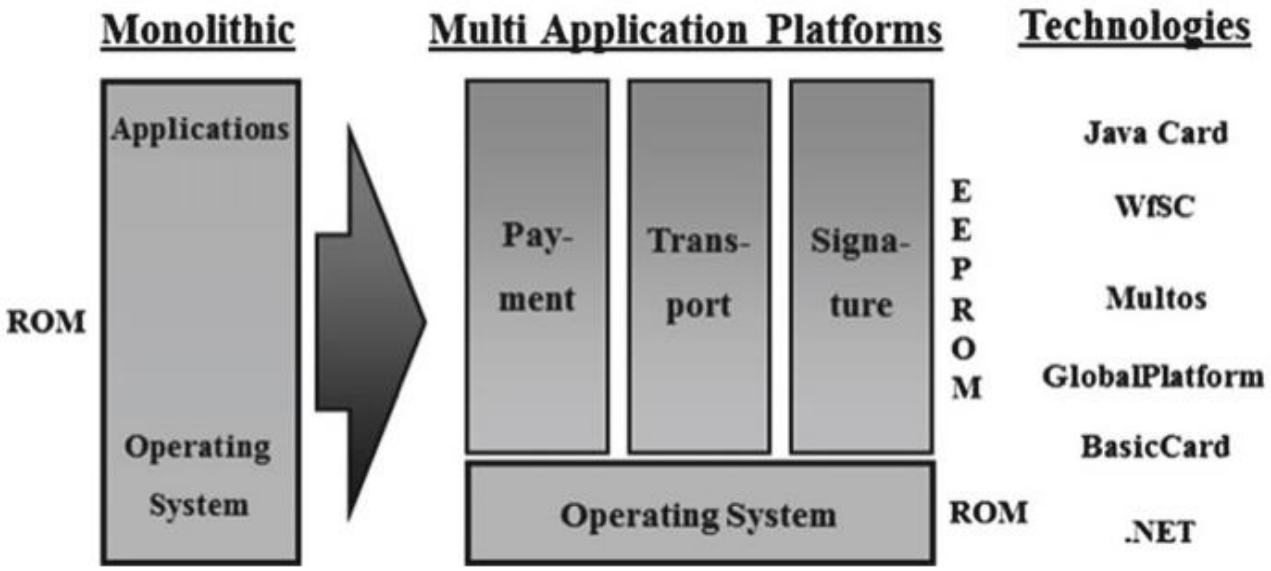


ВИДИ СМАРТ-КАРТОК ТА ОГЛЯД ЇХ БУДОВИ

Сама ідея розробки смарт-картки вперше з'явилась у двох німців у 1960 році (Helmut Gröttrup and Jürgen Dethloff), а у 1978 році ця ідея вже остаточно була втілена у життя (Michel Ugon). На початку 90х смарт-картки мали 1-3КБ ROM (ПЗП, Read Only Memory), приблизно 128Б RAM (Random Access Memory, ОЗП) та приблизно 1-2КБ EEPROM (Electrically Erasable Programmable Read Only Memory). Найбільша частина функціоналу карток знаходилась у RAM, тому що RAM використовувало (і продовжує використовувати) дуже малу площину чіпу на біт у границях загального

розміру мікропроцесору смарт-картки. Більш того, тоді вони були орієнтовані на виконання лише однієї задачі, а через те, що функціонал таких смарт-карток не можна було змінити після того, коли чіпи були вбудованими, структуру таких карт називали монолітною. Дану проблему вирішували за допомогою переходу від ROM до EEPROM при роботі з покращеним кодом. Така реалізація вимагала велику кількість навичок від програмістів, а також була дорогою, так виникли смарт-карти з ОС (SCOS, Smart Card Operating System). Однак, і там були проблеми, які пов'язані з неможливістю використання одного й того ж коду на картках з різними мікропроцесорами. Після цього головною метою стояла незалежність програми від ОС, можливість безпечно оброблювати декілька застосунків, а також можливість змінювати карту після її випуску. Дані вимоги були реалізовані у період з 1995 по 1999 роки з появою потужніших мікропроцесорів, що мали 6-8КБ ROM, 128Б RAM та приблизно 5-12КБ EEPROM, але така апаратна платформа не давала прискорення у швидкості виконання операцій (криптографія була однією з основних проблем).

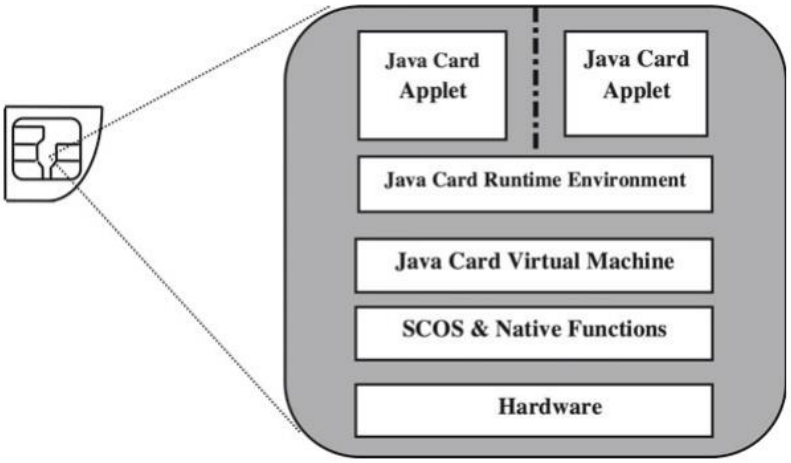




Основні зусилля по представленню платформ смарт-карт з декількома застосунками були запропоновані у період з 1997 по 2001 рік, загалом внаслідок впровадження

таких платформ як Java Card, Windows for Smart Cards (WfSC), MULTOS, GlobalPlatform, BasicCard та .NET.

Java Card



Незабаром після реалізації концепції Java Card (приблизно в 1995 році) був створений форум, який відповідав за просування та підтримку специфікацій Java Card. Цілі Java Card Forum описані на веб-сайті Java Card Forum (https://javacardforum.com). Кожний виробник смарт-карт розробляв власну реалізацію на Java Card, яка є підмножиною звичайної мови Java. Аналогічно віртуальна машина Java Card (VMJC) є підмножиною віртуальної машини Java, а Java Card API є підмножиною Java API. Концепція Java Card усуває залежність від базової операційної системи смарт-карт і може розташовуватися практично на вершині будь-якої операційної системи смарт-карт, яка пропонує рівень апаратної абстракції (HAL). Тривалість життя віртуальної машини Java Card, на відміну від традиційних середовищ Java, працює вічно. Термін життя програми Java Card починається, коли вона правильно встановлена та зареєстрована в реєстрі карток. З цієї причини фреймворк Java Card пропонує ряд процедур. На жаль більшість реалізацій Java Card VM не пропонують функції збирання сміття. Це означає, що програмісти повинні вручну перерозподіляти пам'ять, яка не використовується їх програмою, що змушує програмістів Java Card дуже ретельно думати про те, як їх програми будуть поводитися, особливо в межах обмежених ресурсів пам'яті мікропроцесора смарт-картки.

Однією з перших концепцій, які були визначені в API Java Card, є атомарність транзакцій. Тобто будь-які оновлення постійних об’єктів будуть атомарними, тобто «транзакція повинна бути виконана або скасована повністю». Серед основних функцій безпеки моделі Java Card є брандмауер. Брандмауер Java Card відповідає за забезпечення необхідного механізму ізоляції між програмами, тобто уникнення будь-

якого несанкціонованого зв'язку між програмами. Через механізми спільного використання об'єктів та брандмауер програмам дозволяється спілкуватися один з одним. Механізм, який дозволяє спілкуватися між додатками, дуже суворо визначений API Java Card. У рамках цієї концепції об'єкти належать аплету, який їх створив.

Поняття файлової системи смарт-карт у традиційній формі ISO 7816-4 було вилучено з останніх версій Java Card API. Однак цілком можливо створювати об'єкти файлів зі специфічними для користувача класами, напр. масиви. Додаткові пакети Java Card позначаються «x», що часто називають розширеннями. Пакет java.lang містить усі основні класи Java Card; javacard.framework включає всі основні класи для керування аплетами; javacard.security надає доступ до криптографічних примітивів, а сама криптографія часто забезпечується класом javacardx.crypto, який надає інтерфейс для підтримки криптографічних операцій. Java Card не містить API, який дозволяє аплету безпосередньо викликати власні методи. Однак, компоненти Java Card Runtime Environment (JCRE) можуть отримати доступ до цих власних методів, особливо при використанні криптографічних алгоритмів. Серед недоліків є те, що існування нативних методів руйнує модель інкапсуляції, а також ставить під питанням можливість виконання вимоги portability.

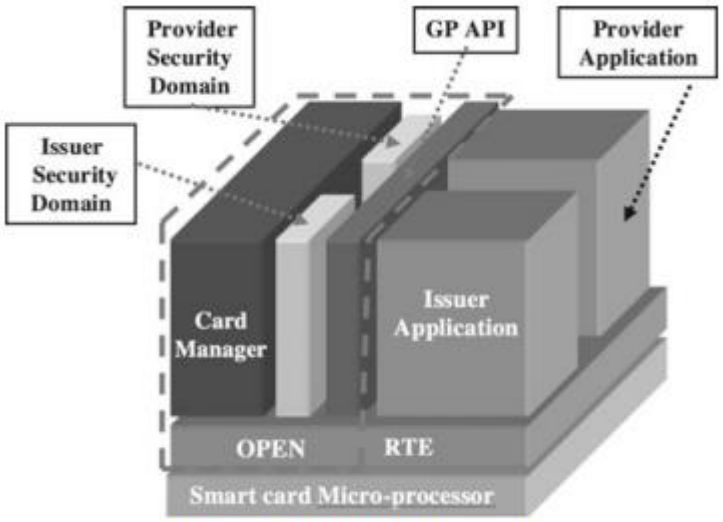
Java Card ефективно розвивається, але чи буде вона домінувати на ринку смарт-карт покаже лише час. Наразі останньою версією є Java Card 3.0.5, документація якої знаходиться за посиланням: https://docs.oracle.com/javacard/3.0.5/guide/index.html.

GlobalPlatform

Наприкінці 1990-х років Visa International, як один із найбільших виробників карток у світі, почала досліджувати проблеми, пов'язані з технологією смарт-карт. Через пару років корпорація створила набір стандартів Visa Open Platform (VOP). Ці стандарти визначали як можна обробляти кілька програм на рівні системи керування карткою, терміналом та смарт-карткою. Незабаром після цього Visa зрозуміла, що для ширшого впровадження цих стандартів вони повинні бути максимально відкритими та загальнодоступними. Це разом з іншими бізнес-рішеннями змусило Visa передати

Visa Open Platform консорціуму OpenPlatform. Консорціум складався з ряду організацій, які цікавилися цією технологією. Приблизно у 1999 році OpenPlatform був перейменований у GlobalPlatform, в результаті чтого специфікації також були перейменовані. GlobalPlatform є незалежною некомерційною асоціацією, яка відповідає за просування стандартів GlobalPlatform та технології смарт-карт загалом. Насьогодні стандарти GlobalPlatform визначають найкращі методи та архітектуру для карт і операційних систем, терміналів і бек-офісних систем (back-office systems). Основна ідея стандартів GlobalPlatform полягає у стандартизації певних аспектів технології, щоб покращити сумісність, доступність і безпеку технології смарт-карт із багатьма програмами.

Специфікація карт GlobalPlatform (GPCS) визначає набір логічних компонентів, які мають на меті підвищити безпеку, портативність та взаємну сумісність смарт-карт із кількома програмами. Специфікація картки GlobalPlatform не залежить від базової платформи смарт-карт. У нижній частині запропонованої архітектури карт GlobalPlatform знаходиться мікропроцесор. RTE розглядається як рівень абстракції між GPCS і базовим обладнанням. Як правило, RTE складається з операційної системи смарт-карти (SCOS), віртуальної машини (VM) та інтерфейсу прикладного програмування (API). Більш того, смарт-карта може підтримувати будь-яку операційну систему чи базову віртуальну машину (наприклад, WfSC, MULTOS).



GlobalPlatform card architecture

Специфікація GlobalPlatform знаходиться за посиланням:

https://globalplatform.org/wp-content/uploads/2018/05/GPC_CardSpecification_v2.3.1_PublicRelease_CC.pdf.

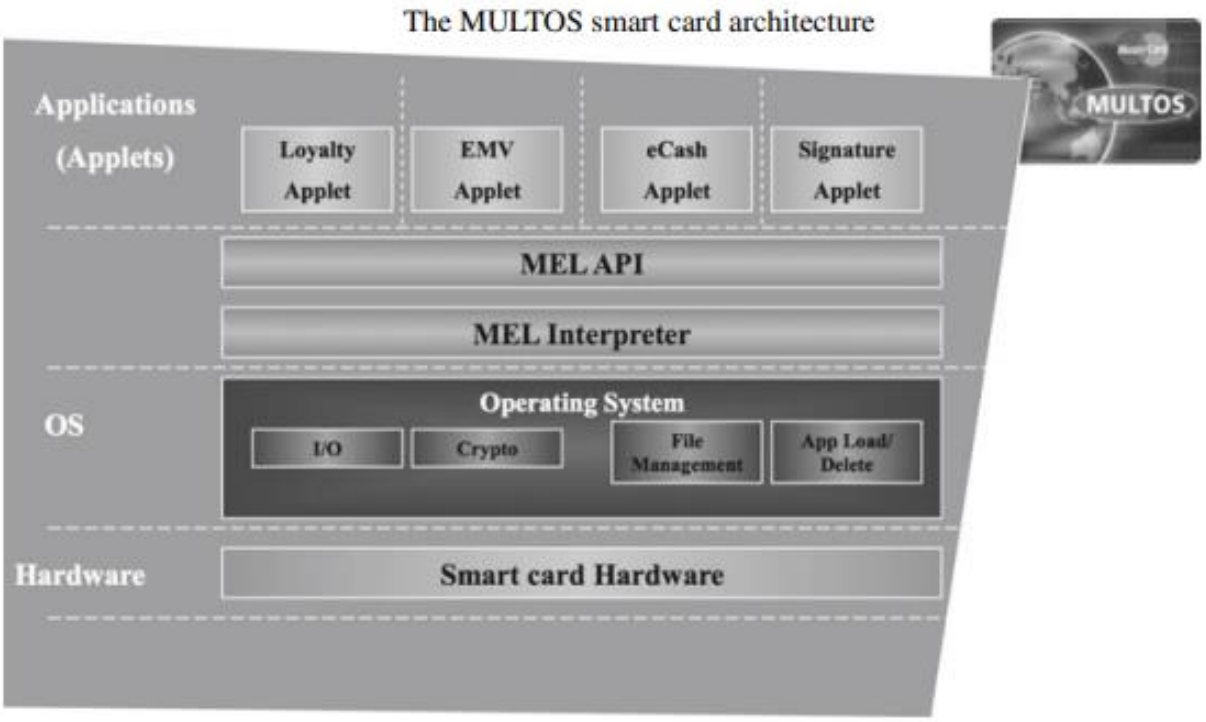
Цікаво зазначити, що боротьба між Visa і MasterCard, принаймні в якійсь мірі, була вирішена, коли MasterCard приєдналася до асоціації GlobalPlatform.

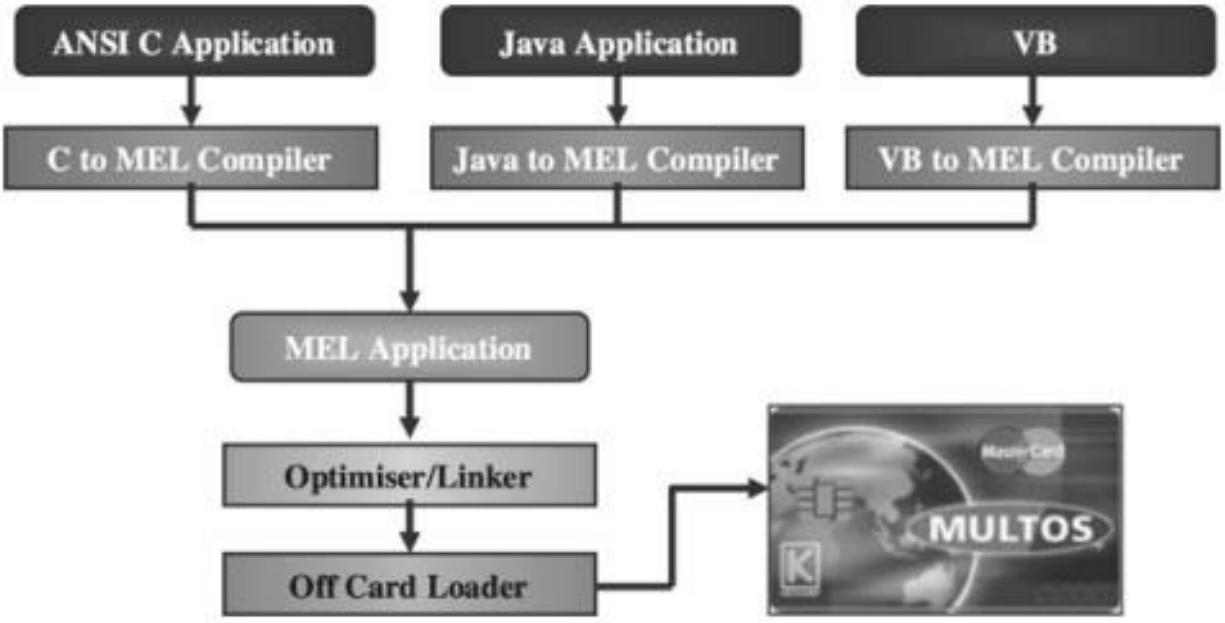
Windows for Smart Cards (WfSC)

У 1998 році була запропонована операційна система WfSC, яка потребувала 8-розрядного CPU, 32 КБ ROM та 32 КБ EEPROM. Розробники WfSC на меті мали реалізацію гнучкої ОС, а також існування GSM та додаткової криптографічної підтримки. Файлова система ОС була заснована на стандарті ISO 7816 і на добре відомій таблиці доступу до файлів (FAT) операційних систем Microsoft для настільних комп'ютерів.

Програми можна розробляти на VB та C++ у середовищі Visual Studio. Зв'язок з картами WfSC здійснюється за допомогою APDU. Віртуальна машина WfSC є однією з найкращих сьогодні віртуальних машин для смарт-карт. На початку жовтня 2021 року компанія Microsoft оновила свою документацію щодо смарт-карт (див. https://docs.microsoft.com/en-us/windows/security/identity-protection/smart-cards/smart-card-windows-smart-card-technical-reference).

Multos

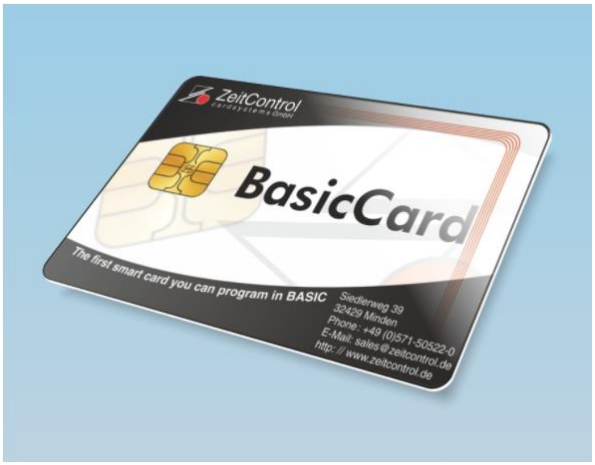




Overview of MULTOS application development cycle

Влітку цього року у MULTOS з'явилась нова документація для розробників, яка знаходиться за посиланням: https://multos.com/wp-content/uploads/2021/06/MDG.pdf.

BasicCard



Basic Card з'явилася приблизно в 1996 році, а назву свою отримала саме через мову Basic, яка є її підґрунтям, а основною мовою картки є ZeitControl-Basic.

Вихідний код ZC-Basic компілюється в P-Code, який розглядається як проміжна мова, подібна до машинного коду, але тільки під час виконання віртуальною машиною. Щоб створити та завантажити P-код на карту BasicCard, ZeitControl має інтегроване середовище. Програмне забезпечення дозволяє налагоджувати та розробляти BasicCard та пов’язані термінальні програми.

В основі операційної системи BasicCard є інтерпретатор Basic, що займає приблизно 17 Кб RAM, який виконує P-коди. Концепція APDU в BasicCard, як зазначають розробники, є найзрозумілішою з усіх, що є перевагою.

З метою подальшого посилення безпечного управління платформою BasicCard визначає «Механізм безпечного транспортування», який дозволяє шифрувати файли та ключі за допомогою ключів, спільних між карткою (завантаженою під час фази ініціалізації карти) та приймачем. Крім того, BasicCard визначає «автоматичний менеджер транзакцій EEPROM», що надає розробникам певну гнучкість при роботі з пам'яттю, а сама концепції атомарності транзакцій аналогічна Java Card.

Також BasicCard пропонує ряд додаткових бібліотек програмування (в основному криптографічних), які спрямовані на підтримку розробки передових і спеціалізованих програм.

Ще одною перевагою BasicCard є низькі ціни продажу та той факт, що їх можна відносно легко отримати через веб-сайт ZeitControl (https://www.zeitcontrol.de/de).

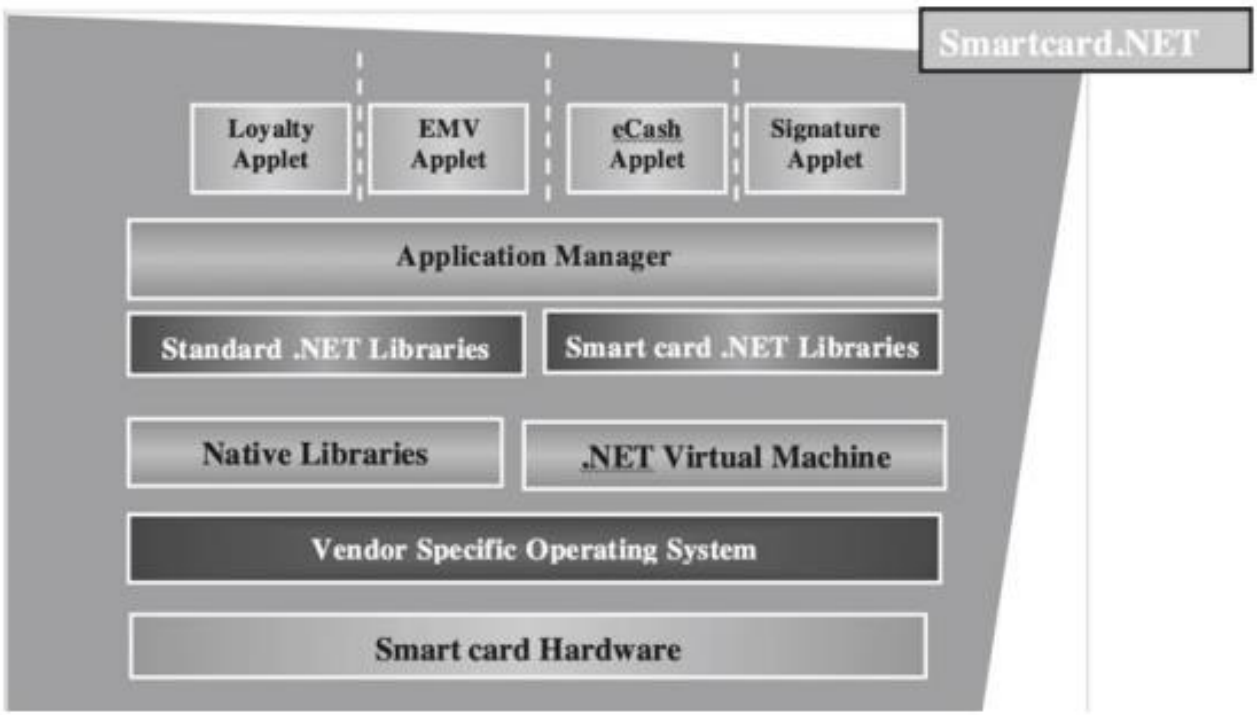
.NET

Фактична назва продукту була Nectar Smartcard.NET і була заснована на технологіях Microsoft.NET. Платформа була розроблена максимально наближеною до міжнародної організації зі стандартизації ECMA335 .NET специфікацій та .NET Framework.

Варто зазначити, що Gemalto (один із лідерів у галузі технології смарт-карт) пропонує продукти для смарт-карт на основі архітектури .NET. Цільова програма компілюється у проміжну мову Microsoft (MSIL). Тобто програми .NET можуть бути написані на C#, C++, Visual Basic (VB), J#, JavaScript тощо, які можуть бути скомпільовані в код MSIL. Оскільки програми, написані вищевказаними мовами, будуть скомпільовані до коду .NET, стає можливим поєднання однієї або кількох мов для однієї програми смарт-карти.

Серед найпомітніших переваг платформи .NET є те, що розробнику не потрібно використовувати APDU для зв'язку із зовнішнім світом. Технологія віддаленого доступу дозволяє розробникам викликати послуги програмного забезпечення, використовуючи TCP/HTTP. Картка Smartcard.NET пропонує віртуальну машину, яка нав’язує концепцію «Домену програми» як механізму ізоляції запущених програм і

уникнення спільного доступу до даних без документів. Платформа також пропонує механізм збору сміття. Механізм RPC в .NET дозволяє взаємодіяти між програмами. Нарешті, остання платформа Nectar Smartcard.NET також підтримує потоки, 64-розрядні цілі числа та перевірку коду картки.



Детальніше про будову смарт-карт та їх функціювання можна почитати у книгах:

1) W. Rankl and W. Effing. *Smart Card Handbook*. 2003.
2) K. Mayes, K. Markantonakis. *Smart Cards, Tokens, Security and Applications*. 2017.

ПСЕВДОВИПАДКОВІ ГЕНЕРАТОРИ ТА СТАТИСТИЧНІ ТЕСТИ

Після впровадження генератора випадкових чисел, як правило, необхідно перевірити якість чисел, які він створює. Ідеально коли у згенерованих випадкових числах майже рівна кількість одиниць і нулів. Випадкові числа можна математично перевірити за допомогою стандартних статистичних процедур. Стандартною практикою є генерація від 10 000 до 100 000 (8-бітових) випадкових чисел для того, щоб статистичний аналіз давав достатньо надійні результати. Єдиний спосіб перевірити таку кількість чисел – використовувати комп’ютеризовані програми тестування. При оцінці якості випадкових чисел необхідно також дослідити розподіл утворених чисел. Якщо він є помітно нерівномірним, то за це можна зачепитись криптоаналітику. Окрім цього

теорема Бернуллі має виконуватись якомога точніше, тобто поява певного числа, незалежно від того, що було до нього, залежить лише від ймовірності появи самого числа. Наприклад, ймовірність того, що при киданні кубика з'явиться 4, завжди дорівнює 1/6, незалежно від того, яке число з'явилося під час попереднього кидання. Період випадкових чисел, тобто кількість випадкових чисел, згенерованих перед повторенням ряду, також дуже важливий. Найкраще коли він є якомога довшим чи принаймні його не можна перебрати за термін більший ніж термін служби генератора випадкових чисел.

Існує багато публікацій на тему тестування на випадковість, а також відповідних стандартів (FIPS 141-2, FIPS 186-4, ANSI X9.17, NIST SP 800-22).

Псевдовипадкові генератори, які схвалені NIST.

У документі 2010 року NIST SP 800-22 «A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications» представлено 15 тестів на випадковість, а також 9 генераторів, що їх вдало проходять. Запропоновані тести базуються на наближені вихідної генерації нормальним розподілом та застосуванні критерію χ^2.

Тести на псевдовипадковість (стор. 23 документу):

1. The Frequency (Monobit) Test
2. Frequency Test within a Block
3. The Runs Test
4. Tests for the Longest-Run-of-Ones in a Block
5. The Binary Matrix Rank Test
6. The Discrete Fourier Transform (Spectral) Test
7. The Non-overlapping Template Matching Test
8. The Overlapping Template Matching Test
9. Maurer's "Universal Statistical" Test
10. The Linear Complexity Test
11. The Serial Test
12. The Approximate Entropy Test

13. The Cumulative Sums (Cusums) Test
14. The Random Excursions Test
15. The Random Excursions Variant Test

Генератори (стор. 115 документу):

1. Linear Congruential Generator (LCG)
2. Quadratic Congruential Generator I (QCG-I)
3. Quadratic Congruential Generator II (QCG-II)
4. Cubic Congruential Generator II (CCG)
5. Exclusive OR Generator (XORG)
6. Modular Exponentiation Generator (MODEXPG)
7. Secure Hash Generator (G-SHA1)
8. Blum-Blum-Shub (BBSG)
9. Micali-Schnorr Generator (MSG)

Часто у статтях, коли проводять тести стосовно генерації псевдопростих (простих) чисел, проводять порівняння нових способів генерації з генератором BBS, останній з яких виступає певною еталонною моделлю. Розглянемо його.

Візьмемо $p, q -$ різні великі прості числа виду $4k + 3$, $n = pq$. Початкове значення r_0 вибирається довільним чином, а вихідна послідовність x_i обчислюється за таким правилом:

$$r_i = {r_{i-1}}^2 mod\ n,$$

$x_i = r_i mod\ 2$, тобто x_i є останнім бітом числа r_i.

Варто зазначити, що можливість вгадувати біти вихідної послідовності генератора BBS еквівалентна задачі факторизації, яка на практиці має принаймні субекспоненційну складність.

Також існує байтова модифікація BBS генератора, яка обчислює вихідну послідовність за такою формулою:

$x_i = r_i mod\ 256$, тобто x_i дорівнює восьми молодшим бітам числа r_i.

Детальніше можна ознайомитись за посиланням:
https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-22r1a.pdf.

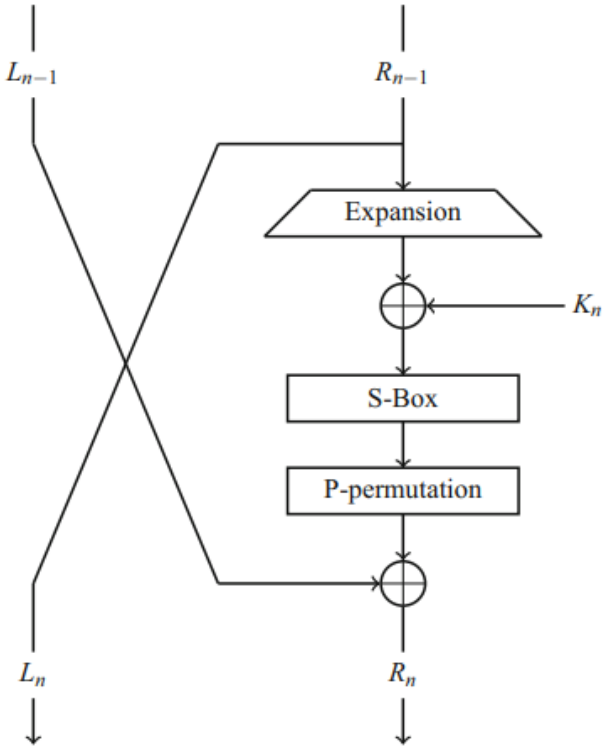
Класична будова справжнього генератора випадкових чисел у смарт-картках.

Існує багато різних способів генерувати випадкові числа за допомогою програмного забезпечення. Однак, оскільки пам’ять у смарт-картках дуже обмежена, а час, необхідний для виконання обчислень, має бути якомога коротшим, то кількість варіантів вибору обмежена. На практиці використовуються лише ті генератори, які використовують функції, що вже наявні в операційній системі (ПВЧ входять до складу мікропроцесора), оскільки вони вимагають дуже мало додаткового програмного коду. Важливо, щоб генератор був побудований таким чином, щоб послідовність випадкових чисел не була однаковою для кожного запиту (сеансу). Це реалізувати не так просто, адже для збереження нового початкового значення наступного сеансу потрібен доступ на запис до EEPROM. Оперативна пам’ять не підходить для цієї мети, оскільки для збереження вмісту їй потрібне живлення. Одним із можливих засобів атаки було б багаторазове генерування випадкових чисел поки EEPROM не вийдє з ладу. Теоретично це призводить до того, що під час кожного сеансу з’являється однакова послідовність випадкових чисел, що робить їх передбачуваними. Цей тип атаки можна легко запобігти, створивши відповідну частину EEPROM як кільцевий буфер і заблокувавши всі подальші дії, як тільки виникає помилка запису. Тут також виникає питання довжини циклу буфера, адже короткі цикли можна зламувати, а нескінченні реалізувати неможливо.

Зазвичай генератори засновані на регістрах зсуву з лінійним зворотним зв'язком (LFSR), які здатні генерувати нове псевдовипадкове значення кожного такту, але вони детерміновані в часі та зазвичай не використовуються для критичних функцій безпеки. Якщо криптографічні алгоритми потребують дійсно випадкових значень, використовується апаратний (справжній) генератор випадкових чисел (АГПВЧ, hardware random number generator, true random number generator). Якщо якість випадкової величини менш важлива, тоді використовують генератор псевдовипадкових чисел. У деяких захищених мікропроцесорах доступні лише генератори

псевдовипадкових чисел. У такому випадку для отримання випадкових значень використовують механізми, які об'єднують випадкове початкове число (random seed, яке може бути вставлене в чіп під час виробництва коли ОС встановлена) із псевдовипадковими значеннями. Прикладом такого генератору є генератор, який представлено у стандарті ANSI X9.17 і засновано на $3DES(x)=E_{k_1}(D_{k_2}(E_{k_1}(x)))$, де x – 64-бітовий вектор, E_k, D_k – функції шифрування та дешифрування DES з 56-бітовим ключем k відповідно. Генератор працює таким чином:

1. Знаходимо $x = 3DES(D)$, де D – якісь дані (та\або час);
2. Знаходимо псевдовипадкове число $R = 3DES(x \oplus S)$, де S – 64-бітове випадкове число (random seed);
3. Відновлюємо $S = 3DES(x \oplus R)$.

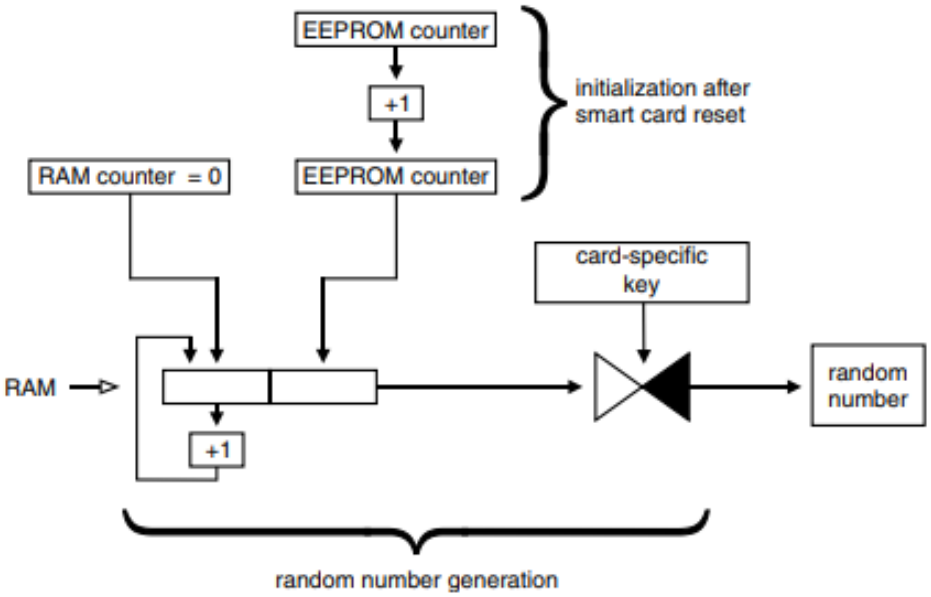


Зауваження: по суті можна використовувати будь-який алгоритм шифрування для генерації, але обмеження за пам'яттю, часом генерації, і відповідні стандарти (наприклад, ANSI X9.17).

Стирання та запис восьми байтів у EEPROM займає близько 14 мс, а виконання алгоритму DES займає близько 17 мс на частоті 3,5 МГц, якщо він реалізований програмно. Час обробки, що залишився, є незначним. Тобто картці потрібно близько 31 мс для того, щоб згенерувати випадкове число. Однак, якщо алгоритм DES обчислюється апаратно (зі швидкістю 0,1 мс/блок), то випадкове число може бути згенеровано за 14,4 мс.

Існують і інші підходи, коли початкову генерацію записують до EEPROM, а вже наступні генерації до оперативної пам’яті, що пришвидшує роботу генератора. Однак недоліком є те, що такий генератор використовує кілька байтів оперативної пам’яті на час сеансу. Статистична якість цього генератора псевдовипадкових чисел не дуже гарна, але цілком достатня для звичайних процедур аутентифікації смарт-карт.

Основне значення таких процедур полягає в уникненні генерації випадкових чисел з короткими циклами повторення, оскільки це дозволяє скомпрометувати аутентифікацію через відтворення повідомлень з попередніх сеансів.



Зауваження: стандарт FIPS 140-2 рекомендує модулям безпеки перевіряти свої вбудовані генератори випадкових чисел після кожної генерації за допомогою статистичних тестів. Тільки після успішного завершення цих тестів генератор випадкових чисел слід випускати для подальшого використання.

Зауваження: 22 березня 2019 року з'явився новий стандарт, а саме FIPS 140-3, який замінює попередню версію стандарту

(https://csrc.nist.gov/publications/detail/fips/140/3/final).

Зауваження: сучасні операційні системи смарт-карт рідко включають перевірку за всіма тестами, оскільки передбачається, що через детермінований характер генератора псевдовипадкових чисел статистика згенерованих випадкових чисел істотно не зміниться.

Зауваження: описи більшості сучасних АГПВЧ є комерційною таємницею, адже вони використовуються у воєнних цілях.

Статистичні тести перевірки простоти (псевдопростоти) чисел, які реалізовані у лабораторній.

- *Критерій перевірки рівномірності знаків*

  Послідовність $\{x_i\}$ задовільняє умові рівномірності знаків, якщо кожна x_i розподілена рівноймовірно на множині усіх можливих значень. Тести на рівноймовірність знаків рекомендують використовувати у першу чергу, адже інакше можна застосувати частотний аналіз і легко все зламати.

  Розглянемо байтову послідовність $\{x_i\}, i = \overline{1,m}$ та формуємо послідовність H_0: всі байти рівноймовірні. Критерій побудовано так:

  1) Обчислюємо статистику $\chi^2 = \sum_{i=0}^{255} \frac{(v_i - n_i)^2}{n_i}$, де v_i – число байтів i, що спостерігається у послідовності, а n_i – очікуване число байтів i у послідовності за умови, що H_0 справедлива, тобто $n_i = \frac{m}{256}, i = \overline{1,m}$.
  2) Обчислюємо граничне значення $\chi^2_{1-\alpha}$=$\sqrt{2l}\, Z_{1-\alpha} + l$, що відповідає рівню довіри α при $l = 255$, де $Z_{1-\alpha}$ – $(1-\alpha)$-квантиль стандартного нормального розподілу.
  3) Перевіряємо $\chi^2 \leq \chi^2_{1-\alpha}$: якщо нерівність виконується, то H_0 виконується, інакше – ні.

- *Критерій перевірки незалежності знаків*

  Послідовність $\{x_i\}$ задовільняє умові незалежності знаків, якщо ймовірність прийняти деяке значення для x_i не залежить від того, які значення прийняли попередні члени послідовності. Оскільки перевірка такої умови є вкрай важкою, розглянемо незалежність від попереднього знаку.

  Розглянемо байтову послідовність $\{x_i\}, i = \overline{1,m}$, а також пари $(x_{2i-1}, x_{2i}), i = \overline{1,n}$, де $n = \left[\frac{m}{2}\right]$. Гіпотеза H_0: кожний наступний байт незалежить від попереднього. Критерій побудовано так:

  1) Обчислюємо статистику $\chi^2 = n\left(\sum_{i,j=0}^{255} \frac{{v_{ij}}^2}{v_i \alpha_i} - 1\right)$, де v_{ij} – кількість появи пари (i,j), v_i (α_i) – кількість появи байта i (j) на першому (другому) місці.

2) Обчислюємо граничне значення $\chi^2_{1-\alpha}=\sqrt{2l}\,Z_{1-\alpha}+l$, що відповідає рівню довіри α при $l=255^2$, де $Z_{1-\alpha}$ – $(1-\alpha)$-квантиль стандартного нормального розподілу.
3) Перевіряємо $\chi^2 \le \chi^2_{1-\alpha}$: якщо нерівність виконується, то H_0 виконується, інакше – ні.

- *Критерій перевірки однорідності знаків*

Послідовність $\{x_i\}$ задовільняє умові однорідності знаків, якщо на довільному своєму фрагменті послідовність веде себе однаково (їх розподіл однаковий). Оскільки перевірка такої умови є вкрай важкою, то на практиці послідовність розбивають на окремі інтервали та перевіряють співпадіння розподілів на цих інтервалах. Важливо підкреслити, що для виконання умови однорідності знати який саме розподіл будуть мати ті чи інші значення x_i не є важливим.

Розглянемо байтову послідовність $\{x_i\}, i=\overline{1,m}$, що розбивається на r відрізків довжиною $m'=\left[\frac{m}{r}\right]$ (вважаємо, що всі відрізки мають однакову довжину), $n=m'r$ – загальне число байтів, що використовується. Гіпотеза H_0: на довільному своєму фрагменті послідовність веде себе однаково. Критерій побудовано так:

1) Обчислюємо статистику $\chi^2=n\left(\sum_{i=0}^{255}\sum_{j=0}^{r-1}\frac{{v_{ij}}^2}{v_i\alpha_i}-1\right)$, де v_{ij} – кількість появи байта i у відрізку j, $v_i=\sum_{j=0}^{r-1}v_{ij}$, $\alpha_i=\sum_{i=0}^{255}v_{ij}=\ m'$.
2) Обчислюємо граничне значення $\chi^2_{1-\alpha}=\sqrt{2l}\,Z_{1-\alpha}+l$, що відповідає рівню довіри α при $l=255(r-1)$, де $Z_{1-\alpha}$ – $(1-\alpha)$-квантиль стандартного нормального розподілу.
3) Перевіряємо $\chi^2 \le \chi^2_{1-\alpha}$: якщо нерівність виконується, то H_0 виконується, інакше – ні.

Зауваження: ще одною гарною практикою є застосування програм стиснення файлів для визначення псевдовипадковості чисел (LZMA, DEFLATE, BWT та інші), адже можна показати, що ступінь стиснення обернено пов'язана з випадковістю набору згенерованих чисел.

ЙМОВІРНІСНІ ТЕСТИ ПЕРЕВІРКИ ПРОСТОТИ ЧИСЕЛ

Для перевірки чисел на простоту (псевдопостоту) використовують тести Ферма, Соловея-Штрассена та Міллера-Рабіна. Перший тест є тестом типу Лас-Вегас, а останніх два – тестами типу Монте-Карло.

- *Тест Ферма*

  Вхід: непарне число n.

  Додатковий вхід: x, таке, що $gcd(x, n) = 1$.

  Перевіряємо виконання порівняння:

  $$x^{n-1} \equiv 1(mod\ n).$$

  Вихід: якщо порівняння виконується, то «n – складене», інакше – «не знаємо».

- *Тест Соловея-Штрассена*

  Вхід: непарне число n.

  Додатковий вхід: x, таке, що $gcd(x, n) = 1$.

  Перевіряємо виконання критерію Ойлера:

  $$\left(\frac{x}{n}\right) \equiv x^{\frac{n-1}{2}}(mod\ n), \text{ де } \left(\frac{x}{n}\right) \text{ – символ Ойлера.}$$

  Вихід: якщо критерій виконується, то «n – просте», інакше – «n – складене».

- *Тест Міллера-Рабіна*

  Вхід: непарне число n: $n - 1 = 2^s t$, де t – непарне.

  Додатковий вхід: x, таке, що $gcd(x, n) = 1$.

  1) Обчислюємо $y_0 = x^t mod\ n$. Якщо $y_0 \equiv \pm 1(mod\ n)$, то вихід: «n – просте», інакше йдемо до пункту 2.
  2) Обчислюємо $y_j = {y_{j-1}}^2 mod\ n$ поки не виявиться, що $j = s - 1$, або $y_j \equiv \pm 1(mod\ n)$ для деякого $j < s - 1$.
  3) Якщо для деякого $j < s - 1$ виконується $y_j \equiv -1(mod\ n)$, то припиняємо роботу алгоритму з виходом: «n – просте», інакше «n – складене».

Приклад генерації та виконання тестів:

```
Генерация последовательности генератором BBS: 100%

Вывод первых 128 байт:

193 41 205 12 137 129 67 242 171 135 16 118 83 64 89 177 10 28 8 210 121 46 75 135 147 147 140 39
2 162 230 189 246 211 87 75 28 63 9 148 204 202 97 144 242 103 81 28 136 118 112 197 114 189 246
224 253 61 167 171 38 89 230 174 218 229 95 22 47 187 229 175 10 83 184 91 57 16 227 73 186 237
241 38 44 40 174 203 102 33 173 107 77 86 199 173 10 2 228 99 55 86 88 93 32 216 253 114 168 219
110 121 114 170 17 13 119 72 183 52 210 248 166 80 198 90 213 1

Генерация последовательности генератором BBS_bytes: 100%

Вывод первых 128 байт:

118 148 218 46 120 175 107 28 237 107 179 26 116 72 91 105 86 199 232 228 92 212 61 138 218 29
199 12 27 159 245 151 105 199 191 184 213 39 98 177 180 84 33 10 202 215 108 56 64 146 42 99 97
16 136 89 96 61 27 80 4 158 255 118 33 106 223 200 141 64 184 28 16 28 147 72 97 127 192 102 19
17 176 74 207 241 120 41 1 67 182 130 240 126 141 162 95 139 177 157 98 36 65 8 240 3 223 188 198
219 237 137 71 34 124 80 239 30 8 161 71 77 40 128 86 194 91 80
```

Для тестов на однородность, равномерность и независимость использовались последовательности длиной 1000000 байт.

Результат: '+' = 'прошел тест', '-' = 'не прошел тест'			
Альфа = 0.01	=> Хи^2 = 307.536	=> Хи^2 = 65863.938	=> Хи^2 = 2452.609
Генератор	Равновероятность	Независимость	Однородность
BBS	258.739 <= Хи^2 => +	64964.008 <= Хи^2 => +	2279.169 <= Хи^2 => +
BBS_bytes	289.326 <= Хи^2 => +	65185.204 <= Хи^2 => +	2293.558 <= Хи^2 => +

Результат: '+' = 'прошел тест', '-' = 'не прошел тест'			
Альфа = 0.05	=> Хи^2 = 292.146	=> Хи^2 = 65618.174	=> Хи^2 = 2406.438
Генератор	Равновероятность	Независимость	Однородность
BBS	258.739 <= Хи^2 => +	64964.008 <= Хи^2 => +	2279.169 <= Хи^2 => +
BBS_bytes	289.326 <= Хи^2 => +	65185.204 <= Хи^2 => +	2293.558 <= Хи^2 => +

```
Результат теста Миллера-Рабина для result_768_BBS: составное

Результат теста Миллера-Рабина для result_768_BBS_Byte: составное

Результат теста Миллера-Рабина для result_1024_BBS: составное

Результат теста Миллера-Рабина для result_1024_BBS_Byte: составное

Результат теста Миллера-Рабина для 41: простое
```

Висновки:

У даній лабораторній роботі я дослідила будову інтелектуальних карток, розглянула наявні смарт-картки, а також розглянула генератори, що запропоновані у різних стандартах, зокрема NIST SP 800-22. Реалізувала один з найкращих алгоритмів

генерації псевдовипадкових чисел – BBS (Blum-Blum-Shub) та його байтову модифікацію BBSByte. Розглянула тести на перевірку псевдовипадковості чисел, ймовірності тести перевірки чисел на простоту, а також реалізувала тести перевірки рівноймовірності знаків, перевірки незалежності знаків, перевірки однорідності знаків та тест Міллера-Рабіна – для чого використовувала мову C++ та платформу .NET – і протестувала BBS та BBSByte генератори на них. Окрім цього, я дослідила стандарти ISO/IEC 7816, FIPS 141-2, FIPS 186-4, ANSI X9.17. Якщо говорити про справжній генератор випадкових чисел, який реалізовано у смарт-картках, то це 3DES, де використовують початкове число (random seed), яке може бути вставлене в чіп під час виробництва коли ОС є встановленою.

<u>У лабораторній роботі я реалізувала BBS та BBSByte генератори таким чином (generators.h):</u>

```
#pragma once
#include <stdlib.h>
#include <iostream>
#include <fstream>
#include <string>
using namespace System::Globalization;
using namespace System::Numerics;
struct byte  // Структура для удобства (по суте char но выводится как int)
{
	unsigned char value;

	byte()
	{
		this->value = 0;
	}

	byte(const int& value)
	{
		this->value = (unsigned char)value;
	}
```

```
        operator unsigned int()
        {
            return (unsigned int)value;
        }

        byte& operator =(const unsigned int &value)
        {
            this->value = (unsigned char)value;
            return *this;
        }

        byte& operator =(const int& value)
        {
            this->value = (unsigned char)value;
            return *this;
        }
};
// Начальные значения для функций рандома
unsigned int seedBBS;
unsigned int seedBBSByte;

static ref struct BBSSeed
{
        static BigInteger n;
        static BigInteger r;
};
// Инициализация начальных значений
void initRandomGenerators()
{
```

```
        seedBBS = 54287434u;

        seedBBSByte = 2347532544u;

        BBSSeed::n = BigInteger::Parse("D5BBB96D30086EC484EBA3D7F9CAEB07",
NumberStyles::AllowHexSpecifier) *

                BigInteger::Parse("425D2B9BFDB25B9CF6C416CC6E37B59C1F",
NumberStyles::AllowHexSpecifier);

        BBSSeed::r = BigInteger::Parse("675215CC3E227D321097E1DB049F1",
NumberStyles::AllowHexSpecifier);

}

byte randBBS()

{

        for (int i = 0; i < 8; i++)

        {

                BBSSeed::r = BigInteger::ModPow(BBSSeed::r, 2, BBSSeed::n);

                seedBBS = static_cast<int>(BBSSeed::r % 2) ? seedBBS << 1 | 1u : seedBBS << 1 & ~1u;

        }

        return static_cast<byte>(seedBBS);

}

byte randBBSByte()

{

        BBSSeed::r = BigInteger::ModPow(BBSSeed::r, 2, BBSSeed::n);

        seedBBSByte = static_cast<unsigned int>(BBSSeed::r % 256);

        return static_cast<byte>(seedBBSByte);

}
```

Реалізація тесту Мілера-Рабіна (millerRabin.h):

```
#pragma once

#include <sstream>

#include <iostream>

#include <string>

using namespace System::Globalization;

using namespace System::Numerics;
```

```
using namespace System;
using namespace std;
BigInteger bigRand(unsigned bitNuber)  // Генерация случайных bitNuber-bitных чисел
{
	bitNuber /= 8;
	// Таблицы шестнадцатеричных чисел
	static unsigned char hexTable[] = { '0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F' };
	if (bitNuber)  // Если больше нуля
	{
		ostringstream oss;  // Поток вывода
		for (size_t i = 0; i < bitNuber; ++i)  // Проход по цифрам
			oss << hexTable[rand() % 16];  // Добавление случайной цифры
		string str(oss.str());  //    Строка в число

		String^ system_str = gcnew String(str.c_str());
		return BigInteger::Parse(system_str, NumberStyles::AllowHexSpecifier);  // Вернуть число
	}
	else
		return BigInteger::Zero;  // Вернуть ноль
}
BigInteger bigRand(BigInteger min, BigInteger max, unsigned bitNuber)  // Генерация случайных bitNuber-bitных чисел от min до max
{
	return BigInteger::Add(min, BigInteger::Remainder(BigInteger(bigRand(bitNuber)), BigInteger::Subtract(max, min)));
}
bool millerRabin(BigInteger p, unsigned bitNuber, int k = 15)
{
	if (BigInteger::Compare(p, BigInteger(2)) == -1 || // Проверка на то, что p < 2
		(BigInteger::Compare(p, BigInteger(2)) != 0 && BigInteger::Remainder(p, BigInteger(2)) == BigInteger(0)) || // Проверка на то, что p четное
```

```
		(BigInteger::Compare(p, BigInteger(3)) != 0 && BigInteger::Remainder(p, BigInteger(3)) == BigInteger(0)) || // Проверка на то, что p кратно 3

		(BigInteger::Compare(p, BigInteger(5)) != 5 && BigInteger::Remainder(p, BigInteger(5)) == BigInteger(0))) // Проверка на то, что p кратно 5

			return false;

	// Разложение на d * 2 ^ s

	BigInteger d = BigInteger::Subtract(p, BigInteger(1));  // P - 1

	while (BigInteger::Remainder(d, BigInteger(2)) == BigInteger(0))  // Пока d четное

		d = BigInteger::Divide(d, BigInteger(2));  // Делим на 2


	for (int i = 0; i < k; i++)   // k итераций

	{

		BigInteger x = bigRand(BigInteger(1), BigInteger::Subtract(p, BigInteger(1)), bitNuber), t = d;  // Случайное число от 1 до p - 1;

		BigInteger xr = BigInteger::ModPow(x, t, p);  // x ^ d

		if (xr != BigInteger::Subtract(p, BigInteger(1)) || xr != BigInteger(1))  // Проверяем условия псевдопростоты

			return true;  // Возврат (простое)

		while (t != BigInteger::Subtract(p, BigInteger(1)) && BigInteger::Compare(xr, BigInteger(1)) != 0 && xr != BigInteger::Subtract(p, BigInteger(1)))  // Пока xr не равно 1 или -1 по модулю p и t != p - 1 (от 1 до s)

		{

			xr = BigInteger::ModPow(xr, 2, p);  // xr = x ^ (d * 2 ^ r)

			t = BigInteger::Multiply(t, BigInteger(2));  // Домножает d на 2 (r принадлежит [1, s])

		}

		if (xr != BigInteger::Subtract(p, BigInteger(1)) && BigInteger::Remainder(t, BigInteger(2)) == BigInteger(0))  // Проверяем условия псевдопростоты

			return false;  // Возврат (составное)

	}

	return true;  // Возврат (простое)

}
```

Файл tests.h:

```
#pragma once
```

```
#include <math.h>
#include <map>
typedef unsigned int uint;
typedef unsigned short ushort;
using namespace std;
struct TestResult  // Результата теста
{
	double x2;  // X^2
	bool right;  // + / -
};
const uint r = 10u;
// Тест равномерности
TestResult testEquiprobability(byte* sequence, const uint m, uint l, double x21a)
{
	const auto max = 255u;
	auto nj = m / (double)max;
	auto x2 = 0.0;
	uint v[max + 1];
	memset(v, 0, (max + 1) * sizeof(uint));

	for (uint i = 0; i < m; i++)
		v[sequence[i]]++;
	for (int j = 0; j <= max; j++)
		x2 += pow(v[j] - nj, 2.0) / nj;

	return { x2, x2 <= x21a };
}
// Тест независимости
TestResult testIndependence(byte* sequence, uint m, uint l, double x21a)
{
```

```
	auto n = m / 2u;
	const auto max = 255u;
	auto x2 = 0.0;
	uint vji[max + 1][max + 1];
	uint v[max + 1];
	uint a[max + 1];

	memset(vji, 0, (max + 1) * (max + 1) * sizeof(uint));
	memset(v, 0, (max + 1) * sizeof(uint));
	memset(a, 0, (max + 1) * sizeof(uint));

	for (uint i = 1; i < n; i++)
	{
		vji[sequence[2 * i - 1]][sequence[2 * i]]++;
		v[sequence[2 * i - 1]]++;
		a[sequence[2 * i]]++;
	}
	for (int i = 0; i <= max; i++)
		for (int j = 0; j <= max; j++)
			if (v[i] * a[j] > 0.0)
				x2 += pow(vji[i][j], 2.0) / (v[i] * a[j]);
	x2 = n * (x2 - 1.0);

	return { x2, x2 <= x21a };
}
// Тест однородности
TestResult testUniformity(byte* sequence, uint m, uint l, double x21a)
{
	auto mi = m / r;
	const auto max = 255u;
```

```
	auto x2 = 0.0;
	uint vji[r][max + 1];
	uint v[max];

	memset(vji, 0, r * (max + 1) * sizeof(uint));
	memset(v, 0, (max + 1) * sizeof(uint));

	for (auto j = 0u; j < r; j++)
	{
		for (auto i = 0u; i < mi; i++)
		{
			vji[j][sequence[j * mi + i]]++;
			v[sequence[j * mi + i]]++;
		}
	}
	for (auto j = 0u; j <= max; j++)
		for (auto i = 0u; i < r; i++)
			if (((double)v[j] * mi) > 0)
				x2 += pow(vji[i][j], 2.0) / ((double)v[j] * mi);
	x2 = m * (x2 - 1.0);

	return { x2, x2 <= x21a };
}
```

Файл Lab2.cpp:

```
#include "generators.h"
#include "tests.h"
#include "millerRabin.h"
#include <string>
#include <iostream>
#include <fstream>
```

```
#include <iomanip>

#include <locale.h>

#include <sstream>

using namespace std;

using namespace System;

// Для отображения прогренсса каждые frequancy итераций

#define show(finish, what) if (i % (finish / 100) == 0 || i == finish - 1) what

// Вывод прогресса в процентах

void outputProgress(ofstream & outputFile, string message, float progress);

// Вывод последовательностей байт

void outputSequences(ofstream & outputFile, byte * sequences);

// Вывод заголовка таблицы

void outputHeader(ofstream & outputFile, double a, double x21ae, double x21ai, double x21au);

// Вывод строки таблицы

void outputRow(ofstream & outputFile, string name, TestResult equiprobability, TestResult independence,
TestResult uniformity);

// Вывод прогресса в процентах (шаблон)

template <typename Stream> void outputProgressTemplate(Stream& stream, string message, float progress);

// Вывод последовательностей байт (шаблон)

template <typename Stream> void outputSequencesTemplate(Stream& stream, byte* sequences);

// Вывод заголовка таблицы (шаблон)

template <typename Stream> void outputHeaderTemplate(Stream &stream, double a, double x21ae, double
x21ai, double x21au);

// Вывод строки таблицы (шаблон)

template <typename Stream> void outputRowTemplate(Stream & stream, string name, TestResult
equiprobability, TestResult independence, TestResult uniformity);

int main()

{

	setlocale(LC_ALL, "rus");  // Корректный вывод кирилицы

	const size_t sequencesSize = 1000000;  // Количество байт в последовательностях

	initRandomGenerators();  // Инициализация генераторов стартовыми значениями
```

```
// Выделение памяти под последовательности
byte* sequenceBBS = new byte[sequencesSize];
byte* sequenceBBSByte = new byte[sequencesSize];
// Файл вывода результата
ofstream outputFile("./result.txt");
ostringstream oss_for_sequenceBBS_768_as_str, oss_for_sequenceBBSByte_768_as_str, oss_for_sequenceBBS_1024_as_str, oss_for_sequenceBBSByte_1024_as_str;  // Потоки вывода
// Заполнение массивов случайными байтами различных генераторов
for (size_t i = 0; i < sequencesSize; i++)
{
    sequenceBBS[i] = randBBS();
    if(i < 96) oss_for_sequenceBBS_768_as_str << sequenceBBS[i];
    if (i < 128) oss_for_sequenceBBS_1024_as_str << sequenceBBS[i];
    show(sequencesSize, outputProgress(outputFile, "Генерация последовательности генератором BBS: ", (i + 1.0f) / sequencesSize));
}
outputSequences(outputFile, sequenceBBS);
string sequenceBBS_768_as_str(oss_for_sequenceBBS_768_as_str.str());
string sequenceBBS_1024_as_str(oss_for_sequenceBBS_1024_as_str.str());
for (size_t i = 0; i < sequencesSize; i++)
{
    sequenceBBSByte[i] = randBBSByte();
    if (i < 96) oss_for_sequenceBBSByte_768_as_str << sequenceBBS[i];
    if (i < 128) oss_for_sequenceBBSByte_1024_as_str << sequenceBBS[i];
    show(sequencesSize, outputProgress(outputFile, "Генерация последовательности генератором BBS_bytes: ", (i + 1.0f) / sequencesSize));
}
outputSequences(outputFile, sequenceBBSByte);
string sequenceBBSByte_768_as_str(oss_for_sequenceBBSByte_768_as_str.str());
string sequenceBBSByte_1024_as_str(oss_for_sequenceBBSByte_1024_as_str.str());
String^ sequenceBBS_768_as_str_msdn = gcnew String(sequenceBBS_768_as_str.c_str());
```

```
String^ sequenceBBSByte_768_as_str_msdn = gcnew String(sequenceBBSByte_768_as_str.c_str());
String^ sequenceBBS_1024_as_str_msdn = gcnew String(sequenceBBS_1024_as_str.c_str());
String^ sequenceBBSByte_1024_as_str_msdn = gcnew String(sequenceBBSByte_1024_as_str.c_str());
BigInteger result_768_BBS, result_1024_BBS, result_768_BBS_Byte, result_1024_BBS_Byte;
result_768_BBS = BigInteger::Parse(sequenceBBS_768_as_str_msdn);
result_768_BBS_Byte = BigInteger::Parse(sequenceBBSByte_768_as_str_msdn);
result_1024_BBS = BigInteger::Parse(sequenceBBS_1024_as_str_msdn);
result_1024_BBS_Byte = BigInteger::Parse(sequenceBBSByte_1024_as_str_msdn);
auto max = 255;  // Максимальное значение байта
auto r = 10u;  // Количество делений массива
auto le = max;  // l для равномерности
auto li = pow(max, 2);  // l для независимости
auto lu = max * (r - 1);  // l для однородности

auto x21ae = sqrt(2.0 * le) * 2.3263479 + le;  // X^2 для равномерности
auto x21ai = sqrt(2.0 * li) * 2.3263479 + li;  // X^2 для независимости
auto x21au = sqrt(2.0 * lu) * 2.3263479 + lu;  // X^2 для однородности
// Тесты и вывод таблицы для a = 0.01
outputHeader(outputFile, 0.01, x21ae, x21ai, x21au);
outputRow(outputFile, "BBS", testEquiprobability(sequenceBBS, sequencesSize, le, x21ae),
        testIndependence(sequenceBBS, sequencesSize, li, x21ai), testUniformity(sequenceBBS,
sequencesSize, lu, x21au));
outputRow(outputFile, "BBS_bytes", testEquiprobability(sequenceBBSByte, sequencesSize, le, x21ae),
        testIndependence(sequenceBBSByte, sequencesSize, li, x21ai),
testUniformity(sequenceBBSByte, sequencesSize, lu, x21au));
x21ae = sqrt(2.0 * le) * 1.6448536 + le;
x21ai = sqrt(2.0 * li) * 1.6448536 + li;
x21au = sqrt(2.0 * lu) * 1.6448536 + lu;
// Тесты и вывод таблицы для a = 0.05
outputHeader(outputFile, 0.05, x21ae, x21ai, x21au);
outputRow(outputFile, "BBS", testEquiprobability(sequenceBBS, sequencesSize, le, x21ae),
```

```
        testIndependence(sequenceBBS, sequencesSize, li, x21ai), testUniformity(sequenceBBS,
sequencesSize, lu, x21au));

    outputRow(outputFile, "BBS_bytes", testEquiprobability(sequenceBBSByte, sequencesSize, le, x21ae),

        testIndependence(sequenceBBSByte, sequencesSize, li, x21ai),
testUniformity(sequenceBBSByte, sequencesSize, lu, x21au));


    cout << "Результат теста Миллера-Рабина для result_768_BBS: ";
    if (millerRabin(result_768_BBS, 768) == true) cout << "простое" << endl;
    else cout << "составное" << endl;
    cout << "Результат теста Миллера-Рабина для result_768_BBS_Byte: ";
    if (millerRabin(result_768_BBS_Byte, 768) == true) cout << "простое" << endl;
    else cout << "составное" << endl;
    cout << "Результат теста Миллера-Рабина для result_1024_BBS: ";
    if (millerRabin(result_1024_BBS, 768) == true) cout << "простое" << endl;
    else cout << "составное" << endl;
    cout << "Результат теста Миллера-Рабина для result_1024_BBS_Byte: ";
    if (millerRabin(result_1024_BBS_Byte, 768) == true) cout << "простое" << endl;
    else cout << "составное" << endl;
    cout << "Результат теста Миллера-Рабина для 41: ";
    if (millerRabin(BigInteger::Parse("41"), 1024) == true) cout << "простое" << endl;
    else cout << "составное" << endl;


    // Закрытие файла результата
    outputFile.close();
    // Освобождение памяти
    delete[] sequenceBBS;
    delete[] sequenceBBSByte;
    cin.get();
    cin.get();
    return 0;
}
```

```
void outputProgress(ofstream& outputFile, string message, float progress)
{
	cout << "\r";
	outputProgressTemplate(cout, message, progress);
	if (progress == 1.0f)
		outputProgressTemplate(outputFile, message, progress);
}
void outputSequences(ofstream &outputFile, byte* sequences)
{
	outputSequencesTemplate(cout, sequences);
	outputSequencesTemplate(outputFile, sequences);
}
void outputHeader(ofstream& outputFile, double a, double x21ae, double x21ai, double x21au)
{
	outputHeaderTemplate(cout, a, x21ae, x21ai, x21au);
	outputHeaderTemplate(outputFile, a, x21ae, x21ai, x21au);
}
void outputRow(ofstream& outputFile, string name, TestResult equiprobability, TestResult independence,
TestResult uniformity)
{
	outputRowTemplate(cout, name, equiprobability, independence, uniformity);
	outputRowTemplate(outputFile, name, equiprobability, independence, uniformity);
}
template <typename Stream> void outputProgressTemplate(Stream& stream, string message, float progress)
{
	stream << message << fixed << setprecision(0) << progress * 100.0f << "%" << (progress == 1.0f ? "\n" :
"");
}
template <typename Stream> void outputSequencesTemplate(Stream& stream, byte *sequences)
{
	stream << "Вывод первых 128 байт:\n";
```

```
	for (int i = 0; i < 128; i++)
		stream << sequences[i] << (i == 127 ? "\n\n" : " ");
}
template <typename Stream> void outputRowTemplate(Stream& stream, string name, TestResult equiprobability, TestResult independence, TestResult uniformity)
{
	stringstream col1, col2, col3;
	col1 << " | " << fixed << setprecision(3) << equiprobability.x2 << (equiprobability.right ? " <= Хи^2 => +" : " > Хи^2 => -");
	col2 << " | " << fixed << setprecision(3) << independence.x2 << (independence.right ? " <= Хи^2 => +" : " > Хи^2 => -");
	col3 << " | " << fixed << setprecision(3) << uniformity.x2 << (uniformity.right ? " <= Хи^2 => +" : " > Хи^2 => -");
	stream << "| " << setw(12) << left << name
		<< setw(31) << col1.str() << setw(31) << col2.str() << setw(31) << col3.str() << " |" << endl <<
		"-------------------------------------------------------------------------------------------------------------" << endl;
}
template <typename Stream> void outputHeaderTemplate(Stream &stream, double a, double x21ae, double x21ai, double x21au)
{
	stringstream headerText;
	headerText << "Альфа = " << fixed << setprecision(2) << a << " | => Хи^2 = "
		<< setw(19) << left << fixed << setprecision(3) << x21ae << "| => Хи^2 = "
		<< setw(19) << left << fixed << setprecision(3) << x21ai << "| => Хи^2 = "
		<< setw(19) << left << fixed << setprecision(3) << x21au << "";
	stream << endl;
	stream << "-------------------------------------------------------------------------------------------------------------" << endl;
	stream << "| Результат: '+' = 'прошел тест', '-' = 'не прошел тест'                                                    |" << endl;
	stream << "-------------------------------------------------------------------------------------------------------------" << endl;
	stream << "| " << setw(43) << left << headerText.str() << "|" << endl;
```

```
	stream << "-----------------------------------------------------------------------------------------------------"
<< endl;

	stream << "| Генератор    | Равновероятность          | Независимость             | Однородность
|" << endl;

	stream << "-----------------------------------------------------------------------------------------------------"
<< endl;

}
```