MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Java Cryptography Architecture Educational Materials

Master's Thesis

BC. JAN SEKANINA

Brno, Spring 2026

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Java Cryptography Architecture Educational Materials

Master's Thesis

BC. JAN SEKANINA

Advisor: Łukasz Michał Chmielewski, PhD

Department of Computer Systems and Communications

Brno, Spring 2026

FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

I utilized LLMs to proofread my writing for grammatical accuracy, logical consistency, and academic tone. Additionally, I applied these tools to image enhancement, code refactoring, text refinement, and brainstorming.

I declare that I used these tools in accordance with the principles of academic integrity. I checked the content and took full responsibility for it.

Bc. Jan Sekanina

Advisor: Łukasz Michał Chmielewski, PhD

Acknowledgements

I would like to thank my advisor, Łukasz Michał Chmielewski, PhD, for his insights and continuous feedback during my writing. I would also like to thank my family and friends for their support.

Abstract

This thesis aims to develop comprehensive educational materials focused on the latest advancements in the Java ecosystem and modern cryptographic standards. With the release of Java 25 (LTS), there is a need for updated curricula that integrate Post-Quantum Cryptography (PQC) algorithms. This research introduces students to the Java Cryptography Architecture (JCA), exploring its provider-based model and the management of cryptographic primitives. The primary output is a structured seminar, including instructional slides and interactive tasks designed to encourage active learning. Among the solved tasks, students will encounter topics such as random number generation, keyed and unkeyed hash functions, symmetric cryptography, password-based key derivation functions, and asymmetric cryptography. Furthermore, the work provides versatile assessment tools, namely exercises adaptable for both examination and homework assignments.

Keywords

Cryptography, Java, Bouncy Castle, PQC, Education, Security

Contents

Introduction and Motivation **1**

1 Cryptographic Primitives **4**
1.1 Randomness 4
1.2 Hash Functions 6
1.3 Cryptographically Secure Hash Functions 6
1.4 Keyed Hashing (MACs) 7
1.5 PBKDF2 7
1.6 Similarity Hash Functions 8
1.7 Stream Ciphers 8
1.8 Block Ciphers 9
1.8.1 Modes of Operation 10
1.9 Asymmetric Cryptography 13
1.9.1 Encryption Schemes and Key Encapsulation . . 14
1.9.2 Signature Schemes 15

2 Java **18**
2.1 Java Sandbox 18
2.2 Java Development Kit (JDK) 19
2.3 Java Cryptography Architecture (JCA) 19
2.4 Java Virtual Machine (JVM) 20
2.4.1 Java Providers 20
2.4.2 Sun Providers 20
2.4.3 Bouncy Castle 21
2.5 Java 25 22

3 Seminar Content **23**
3.1 Seminar Preparation 23
3.2 Seminar Difficulty and Time Structure 24
3.3 Slide Presentation and Seminar Flow 24
3.3.1 Introduction and Context Setup 25
3.3.2 Cryptographic Foundations in Java 25
3.3.3 Developer Best Practices and Tooling 25
3.3.4 Task Execution and Post-Task Discussion (Q&A) 26
3.3.5 Conclusion and Feedback 26
3.3.6 Slide Notes 27

3.4 Code Repository Structure 27

4 Seminar Exercises **30**
4.1 JCA Providers Task 30
4.2 Random Bytes Generators 32
4.3 Symmetric Cryptography 36
4.3.1 Block Ciphers and Modes of Operation (Tasks 1&2) 37
4.3.2 Contemporary Lightweight Cryptography: ASCON (Task 4) 38
4.3.3 Ciphertext Malleability and Integrity Attacks (Task 3) 38
4.3.4 Stream Ciphers and Data Streaming (Tasks 5&6) 39
4.4 Hashing and MACs 40
4.4.1 Cryptographic Hashing and Message Authentication 41
4.4.2 Similarity Hashing in Malware Analysis 41
4.5 Key Derivation Functions (KDFs) 42
4.6 Asymmetric Cryptography Tasks 44
4.7 Matching Signatures Task 47
4.8 Test / Homework Assignments 48
4.8.1 Hash Precomputing (Partial Preimage Attack) . 48
4.8.2 AES CTR Bit-Flipping 50
4.8.3 Cryptographic Misconfigurations (Code Review) 50
4.8.4 Deployment Recommendations 51

5 Pedagogical Approach **52**
5.1 Seminar Execution Variants 52
5.2 Conducting Programming Seminars in the Era of LLMs 53
5.3 Real-World Deployment and Iterative Adjustments . . . 54
5.4 Thematic Data Selection 55

6 Conclusions and Future Work **56**

Bibliography **57**

Index **63**

7 An appendix **63**

List of Figures

1.1 RNGs produce few unreliable bits from analog sources, whereas PRNGs expand those bits to a long stream of reliable bits. [7] 6
1.2 The CBC encryption mode [7]. 11
1.3 The CTR encryption mode. [7] 12
1.4 The GCM usage. [24] 14

2.1 Provider request recommendations [35]. 20
2.2 Selection of provider.[35] 21

3.1 Student Feedback Questionnaire 26
3.2 Toggling presentation notes using Beamer document class options. 27
3.3 Simplified directory tree of the seminar repository. 28

4.1 Conscrypt Provider Setup and Review Questions. [42] . . 31
4.2 JCA Providers Task Solution Output 32
4.3 JCA Providers: Answers 33
4.4 Random Number Generators: Questions 34
4.5 Constants in `java.util.Random` 35
4.6 Random Number Generators: Answers 36
4.7 Symmetric Cryptography: Questions 37
4.8 Symmetric Cryptography: Answers 40
4.9 Hashing & MACs: Questions 41
4.10 Hashing & MACs: Answers 43
4.11 Password-Based Key Derivation: Questions 43
4.12 Password-Based Key Derivation: Answers 45
4.13 Asymmetric Cryptography: Questions 45
4.14 Asymmetric Cryptography: Answers 46
4.15 Signature Matching: Discussion 48
4.16 Signature Matching: Takeaways 49

Introduction

The primary objective of this thesis is to evaluate recent advancements within the Java ecosystem, specifically focusing on the implementation of standard cryptographic primitives in *Java 25* (*LTS*). Furthermore, the research explores the integration of emerging *Post-Quantum Cryptography* (*PQC*) libraries into the modern Java landscape.

A central component of this work is the development of a comprehensive suite of educational resources, comprising practical seminar tasks and supplementary assessment materials suitable for both homework and examinations. While the selection of topics reflects current priorities in information security and cryptography, the curriculum is designed as a modular repository. This flexible structure allows instructors to curate content tailored to specific learning outcomes and standard seminar durations, such as a 100-minute instructional session.

The educational materials are engineered to be adaptable to varying levels of student proficiency in both Java programming and security concepts. It should be noted that this thesis does not provide introductory instruction on Java syntax or the theoretical basics of cryptography. Instead, it demonstrates the applied utility of the language, highlighting the architectural advantages and security features available to developers.

The practical component is structured into *seven submodules*, each focusing on a distinct cryptographic domain. Each module contains multiple subtasks and discussion questions designed to foster active student participation and deepen understanding of the subject matter.

Thesis outline

The thesis is divided into six chapters: Cryptographic Primitives 1, Java 2, Seminar Content 3, Seminar Exercises 4, Pedagogical Approach 5, and Conclusions and Future Work 6. The first two chapters lay the theoretical background, while the subsequent three focus on the learning materials utilized during the seminar and offer recommendations on how to conduct it. The final chapter discusses possibilities for future improvements and material upkeep.

Related Work

The range of teaching materials available for Java programming and cryptography is broad, spanning from formal university curricula to independent online tutorials. However, these subjects are often taught in isolation. This thesis seeks to bridge that gap by synthesizing core programming principles and cryptographic theory into a cohesive, in-person seminar tailored to modern standards.

Institutional Context At Masaryk University, several courses focus on software architecture and Java programming language proficiency, while cryptography courses address the theoretical underpinnings of security. The following information is taken from the MUNI information system catalog. [1]

- **Java Programming Courses** [1]:
  - PB162 Java Programming: A foundational course introducing students to core object-oriented programming (OOP) principles, and Java syntax.
  - PV168 Seminar in Java Programming: A project-oriented seminar designed to provide hands-on experience with practical software development.
  - PA165 Enterprise Applications in Java: An advanced course focusing on the architecture and implementation of large-scale, multi-tiered enterprise systems using modern Java frameworks and web services.
- **Cryptography and Security Courses** [1]:
  - PV080 Information Security and Cryptography: A foundational undergraduate course introducing core security concepts.
  - IA174 Fundamentals of Cryptography: A mathematically intensive course focusing on the theoretical foundations of the field.

  - PV079 Applied Cryptography: Practical course centered on deploying cryptographic mechanisms to achieve specific security goals.
  - PA018 Advanced Topics in Information Technology Security: An advanced course covering the design, development, management, and analysis of secure systems.
  - PA168 Postgraduate Seminar on IT Security and Cryptography: A seminar designed to deepen students' capabilities in analyzing and presenting advanced research topics.

The seminar developed in this thesis is designed to potentially augment **PV181 (Lab Seminar – Practical Exercises in Security and Applied Cryptography)**, providing students with a practical environment to apply Java-based cryptographic implementations.

Online and External Resources In addition to formal coursework, various online resources for self-educators and developers serve as another set of references:

- **medium.com**: A developer-oriented guide providing practical, real-world programming examples and implementation strategies for cryptographic primitives. [2]
- **docs.oracle.com**: The official technical specification for the Java Cryptography Architecture (JCA), outlining its underlying provider-based engine model, design principles, and core API structures. [3]
- **www.tutorialspoint.com**: An introductory tutorial covering the basic lifecycle, object initialization, and programmatic usage of foundational Java cryptographic primitives. [4]

1 Cryptographic Primitives

The challenge of secure communication and cryptography traces its roots back to ancient Egypt and is arguably as old as civilization itself. Historically, the primary obstacle when transmitting messages to distant correspondents was mitigating the threat posed by adversaries who might intercept and decipher the content. While the motivations of modern governments and private individuals remain consistent with these historical objectives, the methodologies employed to achieve privacy today have evolved significantly [5, 6].

This chapter examines a select group of modern cryptographic primitives featured in the seminar exercises, providing the necessary theoretical foundation for each. The discussion begins with an exploration of randomness and hash functions [5, 7] (including fuzzy hashing [8]). Following this, the focus shifts to key derivation functions, specifically *PBKDF2*. The chapter then addresses symmetric cryptography, detailing the *AES* and *ASCON* [9] algorithms, as well as the *ChaCha* stream cipher [7]. Finally, the text transitions to asymmetric cryptography, incorporating post-quantum alternatives [10, 11], to illustrate how these primitives resolve key distribution challenges over insecure channels and facilitate digital signatures.

1.1 Randomness

Unpredictability, in the form of random or pseudorandom data, is the cornerstone of modern cryptography. Without it, cryptographic systems would be entirely deterministic and highly vulnerable to cryptoanalysis. Numerous cryptographic primitives rely on this unpredictability; the one-time pad, which utilizes a purely random string of bits as a key, is the clearest example [5, 6]. However, to mitigate the key-distribution limitations of the one-time pad, practical symmetric and asymmetric cryptosystems sacrifice perfect secrecy for computational security. Both paradigms require unpredictable keys. Furthermore, symmetric systems frequently require random initialization vectors (IVs), while asymmetric systems rely on randomness for secure padding and ephemeral parameters [7]. The mechanisms

for generating this essential randomness can be divided into several categories, each presenting distinct advantages and limitations.

True Random Number Generators (TRNGs) provide the closest approximation to absolute randomness by extracting uncertainty (entropy) from physical processes. The underlying mechanisms rely on measuring microscopic physical phenomena, ranging from inherently non-deterministic quantum mechanical effects to thermal noise in semiconductors and electromagnetic fluctuations [5, 7]. System architectures may also harvest entropy from unpredictable macroscopic events, such as human input (e.g., keystroke timing or mouse movements); however, human-generated entropy is typically low-quality and highly predictable [12]. Furthermore, hardware-based TRNGs typically exhibit low throughput, rendering them too slow to generate sufficient entropy for high-volume cryptographic operations. They can also be unstable and susceptible to environmental tampering. Consequently, rather than using TRNGs directly for all random data needs, their primary function in modern cryptography is to provide a high-entropy seed for Cryptographically Secure Pseudorandom Number Generators (CSPRNGs) [5, 7].

Pseudo Random Number Generators (PRNGs) provide a highly efficient, deterministic mechanism for generating pseudorandom data. Utilizing a high-entropy seed as an initial input, a PRNG expands this starting value into a significantly longer deterministic bit stream. Functionally, such a generator performs three primary operations: it initializes its internal state using the seed, outputs a pseudorandom bit stream, and updates its internal state to ensure subsequent outputs differ [5]. Because the initial seed contains a finite amount of entropy and the generator possesses a bounded number of internal states, the output sequence is periodic. Consequently, after a specific number of iterations, known as the generator's period, the sequence of values will inevitably begin to repeat; however, in cryptographically secure generators, this period is designed to be unbruteforceable [7].

Cryptographically Secure PRNGs (CSPRNGs) represent a specific subset of the aforementioned generators, designed to produce output

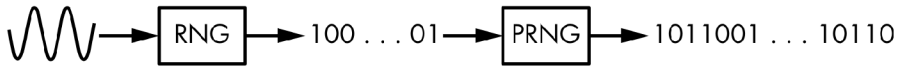


Figure 1.1: RNGs produce few unreliable bits from analog sources, whereas PRNGs expand those bits to a long stream of reliable bits. [7]

that is computationally indistinguishable from true randomness. In Unix-like operating systems such as Linux, cryptographically strong random bits are typically sourced from the special files `/dev/random` or `/dev/urandom` [7]. Furthermore, many CSPRNG implementations rely on secure hash functions, such as SHA-256, to generate the output sequence and securely update their internal state. [5, 7]

1.2 Hash Functions

This section introduces the fundamental concepts of hash functions, detailing the requisite properties of cryptographically secure algorithms alongside their *fuzzy* hashing counterparts. Furthermore, it outlines standard use cases for traditional hash functions and examines the expected characteristics and practical applications of similarity-preserving hash functions. A hash function is a function with the prescription seen in 1.1.

$$H : \{0,1\}^* \mapsto \{0,1\}^k \tag{1.1}$$

The k is some small positive integer, usually 256 or 512 bits [7]. This function maps arbitrarily long bit strings (in practice, anything that can be encoded) onto a small fingerprint.

1.3 Cryptographically Secure Hash Functions

To be deemed cryptographically secure, such functions must exhibit strong collision resistance, preimage resistance, and second preimage resistance . [5, 13] Meaning it is hard to find another bitstring that maps to the same output. Furthermore, they must demonstrate the avalanche effect. This means the function must be highly sensitive to subtle modifications; even a single-bit alteration in the input must yield

a completely uncorrelated and unpredictable output. Typical examples of modern cryptographically secure hash functions (CSHFs) include the SHA-2 and SHA-3 families, as well as those that are obsolete due to established vulnerabilities, such as the historically significant MD5 or SHA-1. [7]

1.4 Keyed Hashing (MACs)

A Message Authentication Code (MAC) ensures both data integrity and authenticity by generating a cryptographic tag that is appended to a message. Because a shared secret key is required for its creation, only authorized parties who possess it can verify or reproduce the tag. A prevalent method for constructing a MAC is via a cryptographic hash function, most notably through the Hash-based Message Authentication Code (HMAC) specification, which remains secure provided the underlying hash function resists specific cryptanalytic attacks. [7]

When integrating symmetric encryption with a MAC to achieve authenticated encryption, developers historically relied on three composition paradigms: *Encrypt-and-MAC*, *MAC-then-Encrypt*, and *Encrypt-then-MAC*. In modern practical deployments, *Encrypt-then-MAC* is widely considered the only secure approach. [7] This design allows a recipient to immediately validate the MAC tag and discard malformed data, without wasting computational resources on decrypting ciphertext that may have been tampered with. This preference aligns with the "Cryptographic Doom Principle", which states that if a system performs any cryptographic operation (such as decryption) before verifying the message's integrity, vulnerabilities will inevitably emerge. [14]

1.5 PBKDF2

Password-Based Key Derivation Function 2 (PBKDF2) is a widely adopted key derivation function designed to protect against brute-force attacks. It operates by applying a pseudorandom function, typically a Hash-based Message Authentication Code (HMAC), to a user's password combined with a cryptographic salt. This process is repeated through a specified number of iterations, a technique known as key

stretching, thereby increasing the computational cost of password guessing. [15] The security of PBKDF2 depends heavily on the choice of the underlying hash function and the iteration count. Because different hash functions possess varying computational complexities, the recommended number of iterations must be adjusted accordingly to maintain a robust defense against modern hardware. For instance, when utilizing PBKDF2 with HMAC-SHA256, the OWASP Foundation currently recommends a minimum of 600,000 iterations to ensure adequate resistance to credential cracking. [16]

1.6 Similarity Hash Functions

The primary objective of fuzzy hashing is to facilitate the identification of structurally similar, yet non-identical, data. This is achieved through algorithms that correlate input similarity with resulting hash values, ensuring that analogous inputs produce statistically comparable outputs. This property intentionally subverts the avalanche effect required of cryptographically secure hash functions, in which even minimal input variations must yield radically divergent hashes. [17] Beyond basic similarity detection, fuzzy hashing can also be used to verify data inclusion, specifically to determine whether one digital object is a substring or an embedded component of another. Practical applications of fuzzy hashing are extensive and include identifying embedded or trace evidence, tracking code versions, clustering related documents, and cross-correlating forensic artifacts across memory, disk, and network sources. [17] Within the domain of similarity detection, `ssdeep` is one of the most prominent and widely adopted tools for fuzzy hashing. [8] It is specifically designed to compute Context-Triggered Piecewise Hashes (CTPH). [18]

1.7 Stream Ciphers

Stream ciphers primarily constitute a class of symmetric-key algorithms. While public-key stream ciphers exist, such as the Blum-Goldwasser probabilistic encryption scheme, they are relatively rare and fall outside the scope of this discussion. Symmetric stream ciphers generate a continuous keystream of pseudorandom bits from a secret

key. They encrypt the plaintext by applying a bitwise XOR operation with the keystream, typically operating on one bit or byte at a time. [5] This characteristic makes them particularly advantageous in environments where transmission errors are highly probable, as bit-flips in the ciphertext do not propagate to subsequent bits during decryption. [5]

Stream ciphers typically require two inputs: a secret key and an initialization vector (IV) or nonce. The key must remain confidential, with standard lengths typically 128, 192, or 256 bits. While the IV does not need to be kept secret, it is a critical security requirement that any specific combination of a key and IV is never reused. Stream ciphers can be broadly divided into stateful and counter-based architectures. A classic example of a stateful stream cipher is RC4 (or Trivium for hardware), which continuously updates an internal state array. Conversely, counter-based stream ciphers, such as Salsa20, do not maintain a continuous internal state between operations; instead, they generate keystream blocks as a function of the key, the IV, and an incrementing counter. [7]

1.8 Block Ciphers

Block ciphers are versatile symmetric primitives that provide confidentiality for data and serve as fundamental building blocks for stream ciphers, message authentication codes (MACs), and hash functions. [5] They operate on fixed-length bit arrays (blocks), utilizing a specific mode of operation (e.g., CBC) and an initialization vector (IV) to securely process arbitrary-length data. Modern implementations standardize on block sizes of 128 bits or larger, aligning with processor registers to optimize hardware performance and provide robust collision resistance. [5, 7]

Internally, these ciphers achieve security through an iterated sequence of rounds driven by a key schedule, which derives unique subkeys from the primary secret key. Each round relies on Claude Shannon's dual properties of confusion and diffusion: non-linear substitution boxes (S-boxes) obscure the relationship between the ciphertext and the key, while linear permutations dissipate the statistical structure of the plaintext. [7] This combination ensures the cipher

satisfies the Strict Avalanche Criterion, where flipping a single input bit predictably alters approximately half of the output bits [5, 7].

The Advanced Encryption Standard (AES) Developed originally as Rijndael, AES was selected by NIST in 2001 following a multi-year public competition to establish a standard symmetric-key block cipher. Operating on a fixed block size of 128 bits, the algorithm treats data blocks as a 4×4-byte matrix, known as the state array. AES supports three cryptographic key lengths, 128, 192, and 256 bits, which dictate an execution sequence of 10, 12, or 14 processing rounds, respectively [7, 19].

Architecturally, AES is a Substitution-Permutation Network (SPN) that achieves security via an iterated series of four distinct round transformations [7, 19]:

- ***SubBytes*:** A non-linear byte substitution using a mathematically optimized S-box to provide confusion.
- ***ShiftRows*:** A linear permutation that cyclically shifts the rows of the state array by varying offsets.
- ***MixColumns*:** An algebraic mixing operation that provides diffusion across column vectors.
- ***AddRoundKey*:** An execution step that blends a round-specific subkey into the state array via a bitwise XOR operation.

1.8.1 Modes of Operation

However, the core AES algorithm alone is insufficient for practical encryption applications. As previously discussed, cryptographic modes of operation must be applied to securely process data. While numerous modes exist, each offering distinct security guarantees, functional properties, and use cases, we will focus exclusively on those relevant to our seminar exercises. [7]

Cipher Block Chaining Mode (CBC) In CBC mode, message blocks are processed sequentially, introducing a chaining dependency: each

plaintext block is XORed with the preceding ciphertext block before encryption; the initial block utilizes an Initialization Vector (IV). Because of this sequential dependency, encryption cannot be parallelized. [7] Conversely, CBC decryption can be parallelized because recovering any given plaintext block requires only its corresponding ciphertext block and the immediately preceding ciphertext block. [7]

To ensure that arbitrary-length plaintexts can be divided into whole blocks, a padding scheme, typically PKCS#7, is mandatory. PKCS#7 appends n bytes, each with a value equal to n, to fill the remaining block space. [7] If the plaintext is already a perfect multiple of the block size, an entire block of padding must still be appended to ensure the unpadding algorithm can unambiguously and deterministically strip the padding during decryption.

While CBC provides confidentiality, it lacks native integrity verification, rendering it susceptible to bit-flipping attacks. An adversary can intentionally alter bits within a ciphertext block; while this action completely scrambles the decrypted output of that specific block, it induces a predictable, bit-for-bit modification in the plaintext of the subsequent block. [7] This scheme is illustrated in Figure 1.2.

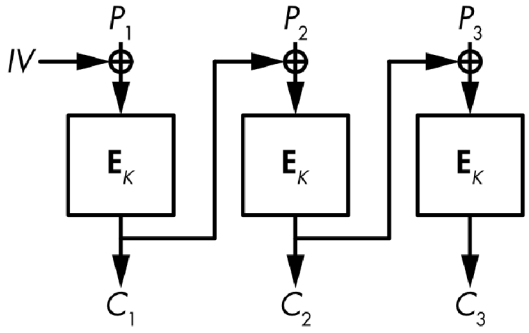


Figure 1.2: The CBC encryption mode [7].

Counter Mode (CTR) Counter mode operates similarly to a stream cipher (see Section 1.7). In this mode, a block cipher such as AES acts as a keystream generator by encrypting a unique counter value (often combined with an initialization vector, or IV). [7] The resulting output is then XORed with the plaintext to produce the ciphertext.

The counter is incremented for each subsequent block. It is crucial that the same key-counter combination is never reused, as this would compromise the encryption. [7] This scheme is illustrated in Figure 1.3.

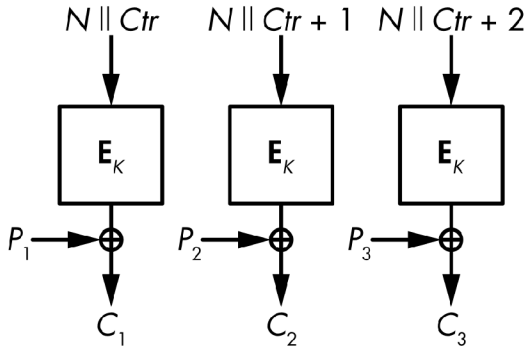


Figure 1.3: The CTR encryption mode. [7]

Authenticated Encryption with Associated Data (AEAD) The main advantage and differentiator of AEAD encryption modes is that they provide both confidentiality and integrity through a single unified interface. [20] This enables users to focus on other matters without worrying about combining encryption and MAC generation, as explained in Section 1.4. They are generally faster (optimized by performing both operations simultaneously) than classical algorithms that encrypt and then separately create a MAC. These properties led to their mandatory usage in protocols such as TLS 1.3 (Transport Layer Security) and IPsec ESP (Encapsulating Security Payload). [7, 21]

An AEAD operation takes four inputs: a key, an IV, plaintext, and associated (or additional) data. The plaintext is both encrypted and authenticated; however, the associated data is only authenticated. The output is a ciphertext, and an error indication should be returned if the encryption fails. Symmetrically, for AEAD-authenticated decryption, the algorithm takes four inputs: the key, the IV, the ciphertext, and the associated data. The operation checks the integrity of the ciphertext and associated data and outputs the plaintext. A special FAIL symbol is output if the authentication fails. [22] In the following text, we will focus on concrete implementations of AEAD algorithms.

Galois Counter Mode (GCM) builds on top of standard CTR mode. Because CTR mode lacks a built-in message authentication algorithm, it natively provides no protection against transmission errors or bit-flipping attacks. GCM has a unique additional property: it can work as a standalone Message Authentication Code (MAC) that authenticates additional data without requiring any plaintext payload. [7, 23]

The authenticated encryption operation for GCM takes four inputs, each of which is a bit string: a secret key (K), an Initialization Vector (IV) with a variable length between 1 and 2^{64} - 1 bits, a plaintext (P) that can range between 0 and 2^{39} - 256 bits, and Additional Authenticated Data (AAD) ranging between 0 and 2^{64} - 1 bits. A 96-bit IV is highly recommended because it allows for faster processing. [24, 22]

This mode has two outputs: a ciphertext (C) with a length exactly matching that of the input plaintext, and an authentication tag (T) whose length can vary between 0 and 128 bits. [24]

Authenticated decryption for GCM takes five inputs: K, IV, C, AAD, and T. It yields only one output: either the plaintext or a special FAIL symbol indicating that authentication failed. The AAD is used for data that must remain unencrypted but still requires authentication. This is essential when using GCM in secure network protocols; routing and metadata that internet nodes rely on, such as addresses, ports, protocols, and packet sequence numbers, are authenticated this way. [7] You can take a look at the GCM usage Figure 1.4 for visual input.

ASCON Ascon is a family of lightweight, permutation-based cryptographic algorithms designed as an efficient alternative to AES for resource-constrained devices. Built upon a custom iterative permutation, the suite provides Authenticated Encryption with Associated Data (Ascon-AEAD128), cryptographic hash functions (Ascon-Hash256), and extendable output functions (XOFs). [9]

1.9 Asymmetric Cryptography

Asymmetric cryptography was introduced in the 1970s with the emergence of Rivest–Shamir–Adleman (RSA) cryptosystem [7]. In contrast to classical symmetric-key encryption schemes, which utilize a singular, shared secret key for both the encryption and decryption of data,

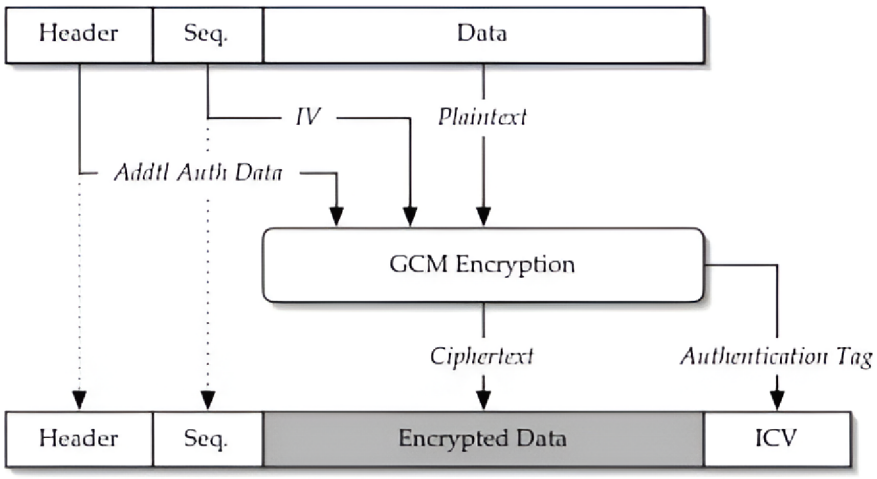


Figure 1.4: The GCM usage. [24]

asymmetric cryptography (also referred to as public-key cryptography) operates on a key pair architecture. This paradigm employs a universally accessible public key, alongside a mathematically linked (and securely retained) private key. [5, 6] In the following chapter, we will introduce a number of algorithms for both encryption and signature creation, including classical and post-quantum algorithms (those not threatened by the advent of a powerful quantum computer [7]).

1.9.1 Encryption Schemes and Key Encapsulation

In asymmetric encryption, the freely available public key is used to encrypt a message (or, in the case of KEM, encapsulate a symmetric key), which is sent to the recipient. The recipient then uses their securely stored private key to decrypt the ciphertext and recover the original data. [7]

RSA is a classical cryptosystem utilized for both encryption and signature schemes, relying on the computational hardness of integer factorization. The key generation process is as follows: we randomly generate two very large primes, p and q, and multiply them to compute the modulus $N = pq$. Because p and q are known, we can easily

compute Euler's totient function, $\varphi(N) = (p-1)(q-1)$. Next, we select a public exponent e such that $\gcd(e, \varphi(N)) = 1$. We then compute the private exponent d as the modular inverse of e, meaning $e \cdot d \equiv 1 \pmod{\varphi(N)}$, using the extended Euclidean algorithm. Often, e is chosen to have a desired property, such as a small size (e.g., 65537), for encryption efficiency. To encrypt a message m, we compute the ciphertext $c \equiv m^e \pmod{N}$. For decryption, we compute $m \equiv c^d \pmod{N}$ to recover the original message. It should be noted that this "textbook" RSA is insecure without proper padding schemes. [5, 6, 7]

ML-KEM (Formerly Kyber) The Module-Lattice-Based Key-Encapsulation Mechanism is an algorithm that uses the Shortest Vector Problem in a lattice to enable two parties to share a symmetric key. [7] It is considered to be secure against an adversary in the role of a quantum computer, it derives its security from the Module Learning with Errors (MLWE) assumption. This foundational mathematical problem is currently considered intractable, providing a robust defense against both conventional and quantum computational attacks. [10, 25]

1.9.2 Signature Schemes

In signature schemes, a private key is used to sign a cryptographic hash of the data. The freely available public key is then used by anyone to validate the signature, verifying both the data's integrity and its origin (that it was indeed signed by the owner of the corresponding private key).[7]

DSA The Digital Signature Algorithm (DSA) is a standardized public-key signature scheme whose security is fundamentally based on the discrete logarithm problem over finite fields. DSA is an optimized variant of the ElGamal signature scheme that incorporates design elements from Schnorr signatures to significantly reduce signature size and improve efficiency. [5] It uses randomization to generate a signature; the algorithm requires a newly generated, cryptographically secure random integer (a nonce, k) for every individual message. This randomness is important; if a nonce is reused or generated in a

predictable way, an attacker can trivially compute the signer's private key. [26]

ECDSA The Elliptic Curve Digital Signature Algorithm (ECDSA) is a signature scheme relying on the algebraic structure of elliptic curves over finite fields. While classical systems like RSA depend on the simple multiplication of large integers, elliptic curve cryptography (ECC) achieves its security by performing scalar multiplication to combine points on a defined mathematical curve. The security and efficiency of the system depend heavily on the specific curve chosen, as curves vary significantly in their cryptographic strength. In this scheme, all parties must agree in advance on the domain parameters: the specific elliptic curve, its order n (the total number of points on the curve), and a designated generator base point G. The signer retains a randomly selected integer d as their private key, while the public key is calculated and distributed as the curve point $P = dG$. [7]

ML-DSA (Formerly Dilithium) The Module-Lattice-Based Digital Signature Standard (ML-DSA) is a post-quantum digital signature standard. Its underlying security is derived from the computational hardness of both the Module Learning with Errors (MLWE) and Module Short Integer Solution (MSIS) problems over polynomial rings. ML-DSA operates on the "Fiat-Shamir with Aborts" framework. [11] Practical deployments of ML-DSA also require efficient public key management techniques within existing PKI architectures. [27]

Falcon Falcon (Fast-Fourier Lattice-based Compact Signatures over NTRU) is a standardized post-quantum digital signature algorithm. Its underlying security relies on the computational hardness of the Short Integer Solution (SIS) problem over NTRU lattices. Falcon signs a message by first hashing it to a random point within the lattice. The signer then utilizes their private key to compute a lattice vector that is very close to this hashed point. To authenticate the signature, the verifier uses the public key to confirm that the vector accurately corresponds to the message hash and is sufficiently "short". [28]

SLH-DSA (Formerly SPHINCS+) The Stateless Hash-Based Digital Signature Algorithm (SLH-DSA) utilizes cryptographic hash functions to produce post-quantum secure signatures. Before it was standardized, it was called SPHINCS+. Its security relies entirely on the mathematical robustness of well-established hash functions rather than complex mathematical structures like lattices. [6, 7] The scheme employs the Winternitz One-Time Signature (WOTS+) method; however, because a WOTS+ private key can securely sign only a single message, SLH-DSA uses a Merkle hypertree (a tree of nested trees) to manage the vast number of required keys. Consequently, the resulting signatures are quite large, typically tens of kilobytes. [29]

2 Java

Although the Java programming language was created in 1995 [30], it still remains a major programming language due to its unique synthesis of security, portability, and developer efficiency. Moreover, there are billions of devices running Java; we can not expect its significance to diminish any time soon. Java remains a dominant top-tier programming language, with roughly 29.4 percent of developers using it by 2025. [31]

Central to its value is the "write once, run anywhere" [32] architecture, which utilizes a standardized runtime environment to eliminate the need for platform-specific porting across diverse hardware. [30]

While Java shares a foundational syntax with C and C++, it is a distinct evolution designed to prioritize safety and simplicity over manual low-level control; nonetheless, it is used for programming smart (Java) cards [33] as well as large clouds. Its familiar structure allows C-family programmers to transition quickly, yet Java intentionally departs from its predecessors to eliminate their most error-prone characteristics. [30]

Java eliminates pointer arithmetic entirely. Instead, it uses object references. These references function similarly to pointers but are opaque handles that cannot be mathematically manipulated to access arbitrary memory locations. This ensures that a program cannot accidentally (or maliciously) corrupt the memory space of another application or the operating system. [30]

2.1 Java Sandbox

The Java sandbox architecture provides a restricted, encapsulated execution environment explicitly designed to isolate untrusted programs from the underlying host system. To enforce this isolation, the sandbox relies on robust security mechanisms, starting with bytecode verification to ensure that the code is structurally valid and memory-safe before it is ever executed. [3] Once running, the Java Virtual Machine (JVM) maintains this secure boundary through continuous runtime checks by the interpreter. Historically, this architecture also managed application privileges using comprehensive trust and policy

frameworks; by leveraging code signing to verify a program's origin and enforcing customizable permission policies, administrators could strictly control the specific actions an application was authorized to perform. [3]

Java Encapsulation Java's object-oriented principles heavily utilize encapsulation, the ability to restrict access to data. By leveraging access modifiers (such as private and protected), developers can strictly control application state, hiding sensitive internal data and methods from untrusted external classes. [30]

2.2 Java Development Kit (JDK)

The Java Development Kit is a bundle of software used for software development. Namely, to build, compile, debug, and execute Java-based applications. It contains the Java Runtime Environment (JRE) and the Java compiler (javac), which translates human-readable Java source code into platform-independent Java bytecode. [30]

2.3 Java Cryptography Architecture (JCA)

The Java Cryptography Architecture (JCA) is designed around three core principles: implementation independence, implementation interoperability, and algorithm extensibility. Implementation independence is achieved by encapsulating security services within distinct providers that plug into the Java platform via a standardized interface. This design inherently supports implementation interoperability, ensuring that applications are not tightly bound to any specific provider, and conversely, providers are not restricted to specific applications. Finally, the architecture ensures algorithm extensibility by allowing the Java platform to easily support the installation and integration of custom or third-party cryptographic providers. [3]

2.4 Java Virtual Machine (JVM)

The Java Virtual Machine (JVM) executes cross-platform Java programs by interpreting pre-compiled bytecode from .class files (which are outputted by the Java compiler). Beyond basic interpretation, modern JVMs like HotSpot self-optimize using Just-In-Time (JIT) compilation. By dynamically monitoring runtime data to identify and directly compile frequently executed "hot methods" into machine code, the JVM achieves exceptional performance that can rival or even surpass statically compiled languages like C and C++. [34]

2.4.1 Java Providers

The Java platform natively supports a wide range of security features, such as cryptography and secure communication, allowing developers to embed security directly into their applications. Within the Java Development Kit (JDK), cryptographic services are divided among multiple providers, such as “SunJCE” and “SunJSSE”. This separation exists largely due to the platform’s history and the distinct operational focus of each provider. When invoking these cryptographic services, developers can either explicitly request a specific provider or allow the Java Runtime Environment (JRE) to select one automatically (see Figure 2.2). Oracle strongly advises against hardcoding specific providers, such as “SunJCE”. As demonstrated in Figure 2.1, explicit requests tightly couple the code to a specific implementation, which can break the application if it is migrated to an environment where that specific provider is absent [35].

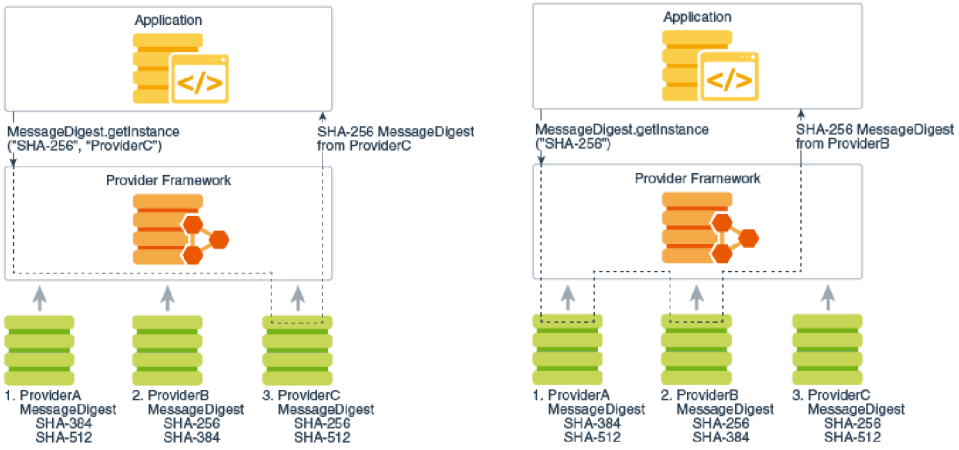
```
getInstance("SHA-256", "SunJCE");  // Not recommended

getInstance("SHA-256");            // Recommended
```

Figure 2.1: Provider request recommendations [35].

2.4.2 Sun Providers

Maintained by Oracle, the JDK’s built-in suite of cryptographic providers forms the backbone of the Java Cryptography Architecture (JCA) by



(**a**) Selection of provider with specification.[35]

(**b**) Selection of provider without specification.[35]

Figure 2.2: Selection of provider.[35]

supplying standard algorithm implementations. The collection encompasses specialized components like the "SUN" provider for basic algorithms, "SunRsaSign" for RSA signatures, and "SunPKCS11" for hardware module integration, which together establish a robust security baseline without the need for third-party libraries [35].

2.4.3 Bouncy Castle

Bouncy Castle is the primary focus of the seminar exercises. Developed by the Legion of the Bouncy Castle, the project was inherently designed "for developers, by developers." The library supplies robust open-source cryptographic APIs across languages like Java, Kotlin, and C#. It empowers developers to manage a wide range of security operations, from basic authentication and secure data transmission to complex tasks involving digital signatures and public key infrastructure (PKI). Notably, for environments requiring strict regulatory compliance, the project also maintains FIPS-certified builds for Java and C# [36].

Conscrypt Although its role in the project is primarily supplementary, Conscrypt serves as an example of a Google-backed crypto-

graphic provider widely adopted within the Android ecosystem. Its inclusion exposes students to an architecture that is not purely Java-based; instead, it operates as a wrapper around the native BoringSSL library [37, 38].

2.5 Java 25

Java 25 Long Term Support (LTS) was released on September 16, 2025. It has introduced new features designed to streamline coding and improve developer productivity, including module import declarations to reduce boilerplate code, flexible constructor bodies, scoped values for safer data sharing across threads, and a built-in Key Derivation Function (KDF) API. Additionally, promising new preview features, such as stable values and PEM encodings, alongside notable runtime and performance improvements, such as compact object headers and the Generational Shenandoah Garbage Collector, were released. This version is of our main interest. [39]

3 Seminar Content

This chapter outlines the structure and content of the in-person seminar. It focuses primarily on the underlying technology, the presentation slides and accompanying notes, student feedback (Figure 3.1), the project's code repository (available on GitLab) and its structure (Figure 3.3). The specific seminar tasks are detailed in the chapter "Seminar Exercises"(Section 4).

3.1 Seminar Preparation

During the seminar, students are expected to use either their personal laptops, which should be preconfigured with the recommended software, or the similarly equipped faculty workstations.

The recommended software environment includes the Java Development Kit (JDK) version 25, Apache Maven, IntelliJ IDEA, and, optionally, Python. Students will use IntelliJ IDEA to complete the project setup, which includes automatically resolving all Maven dependencies. Because inadequate software preparation can consume valuable instructional time, individuals unable to properly configure their personal devices are strongly advised to use the provided faculty workstations. The specific Maven dependencies are detailed in the subsection "Project Dependencies".

From the instructor's perspective, preparation extends beyond simply reviewing the course material. The instructor must preselect the specific seminar tasks students will complete and determine the assessment format, such as a test, homework assignment, or neither. They may draw inspiration from the thematic selections in the section "Seminar Execution Variants"(5.1). Based on these decisions, the presentation slides must be customized. Specifically, the slide outlining the highlighted tasks must be updated to clearly indicate the assigned exercises. Additionally, the instructor should personalize the presentation with their name, the course title, and the project repository link. Finally, any question-and-answer slides that do not pertain to the selected exercises should be hidden or removed.

3.2 Seminar Difficulty and Time Structure

The skill levels and background knowledge of students attending the seminar may vary significantly. Students without prior Java experience are encouraged to consult the online tutorials referenced in the presentation slides prior to the seminar. The instructor should assess student proficiency at the beginning of the session to assign tasks of appropriate complexity. To accommodate these varying skill levels, the exercises are designed to span a broad range of difficulty. This ensures that less experienced students can actively participate, while advanced learners still derive substantial value from the session. For instance, highly proficient students are likely to benefit from the advanced Matching Signatures(4.7) exercise.

The seminar is structured as a 100-minute session, comprising a 20-minute introduction, a 70-minute block dedicated to practical tasks, and a 10-minute concluding segment for key takeaways, solution discussions, and student feedback. While this schedule provides a general framework, it should not deter students from asking questions during the practical segment. Individual discussions and targeted assistance are encouraged throughout the exercise period. However, overarching concepts and critical takeaways should be reserved for the final 10 minutes to ensure the undivided attention of the entire class.

3.3 Slide Presentation and Seminar Flow

The presentation slides are a fundamental structural component of the seminar, designed to guide both the instructor and the students through the material. They function as visual aids to maintain context, re-engage distracted students, and provide immediate reference points for complex topics. The presentation is typeset using the LaTeX Beamer class, which allows for the inclusion of hidden instructor notes to assist with pacing and the delivery of key talking points.

The slides are chronologically structured to mirror the seminar's flow and are divided into five primary thematic segments.

3.3.1 Introduction and Context Setup

The presentation begins with an agenda and an interactive assessment of the students' prior Java programming experience, polling their familiarity with IDEs and build tools. To accommodate varying skill levels, the slides provide links to self-study resources, such as the official Java tutorials and W3Schools. This section also motivates the use of Java by highlighting its "write once, run anywhere" [32] execution model, the role of the Java Virtual Machine (JVM), and its inherent memory-safety features. Furthermore, it briefly addresses Java's sandboxing architecture and bytecode verification processes.

3.3.2 Cryptographic Foundations in Java

Prior to the practical exercises, students are introduced to the Java Cryptography Architecture (JCA) and the historical context of the Java Cryptography Extension (JCE). The slides clarify the provider-based architecture, contrasting default JVM providers, such as SunJCE, with advanced third-party alternatives like Bouncy Castle. This section also outlines the standard lifecycle of JCA engine methods , detailing the instantiation, initialization, update, and finalization stages. This theoretical foundation ensures that students understand the state-machine nature of Java cryptographic objects.

3.3.3 Developer Best Practices and Tooling

To transition into the practical exercises, the slides introduce the recommended development environment, specifically IntelliJ IDEA and Apache Maven. The presentation explains dependency management via the `pom.xml` file and details the repository's directory structure. This includes clarifying the locations of assignments and solutions, as well as the distinction between the `DynamicExercisesData` and `StaticExercisesData` directories. Furthermore, this section highlights critical security paradigms. Most notably, it addresses the memory vulnerabilities associated with storing sensitive data in immutable `String` objects, contrasting this practice with the use of mutable `char[]` arrays, which can be securely zeroed out.

3.3.4 Task Execution and Post-Task Discussion (Q&A)

The core of the presentation directly supports the seminar's coding tasks. Rather than simply listing the assignments, the slides function as a post-task review mechanism. For each seminar module, a set of questions is presented to the class. These questions, which are also included in the project files, are followed by a slide detailing the correct answers and core takeaways.

3.3.5 Conclusion and Feedback

The final segment of the slide deck facilitates the administrative wrap-up. It includes a customizable slide for the instructor to announce upcoming homework deadlines or tests. Finally, it emphasizes the importance of iterative curriculum improvement by directing students to a digital evaluation form, allowing them to provide feedback.

Student Feedback Upon the conclusion of the seminar, participants are requested to complete an evaluation form designed to facilitate iterative updates to the curriculum. The form consists of the questions listed in Figure 3.1.

1. Which exercises did you work on?
2. Which exercises did you like the most?
3. Which exercises did you like the least?
4. What did you like about the seminar?
5. What did you not like about the seminar?
6. Did you encounter any technical or conceptual issues?
7. What would you improve about the seminar?

Figure 3.1: Student Feedback Questionnaire

The questionnaire employs a mix of input methods to gather accurate data. The first question uses a multiple-select format (checkboxes) to identify exactly which tasks the students engaged with. The subsequent two questions are single-choice, while the remaining prompts require open-ended text responses to capture detailed qualitative feedback. An answer to the first question is mandatory for submission.

The feedback questionnaire can be accessed via the following link: `https://forms.gle/hKmq1q5tS4qpPvPZ9`

3.3.6 Slide Notes

The slide deck incorporates extensive supplementary notes designed to help the instructor navigate the curriculum effectively. The LaTeX Beamer class provides built-in functionality to selectively include or exclude these notes during compilation. This feature allows the instructor to customize the materials for optimal preparation and delivery.

The instructor can choose to compile the standard slides, the slides alongside the speaker notes, or a standalone document containing only the notes. This is achieved by commenting or uncommenting specific document class options in the preamble of the LaTeX source code [40].

```
%\documentclass[notes]{beamer}       % print frame + notes
\documentclass[notes=only]{beamer}   % only notes
%\documentclass{beamer}               % only frames
```

Figure 3.2: Toggling presentation notes using Beamer document class options.

3.4 Code Repository Structure

The practical tasks for the seminar are distributed to students via a structured project repository. Figure 3.3 illustrates a simplified directory tree of this repository, starting from the root level.

```
SomeMoreJava/
├── pom.xml
├── DynamicExercisesData/
├── StaticExercisesData/
│   ├── FuzzyHashingTargets/
│   │   └── *.py
│   └── Quotes/
│       └── *.txt
└── src/main/java/course/
    ├── Main.java
    ├── *Solution.java
    └── *Assignment.java
```

Figure 3.3: Simplified directory tree of the seminar repository.

The repository is organized into several high-level directories and files to separate configuration, datasets, and source code:

`pom.xml`: The configuration file for Apache Maven, containing project metadata and required cryptographic dependencies. Students directly interact with this file during the JCA Providers task.

`DynamicExercisesData/`: A directory designated for volatile data that is generated, modified, or encrypted during the seminar execution (e.g., newly generated symmetric keys or modified ciphertexts).

`StaticExercisesData/`: A directory containing read-only datasets used by the tasks, such as text quotes for encryption and Python scripts for fuzzy hashing. The only exception to this read-only constraint occurs when executing specific scripts that replicate themselves into other files within the same directory.

`src/main/java/course/`: The primary workspace where students find the application's entry point (`Main.java`), the task templates (`*Assignment.java`), and the reference solutions (`*Solution.java`).

Build and Dependency Management Apache Maven serves as the project's build automation and dependency management tool. It relies on the `pom.xml` file to handle the compilation process and automatically fetch external libraries. This simplifies the setup process and prevents students from having to manually download the required packages for the cryptographic exercises. The following list outlines these project dependencies. [41]

- **Bouncy Castle** (`org.bouncycastle:bcprov-jdk18on`, version 1.84) – Used for advanced algorithms and Post-Quantum Cryptography (PQC).
- **ssdeep4j** (`net.k-pan:ssdeep4j`, version 1.2.0) – Required for fuzzy hashing and similarity comparisons.
- **Conscrypt** (`org.conscrypt:conscrypt-openjdk-uber`, version 2.5.2) – Included only to demonstrate how to add a JCA provider dependency to the project.

4 Seminar Exercises

The core component of the seminar consists of practical Java programming and code review exercises. As outlined in the previous chapter, these applied tasks are intended to occupy the majority of the scheduled session. However, it is neither expected nor practically feasible for most students to complete the comprehensive suite of exercises within the allotted time. Consequently, a curated subset of tasks should be selected to optimize the learning experience and accommodate time constraints. For guidance on selecting an appropriate task set, refer to Section 5.1. To facilitate this selection, the exercises are systematically classified into the following modules: JCA Providers, Random Number Generators, Symmetric Cryptography, (Keyed) Hashing, Key Derivation Functions, Asymmetric Cryptography, and Signature Matching.

4.1 JCA Providers Task

This exercise is designed to introduce students to the Java Cryptography Architecture (JCA) Provider model by integrating three distinct cryptographic providers: the native SunJCE provider [35], the standard Bouncy Castle library [36] (which serves as the primary provider for the majority of the coursework), and Conscrypt [37], which is introduced specifically for this module. Figure 4.1 outlines the instructions presented to the students.

Initially, the curriculum design required students to manually add the Bouncy Castle dependency. However, from a structural perspective, doing so during this specific exercise would inadvertently make the task a prerequisite for subsequent modules. To maintain module independence, Bouncy Castle is instead pre-configured in the project's baseline. Consequently, the configuration component of this task remains strictly isolated: students must manually integrate only the Conscrypt dependency into the project's Maven `pom.xml` file, the location of which is illustrated in Figure 3.3.

Subsequently, students synchronize the build configuration within their IDE (using the "Reload All Maven Projects" function) to ensure the newly added dependency is properly resolved. Upon successful compilation and execution of the application, they perform a com-

Instructions:

1. Add following dependency into `pom.xml` file among other dependencies:

```
<dependency>
    <groupId>org.conscrypt</groupId>
    <artifactId>conscrypt-openjdk-uber</artifactId>
    <version>2.5.2</version>
</dependency>
```

2. Use IDEA Maven to Sync all Maven projects - this will download the dependency.
3. Run this code and observe the console output. (Optionally fix the warning.)
4. Match the printed output to the code to understand how Java selects a Cryptography Provider.

Get ready to answer the following questions:

1. Which provider did Java use by default before we made any changes?
2. What is the functional difference between `addProvider()` and `insertProviderAt()`?
3. Why did the HC256 cipher use Bouncy Castle, even though Conscrypt was set to priority 1?

Figure 4.1: Conscrypt Provider Setup and Review Questions. [42]

parative analysis between the source code and the resulting console output to deduce the answers to the assignment's questions. Figure 4.2 displays this expected console output.

```
--- 1. Default Provider Resolution ---
Default AES Provider: SunJCE
Details: SunJCE Provider (implements ...)

--- 2. Adding Conscrypt and Bouncy Castle Providers ---
Explicit Conscrypt Provider: Conscrypt
Explicit BC Provider: BC
Explicit SunJCE Provider: SunJCE

--- 3. Modifying Provider Priority ---
New Default AES Provider (Position 1): Conscrypt
Fallback Cipher (HC256) Provider resolved to: BC
```

Figure 4.2: JCA Providers Task Solution Output

Students should verify their answers against the reference solutions provided in the slide deck, as illustrated in Figure 4.3.

Takeaway There are two pedagogical objectives in this exercise: first, to ensure students can successfully integrate a new cryptographic provider into a Maven project; second, to develop their ability to trace provider resolution. By the end of the task, students should be able to accurately distinguish between providers resolved implicitly via the JVM's default priority list, those requested explicitly by name, and those invoked via the fallback mechanism.

4.2 Random Bytes Generators

The primary objective of this exercise is to clarify the operational and security distinctions between standard pseudo-random number generators (PRNGs) and cryptographically secure pseudo-random number generators (CSPRNGs). Students are assigned four specific tasks: generating a predefined volume of bytes from a CSPRNG and

1. **Built-in Defaults:** Java uses its built-in providers (`SunJCE`) by default.
2. **Positioning in the Chain:**
   - `addProvider()` appends the provider to the *end* of the priority list, making it a fallback.
   - `insertProviderAt(..., 1)` shifts the list and places the provider at the *beginning* (priority hijacking), making it the primary default for any algorithm it supports.
3. **The Fallback Mechanism:** Providers act as a chain of responsibility. Conscrypt didn't support HC256, so the JVM continued down the priority list until it reached Bouncy Castle!

Figure 4.3: JCA Providers: Answers

saving them to the project directory (Figure 3.3); benchmarking the execution times of both a CSPRNG and a standard PRNG; deducing the classification of an unknown generator based on these recorded metrics; and, finally, exploiting the deterministic nature of a Linear Congruential Generator (LCG) to predict subsequent "random" outputs. Additionally, students are instructed to prepare answers for the questions presented in Figure 4.4.

1. Which generator is faster: `Random` or `SecureRandom`?
2. What is the approximate difference in magnitude?
3. How could an attacker exploit this timing difference?
4. What does it mean for an application if an attacker discovers it uses an insecure RNG?
5. What happens if the attacker successfully reconstructs the internal "seed"?

Figure 4.4: Random Number Generators: Questions

Benchmarking During this phase, students benchmark generation latencies and leverage the resulting timing discrepancies to execute a rudimentary side-channel analysis. To ensure benchmarking accuracy, the implementation explicitly accounts for the Java Virtual Machine (JVM) execution environment. Because the JVM initially interprets bytecode, introducing significant execution overhead, the code utilizes a "warm-up" loop. This forces the Just-In-Time (JIT) compiler to optimize the methods into native machine code before formal benchmarking commences, thereby yielding reliable timing data.

Timing Side-Channel Attack A subsequent objective of the benchmarking phase is to identify an underlying generator based on its average generation latency. By establishing baseline execution times for the `SecureRandom` and `Random` classes over thousands of iterations,

the application can analyze the performance profile of an unknown generator. If the observed execution time falls within a defined statistical boundary of the baseline metrics, students can reliably deduce the algorithm of the black-box generator without inspecting its source code.

Breaking the LCG This phase demonstrates why `java.util.Random` is strictly prohibited in security contexts. Once an insecure PRNG is passively identified via a timing attack, its deterministic nature can be exploited to recover its state. The next state of the underlying linear congruential generator (LCG) depends on the previous state and three constants (Figure 4.5): A, C, and M. The exact relation is shown in Equation 4.1, where the next state, X_{n+1}, is computed from the previous state, X_n.

Because the internal state is only 48 bits long and each invocation exposes only the upper 32 bits, an attacker needs to capture only two consecutive outputs. By extracting the known 32 bits from the first output, the attacker can perform a brute-force search over the remaining 2^{16} possible lower bits, verifying each candidate state by computing the next value and comparing it with the second captured output.

```
final long A = 0x5DEECE66DL;
   // Multiplier
final long C = 0xBL;
   // Constant addition
final long M = (1L << 48) - 1;
   // Modulo mask 2^48
```

Figure 4.5: Constants in `java.util.Random`

$$X_{n+1} = (A \cdot X_n + C) \pmod{M} \tag{4.1}$$

Takeaways The core pedagogical takeaway is recognizing the inherent trade-off between security and performance. CSPRNGs exe-

cute computationally intensive operations out of cryptographic necessity, rendering them significantly slower than their insecure counterparts, which are optimized for performance and require minimal CPU cycles. Consequently, `java.util.Random` prioritizes speed at the expense of security, whereas `java.security.SecureRandom` ensures cryptographic integrity at the cost of computational efficiency. Upon completing this exercise, students should come up with the similar answers as in Figure 4.6.

1. `java.util.Random`.
2. It is significantly faster (often 5x to 10x faster).
3. An attacker can measure server response times. If generating a token is suspiciously fast, they know to try an LCG brute-force attack.
4. `java.util.Random` only has a 48-bit seed. By observing just two outputs, an attacker can brute-force the missing 16 bits in milliseconds.
5. Once the seed is known, the attacker can perfectly predict all past and future values (e.g., password reset tokens, session IDs).

Figure 4.6: Random Number Generators: Answers

4.3 Symmetric Cryptography

This segment of the seminar focuses on symmetric cryptography. While the primary emphasis is on the Advanced Encryption Standard (AES) block cipher, a supplementary exercise addressing stream ciphers is incorporated to provide a comprehensive overview. The practical component comprises six distinct tasks. The first four assignments are dedicated to implementing and applying various encryption modes to secure data. The fifth task explores cryptographic vulnera-

bilities, requiring students to execute a bit-flipping malleability attack against a message encrypted in Cipher Block Chaining (CBC) mode. Finally, the sixth task involves a critical review of a contemporary ASCON implementation, specifically analyzing its direct provider integration that circumvents the standard Java Cryptography Architecture (JCA) `Cipher` interface. Students are also encouraged to prepare answers to the questions presented in Figure 4.7.

1. Did the output differ when using a different JCA provider?
2. Why is the GCM ciphertext length different from CBC, even though the plaintext was exactly the same?
3. Did you manage to successfully alter the value in the CBC ciphertext?
4. Besides the targeted altered value, how else did the CBC plaintext change after your attack?
5. What error did you encounter when modifying the GCM ciphertext before decryption?

Figure 4.7: Symmetric Cryptography: Questions

To establish continuity within the broader curriculum, students begin these tasks by reusing a pseudorandom byte sequence generated in the previous RNG exercise (Section 4.2). From this sequence, they must systematically extract the cryptographic keys and initialization vectors (IVs) required for the subsequent implementations.

4.3.1 Block Ciphers and Modes of Operation (Tasks 1&2)

The foundational exercises require students to implement AES encryption using different operational modes, thereby observing their distinct characteristics and parameters. Students first implement AES in Cipher Block Chaining (CBC) mode. A critical learning objective is understanding parameter constraints; specifically, students must

extract and apply a strict 16-byte IV, which aligns with the AES block size. To contrast with CBC, students subsequently implement AES using Galois/Counter Mode (GCM). This exercise highlights two fundamental differences introduced by AEAD ciphers. First, students are instructed to use a 12-byte (96-bit) nonce, which is the mathematically optimal parameter for GCM, rather than the 16 bytes expected by CBC. Second, upon encrypting identical plaintext with both modes, students are asked to compare the resulting ciphertext lengths. This discrepancy serves as a practical demonstration of the authentication tag (typically 128 bits) that GCM appends to the ciphertext to ensure data integrity. Finally, students are encouraged to encrypt the data using a different cryptographic provider (with the same parameters) and compare the outputs.

4.3.2 Contemporary Lightweight Cryptography: ASCON (Task 4)

The curriculum incorporates a code review task centered on ASCON [9], a recently standardized lightweight cryptographic algorithm. During the design of the seminar, it was observed that ASCON had not yet been fully integrated into the standard Java Cryptography Architecture (JCA) via the generic `Cipher.getInstance()` abstraction. Consequently, the implementation provided by Bouncy Castle bypasses the JCA entirely, interacting directly with the provider's `AsconEngine` API. This entails invoking the methods described in Section 3.3.2 directly on the `AsconEngine` instance rather than through the generic `Cipher` class.

This architectural variance is intentionally retained as a valuable pedagogical tool. It exposes students to standard industry practices regarding the software lifecycle of emerging cryptographic algorithms, demonstrating how developers leverage direct provider APIs before generic abstractions are updated to accommodate them. Naturally, this specific exercise may evolve or become obsolete as standard JCA interfaces are updated to natively support ASCON.

4.3.3 Ciphertext Malleability and Integrity Attacks (Task 3)

To demonstrate the critical vulnerability of unauthenticated encryption modes, the seminar introduces a targeted bit-flipping attack. Stu-

dents are presented with a CBC-encrypted message containing a specific monetary value (e.g., "00005"). By exploiting the mathematical properties of the CBC decryption process, students must apply an XOR operation to the raw ciphertext. This allows them to predictably alter the underlying plaintext without possessing the cryptographic key.

Following the successful manipulation of the CBC ciphertext, students are instructed to attempt the exact same bit-flipping technique against the GCM-encrypted data. Rather than returning modified or corrupted plaintext, the GCM implementation throws a cryptographic exception (e.g., `AEADBadTagException`). This effectively demonstrates the necessity of AEAD modes in modern cryptography, illustrating how GCM's authentication tag inherently prevents malleability attacks.

4.3.4 Stream Ciphers and Data Streaming (Tasks 5&6)

ChaCha20 Implementation While block ciphers form the core of the module, addressing stream ciphers is essential for a comprehensive cryptographic education. Therefore, a dedicated task requires students to implement encryption and decryption routines using the ChaCha20 stream cipher (specifically, the `ChaCha7539` algorithm variant provided by Bouncy Castle).

Large File Processing Finally, a bonus exercise challenges students to encrypt a large file, such as a video. This task introduces the practical limitations of in-memory data processing. Students are instructed to transition from standard byte-array manipulation to Java's `CipherOutputStream`, demonstrating how to efficiently process and encrypt continuous data streams without exhausting system memory.

Takeaways This segment equips students with a practical understanding of modern symmetric cryptography. By implementing AES in both CBC and GCM modes, students directly observe the necessity of Authenticated Encryption with Associated Data (AEAD) to prevent malleability attacks and ensure data integrity. Furthermore, exploring the ChaCha20 stream cipher, addressing large file processing constraints, and analyzing an early-stage ASCON integration exposes

students to the complexities of real-world API architectures. Finally, students should be able to formulate answers to the questions in Figure 4.7 similar to those presented in Figure 4.8.

1 No difference. Standardized algorithms produce identical results across any provider, as long as the Key and IV are identical.

2
- **CBC** adds PKCS#7 padding to fill out the 16-byte block size.
- **GCM** acts as a stream cipher but appends an Authentication Tag (usually 16 bytes) to the end.

3,4 Modifying ciphertext block N-1 allows you to predictably flip bits in plaintext block N, but it completely *corrupts/scrambles* plaintext block N-1.

5 It throws an `AEADBadTagException`. GCM detects the tampering via the Authentication Tag and safely refuses to decrypt.

Figure 4.8: Symmetric Cryptography: Answers

4.4 Hashing and MACs

This module of the seminar is structured into two distinct segments: cryptographic hashing (including keyed hashes) and similarity hashing (often referred to as fuzzy hashing). By placing these methodologies side-by-side, the exercise highlights the diverse applications of hash functions within the broader cybersecurity landscape.

By applying both techniques to the exact same dataset, students observe that modifying even a single byte in a file yields a completely divergent cryptographic hash (a necessity for pre-image and collision resistance). Conversely, they witness how similarity hashing intentionally strips away the avalanche effect, sacrificing data integrity

verification to enable the clustering and identification of minimally altered files. Finally, students are expected to derive answers to the questions presented in Figure 4.9.

1. Which operation was faster: computing the Hash or the MAC?
2. Conceptually, why does this performance difference make sense?
3. Does fuzzy hashing possess standard cryptographic properties?

Figure 4.9: Hashing & MACs: Questions

4.4.1 Cryptographic Hashing and Message Authentication

The first segment examines fundamental cryptographic primitives, specifically comparing standard hash functions (e.g., SHA3-512) against Hash-based Message Authentication Codes (HMACs). Students are tasked with implementing both algorithms to generate digests from file data. Furthermore, they conduct microbenchmarking to measure and compare the computational overhead of each operation over numerous iterations. This practical measurement prompts students to critically evaluate the performance implications of using a keyed authentication mechanism versus an unkeyed integrity check.

4.4.2 Similarity Hashing in Malware Analysis

While similarity hashing traditionally aligns with malware analysis and digital forensics rather than pure cryptography, it is introduced here as a relevant, complementary security concept. Using the `ssdeep` library for Java, students analyze a dataset comprising simulated, benign, self-replicating Python scripts (i.e., code capable of reading and propagating its own source). The provided files include the original payload (`virus.py`), a slightly modified version of the payload

(`modified_virus.py`), an infected target containing both the virus and benign utility code (`infected_target.py`), and a clean target entirely devoid of malicious code (`target.py`). [8]

To determine which files are infected, students compute the fuzzy hashes for each file and use a comparator algorithm to generate similarity scores. As an interactive component, students may execute the Python script within an isolated directory to observe the file infection mechanism prior to conducting their hash comparisons.

Self-Replicating Python Code This self-replicating Python script (`virus.py`), selected for its structural simplicity, serves as an effective practical model for educational environments focusing on fuzzy hashing. The script's architecture is divided into three distinct components. [43]

The first component dynamically accesses its host file, using a defined pair of boundary tags to locate and store the viral code within an array. [43]

The second component systematically scans only the local directory for other Python scripts. It propagates the infection by prepending the stored viral code to the beginning of each target file, ensuring the viral code executes before the host's original instructions. To prevent multiple infections within a single file, it explicitly excludes any scripts that already contain the boundary tags. [43]

The third and final component executes the payload. For demonstrative purposes in this context, the payload consists solely of a benign terminal output, printing "YOU HAVE BEEN INFECTED HAHAHA !!!" to confirm successful execution. [43]

Takeaways Upon completing this module, students should understand the fundamental differences between various hash functions and be capable of formulating answers to the questions in Figure 4.9 similar to those presented in Figure 4.10.

4.5 Key Derivation Functions (KDFs)

A common practical challenge in applied cryptography involves bridging the gap between low-entropy, human-memorable passwords and

1. Hashing is generally faster.
2. A MAC (specifically HMAC) requires hashing the data *plus* securely mixing in a secret key (which usually involves executing the underlying hash function twice).
3. **No.** Cryptographic hashes require pre-image resistance, collision resistance, and a strict **avalanche effect** (changing 1 bit changes ∼50% of the output).

Figure 4.10: Hashing & MACs: Answers

the high-entropy, pseudorandom bit strings required by symmetric algorithms. To address this, a dedicated module on Key Derivation Functions (KDFs) is introduced.

This task presents students with a scenario: a user wishes to encrypt sensitive personal data, such as a digital diary, using a symmetric cipher. However, to mitigate the security risks associated with storing a raw cryptographic key, the user prefers to memorize a passphrase from which the key can be deterministically reconstructed on demand.

1. Can you give an example of a real-world scenario where deriving a key from a password is used?
2. Can you recover the AES key without the password or the salt?
3. Why does the code specifically run 600,000 iterations?
4. Why did Task 3 ask you to overwrite the password character array with `'\0'` (null characters)?

Figure 4.11: Password-Based Key Derivation: Questions

Students are tasked with implementing Password-Based Key Derivation Function 2 (PBKDF2) to securely stretch a human-readable passphrase

into a 256-bit AES key. This implementation introduces the critical concept of the cryptographic "work factor" or iteration count, a configurable computational delay designed specifically to thwart brute-force and dictionary attacks against relatively weak human inputs. Furthermore, the exercise emphasizes secure memory management by requiring students to handle the password as a mutable character array (`char[]`) rather than an immutable string. This ensures the sensitive plaintext can be explicitly zeroed out of system memory immediately after key derivation. Through this practical implementation, students also recognize the vital role of cryptographic parameters. Students must explicitly export and preserve these parameters alongside the ciphertext; otherwise, the original encryption key cannot be re-derived, rendering the data permanently unrecoverable. Finally, as an integration exercise, students pass this newly derived key and the saved parameters into their previous symmetric encryption routines to securely encrypt and decrypt a target file. Throughout the implementation, students are expected to formulate answers to the questions presented in Figure 4.11.

Takeaways Students learn that while the salt and nonce must be randomly generated during the encryption phase to prevent pre-computation (rainbow table) attacks, they are not secret. Furthermore, they should understand that sensitive data must be stored in mutable data structures that can be explicitly zeroed out of memory. Finally, students should be capable of formulating answers to the questions in Figure 4.11 similar to those presented in Figure 4.12.

4.6 Asymmetric Cryptography Tasks

This module is divided into two practical exercises focused on modern asymmetric cryptography, bridging the gap between classical and post-quantum algorithms. The questions associated with this exercise (Figure 4.13) are primarily centered on the benchmarking component.

Signature Scheme Benchmarking In the first task, students execute a benchmarking suite that signs and verifies a payload using classical Elliptic Curve Cryptography (EC P-384) alongside the post-quantum

1. Password managers (KeePass, Bitwarden), Full Disk Encryption (LUKS, BitLocker), or cryptocurrency wallets that encrypt private keys.
2. **No.** You need the exact password *and* the exact original salt to reproduce the same AES key.
3. It intentionally makes the derivation process slow. This makes brute-force and dictionary attacks computationally expensive for attackers.
4. Variables can linger in RAM even after they are no longer used. Overwriting passwords prevents extraction via memory dumps.

Figure 4.12: Password-Based Key Derivation: Answers

1. Which algorithm was faster overall: EC-based or Post-Quantum?
2. Did you observe an asymmetry in performance? (How did signature creation compare to verification for each algorithm?)

Figure 4.13: Asymmetric Cryptography: Questions

ML-DSA scheme. Students must run the code, account for execution variables such as JVM warmup, and record the average execution times in nanoseconds. They will use these results to analyze the practical performance trade-offs between the two schemes regarding signature creation and verification during the concluding seminar discussion.

Key Encapsulation Mechanism Scenario The second task challenges students to implement a secure key exchange workflow between two communicating parties (Alice and Bob). While establishing a shared secret for AES-GCM encryption is a standard pedagogical exercise, this task modernizes the approach by requiring students to use the recently standardized post-quantum algorithm, ML-KEM. Students must implement the requisite cryptographic logic using the Bouncy Castle provider for both the sender (performing the encapsulation) and the receiver (performing the decapsulation) to successfully recover and decrypt a hidden message.

Takeaways Upon completing these exercises, students acquire hands-on experience with post-quantum cryptography (PQC), specifically regarding the practical implementation of the ML-DSA and ML-KEM standards. Utilizing these practical observations, students should be capable of formulating well-reasoned answers to the benchmarking questions, achieving analytical outcomes similar to those detailed in Figure 4.14.

1. It depends on the operation: Classical EC (P-384) is generally faster at generating signatures, but Post-Quantum (ML-DSA) is significantly faster at verifying them.

2. Yes, strong asymmetry: ML-DSA signing is significantly slower than its verification. Conversely, EC P-384 verification actually takes slightly longer than its signature creation.

Figure 4.14: Asymmetric Cryptography: Answers

4.7 Matching Signatures Task

This section outlines an experimental approach to teaching various signature schemes within the Java Cryptography Architecture (JCA). The objective is to design a challenging, open-ended task that goes beyond a standard, linear programming exercise. This exercise challenges students to dynamically map authors' quotes to randomized keys and signatures. It integrates both classical algorithms (RSA, ECDSA, DSA) and post-quantum schemes (ML-DSA, Falcon, SLH-DSA). Participants are provided with anonymized and shuffled sets of quotes, public keys, and signatures. The mapping between the set of quotes and the set of signature types is surjective, ensuring that each of the six signature schemes is applied at least once.

While it is theoretically possible to solve this task without writing any code by manually inspecting file sizes, analyzing ASN.1/DER structures, or using command-line tools, doing so is highly non-trivial. Consequently, developing a programmatic solution using the JCA API often proves to be the more efficient route.

The primary pedagogical goal of this exercise is to highlight the distinct properties inherent to various cryptographic algorithms. Most notably, students must observe and navigate the stark contrast in key and signature lengths, particularly when comparing classical schemes (such as RSA or ECDSA) against newer post-quantum algorithms (such as ML-DSA or Falcon). Given the randomized nature of the provided files, the most effective strategy is a structured, programmatic brute-force approach. This methodology forces students to learn how to iteratively attempt signature verification, relying heavily on JCA exception handling (e.g., catching `InvalidKeySpecException` or `SignatureException`) to systematically deduce the correct algorithmic pairings.

Takeaways Through this open-ended exercise, students practically observe the significant disparities in size between classical (RSA, ECDSA) and post-quantum (ML-DSA, Falcon) cryptographic signatures. By automating a brute-force solution, they also deepen their proficiency with the Java Cryptography Architecture (JCA), learning to utilize exception handling as a diagnostic tool to systematically

identify unknown cryptographic data. Unlike the preceding modules, the questions and answers in Figure 4.15 and Figure 4.16 are intended primarily as discussion material.

- Did you use the programmatic approach (JCA exception handling) or the analytical approach (looking at file sizes and ASN.1 structures)?
- How many combinations did you manage to match? How long did it take you to write the code?
- What did you notice about the sizes of the public keys and signatures across the different algorithms?
- Why does a bitwise comparison (with some algorithms) fail if you try to sign the quote yourself and compare it against the provided `.sig` files?

Figure 4.15: Signature Matching: Discussion

4.8 Test / Homework Assignments

The assignment consists of three components: Hash Precomputing, the AES CTR Bit-Flip Attack, and a Code Review. Given a maximum evaluation of 10 points, I recommend a distribution of 4, 3, and 3 points, respectively. Supplemental points may be awarded for particularly creative solutions or unique technical insights.

4.8.1 Hash Precomputing (Partial Preimage Attack)

This assignment places the student in the role of an attacker attempting to exploit human error in manual checksum verification. Attackers often alter malware to produce a hash that matches the original software's hash at the very beginning and the end, anticipating that careless victims will only visually inspect those specific bytes. [44, 45]

- **Signature Size:** You can often identify algorithms just by **file size**. Classical ECC produces tiny keys and signatures, whereas Post-Quantum algorithms (especially SLH-DSA) are massive.
- **Dynamic JCA Discovery:** Looping through `KeyFactory` algorithms and catching `InvalidKeySpecException` is a practical trick to dynamically discover the algorithm of an unknown DER file in Java.
- **Randomized Signatures:** Many modern signature schemes (like ECDSA, ML-DSA, and Falcon) incorporate randomness. Signing the exact same file with the same private key twice produces different bytes. You *must* use the cryptographic `verify()` method!

Figure 4.16: Signature Matching: Takeaways

The simulation requires students to write a brute-force search loop to find a SHA-256 hash collision. To ensure individual effort and prevent hardcoded solutions, the target hash is uniquely derived from the student's University Identification Number (UČO). The student appends an incremental modifier to their UČO until the resulting hash matches the required target bytes.

If deployed as a take-home assignment, requiring a 4-byte collision (e.g., the first two and last two bytes) is appropriate, as finding a 32-bit partial preimage can take approximately 30 minutes on standard hardware. Conversely, for an in-class or inter-semester exam with strict time limits, the requirement should be reduced to a 3-byte collision (24 bits), which typically computes in under a second.

4.8.2 AES CTR Bit-Flipping

This task is structurally similar to exercises previously covered in the seminar, designed to reward students who have thoroughly prepared. The objective is to perform a targeted modification of a ciphertext (e.g., changing a financial transfer amount from "00005" to "99999") without possessing the encryption key.

Because the AES-CTR mode acts as a stream cipher, students can manipulate the decrypted plaintext by applying an exact XOR mask to the ciphertext. Students must construct an attack vector precisely matching the length of the ciphertext and calculate the required flipped bytes using the property $C' = C \oplus P \oplus P'$, where C is the original ciphertext, P is the known original plaintext, and P' is the target plaintext. This demonstrates the critical lack of integrity checks in unauthenticated stream ciphers.

4.8.3 Cryptographic Misconfigurations (Code Review)

In this code review task, students must analyze an insecure implementation of the Java Cryptography Architecture (JCA) using AES-GCM and identify fundamental security flaws. The snippet contains three major vulnerabilities: the use of an insecure random number generator `java.util.Random` for key creation, an uninitialized (zeroed) Initialization Vector (IV), and a dangerously short Authentication Tag length of 32 bits (well below the NIST recommended 96 bits). This

exercise forces students to transition from an attacker's mindset to a defender's perspective, emphasizing the importance of secure API usage.

4.8.4 Deployment Recommendations

As noted, the difficulty of these assignments can be easily scaled depending on whether the task is a timed exam or a take-home assignment. For example, an instructor might increase the required brute-force match from three bytes to four, include more complex "filler" code in the Code Review task, or require students to actively remediate the discovered vulnerabilities rather than simply identifying them.

If additional assessment materials are required, several of the seminar exercises could serve as examination candidates, provided the specific datasets remain distinct from the standard seminar materials. As an alternative to the assignments described here, the "Matching Signatures Task" (Section 4.7) is recommended for its open-ended nature.

5 Pedagogical Approach

5.1 Seminar Execution Variants

Because the number of developed exercises exceeds what is manageable within the allotted time, we have designed four distinct curriculum variants to accommodate different pedagogical goals and student proficiency levels. While the provided presentation slides are tailored by default to the *Standard Option*, instructors can easily adapt them should an alternative variant be selected.

Standard Option This variant offers a balanced approach covering core cryptographic concepts. It prioritizes tasks directly relevant to the final assessment alongside an introduction to post-quantum cryptography (PQC). When utilizing this option, the instructor should advise students to review the section on (Keyed) Hash Functions independently as part of their homework or test preparation.

[1.] JCA Providers

[2.] Random Number Generators

[3.] Symmetric Cryptography

[6.] Asymmetric Cryptography

Test Preparation Option This variant focuses strictly on the core concepts required for the evaluation, maximizing direct test preparation. To accommodate the condensed focus, the "Matching Signatures Task" and the post-quantum benchmarking exercises are omitted, allowing for a more thorough, in-class review of symmetric encryption and Hash Functions.

[2.] Random Number Generators

[3.] Symmetric Cryptography

[4.] (Keyed) Hash Functions

Advanced Coding Variant Designed for cohorts with prior Java proficiency, this variant emphasizes advanced implementation techniques and post-quantum cryptography. In this curriculum path, foundational discussions are condensed to prioritize the "Matching Signatures Task" and more complex PQC implementation scenarios, challenging students to engage deeply with the technical nuances of the Java Cryptography Architecture (JCA).

[3.] Symmetric Cryptography

[5.] Password-Based Key Derivation Function (PBKDF)

[6.] Asymmetric Cryptography

[7.] Matching the Signatures

Novel Concepts Variant This variant de-emphasizes rote coding and direct test preparation in favor of exploring emerging or specialized cryptographic paradigms. This exploratory path introduces students to the extensibility of JCA providers, the mechanics of fuzzy hashing, and advanced key derivation techniques, culminating in the open-ended signature matching challenge. It is designed to foster a higher-level analytical understanding of the cryptographic landscape rather than focusing exclusively on implementation syntax.

[1.] JCA Providers

[4.] (Keyed) Hash Functions

[5.] Password-Based Key Derivation Function (PBKDF)

[7.] Matching the Signatures

5.2 Conducting Programming Seminars in the Era of LLMs

Modern Large Language Models (LLMs) are highly capable of generating accurate solutions for many of the programming tasks presented in this seminar. Consequently, a clear pedagogical strategy must be

established regarding their use. While the utilization of LLMs should be strictly prohibited during graded homework or examinations to ensure an authentic assessment of student knowledge, their role during the seminar sessions remains a subject of active debate. To maximize the educational benefit in this evolving landscape, instructors may consider the following approaches:

1. **Strict Documentation Reliance:** Instructors may prohibit the use of LLMs to encourage students to navigate official Bouncy Castle and Oracle documentation, thereby fostering traditional software engineering research skills. This is arguably the most effective approach for deep learning; the JCA Reference Guide, which is emphasized throughout the presentation slides, contains the vast majority of the information required to complete the tasks. Should specific details be missing from the guide, students are encouraged to utilize standard search engines to find reputable technical resources.

2. **Guided LLM Assistance:** Tasks and constraints can be modified such that students utilize LLMs primarily as a personalized tutor to explain complex concepts or debug specific logic errors, rather than as a tool for generating entire solutions.

3. **Full LLM Integration:** Alternatively, instructors could remove the provided solution templates entirely and challenge students to leverage LLMs to their full potential. This approach shifts the focus toward teaching effective prompt engineering and the nuances of AI-assisted development within a security-sensitive context.

5.3 Real-World Deployment and Iterative Adjustments

It is anticipated that the seminar structure and materials will undergo refinement following their initial deployment in a classroom environment. Student feedback and observational data will be instrumental in identifying potential areas for optimization. While the current curriculum has been carefully calibrated to meet its stated learning objectives, the final degree of success will be determined by its practical reception

and the subsequent minor adjustments required to address student needs.

5.4 Thematic Data Selection

To increase student engagement and make the practical exercises more entertaining, the cryptographic tasks utilize a thematic dataset rather than generic dummy text. The plaintext files used for encryption, hashing, and signature matching contain selected quotations from prominent science fiction and fantasy authors, including Frank Herbert, Steven Erikson, Terry Pratchett, J.R.R. Tolkien, George R.R. Martin, Douglas Adams, Brandon Sanderson, and Andrzej Sapkowski. This approach not only provides a more cohesive narrative across the tasks but also aids in the "Matching Signatures Task", where students must identify the literary origin of various signed payloads.

6 Conclusions and Future Work

This work successfully developed a comprehensive suite of seminar exercises and accompanying evaluation metrics designed to stimulate student interest and engagement in cryptography. The implemented tasks encompass random number generation, keyed and unkeyed hash functions, symmetric and asymmetric cryptography (including post-quantum cryptography), and password-based key derivation functions. These resources are designed to encourage active student engagement with the course materials, fostering a deeper understanding and long-term retention of the concepts compared to passive learning methods.

Future Work Future work can be categorized into three primary directions. The first involves iteratively refining the current materials based on continuous student feedback. Possibly improving the task selection, omitting some topics, or introducing new ones. Validating and adjusting the time schedule. Potentially calibrating the homework/test assignment difficulty.

The second direction focuses on maintaining the curriculum's relevance by incorporating new developments and research in the field. The necessity of this ongoing maintenance was already demonstrated during the creation of these materials: following the release of a new version of the Bouncy Castle library, the slides, seminar exercises, and the thesis text were systematically updated to ensure the content remained up to date upon completion of this work.

A third future direction explores integrating Large Language Models (LLMs). An excellent suggestion emerged to enhance the RNG module by having students generate random values using LLMs and analyze them for underlying biases. Incorporating this into the Java-based framework, potentially via direct API calls once the standard libraries become widely available, would be a highly relevant and modern addition to the seminar.

Bibliography

1. MASARYK UNIVERSITY. *Information System of Masaryk University: Course Catalogue* [online]. Faculty of Informatics. [visited on 2026-05-18]. Available from: `https://is.muni.cz/auth/predmety/`. Authenticated access required.
2. CIBOFDEVS. *Mastering Cryptography in Java: A Comprehensive Guide for Developers* [online]. Medium. [visited on 2026-05-18]. Available from: `https://medium.com/@cibofdevs/mastering-cryptography-in-java-a-comprehensive-guide-for-developers-dac0c0da1d30`.
3. ORACLE CORPORATION. *Java Platform, Standard Edition Security Developer's Guide*. Java Language Security and Bytecode Verification. Oracle Help Center, 2025. Java SE 25. Available also from: `https://docs.oracle.com/en/java/javase/25/security/java-security-overview1.html#GUID-2EF91196-D468-4D0F-8FDC-DA2BEA165D10`. Accessed: 28 January 2026.
4. TUTORIALSPOINT. *Java Cryptography Tutorial* [online]. [visited on 2026-05-18]. Available from: `https://www.tutorialspoint.com/java_cryptography/index.htm`.
5. MENEZES, Alfred J.; OORSCHOT, Paul C. van; VANSTONE, Scott A. *Handbook of Applied Cryptography*. Boca Raton, FL: CRC Press, 1996. Discrete Mathematics and Its Applications. ISBN 978-0849385230.
6. BUELL, Duncan. *Fundamentals of Cryptography: Introducing Mathematical and Algorithmic Foundations*. Springer International Publishing, 2021. Undergraduate Topics in Computer Science.
7. AUMASSON, Jean-Philippe. *Serious Cryptography: A Practical Introduction to Modern Encryption*. San Francisco, CA: No Starch Press, 2017. ISBN 978-1-59327-826-7. Available also from: `https://theswissbay.ch/pdf/Books/Computer%20science/Cryptography/SeriousCryptography.pdf`. Accessed: 28 January 2026.
8. SSDEEP PROJECT. *ssdeep: Fuzzy Hashing API and Tool* [`https://ssdeep-project.github.io/ssdeep/index.html`]. 2026. Accessed: 2026-05-16.

9. TURAN, Meltem Sönmez; MCKAY, Kerry A.; CHANG, Donghoon; KANG, Jinkeon; KELSEY, John. *Ascon-Based Lightweight Cryptography Standards for Constrained Devices: Authenticated Encryption, Hash, and Extendable Output Functions*. 2025. NIST Special Publication (SP), 800-232. National Institute of Standards and Technology. Available from DOI: 10.6028/NIST.SP.800-232.

10. NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. *Module-Lattice-Based Key-Encapsulation Mechanism Standard*. 2024-08. Tech. rep., FIPS PUB 203. U.S. Department of Commerce. Available from DOI: 10.6028/NIST.FIPS.203.

11. NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. *Module-Lattice-Based Digital Signature Standard*. 2024-08. Tech. rep., FIPS PUB 204. U.S. Department of Commerce. Available from DOI: 10.6028/NIST.FIPS.204.

12. WAGENAAR, W. A. Generation of random sequences by human subjects: A critical survey of the literature. *Psychological Bulletin*. 1972, vol. 77, no. 1, pp. 65–72. Available from DOI: 10.1037/h0032060.

13. *Cryptographic hash function* [https://en.wikipedia.org/w/index.php?title=Cryptographic_hash_function]. Wikipedia, The Free Encyclopedia, 2025. Accessed: 14 December 2025.

14. MARLINSPIKE, Moxie. *The Cryptographic Doom Principle* [online]. 2011-12-13. [visited on 2026-03-12]. Available from: https://moxie.org/2011/12/13/the-cryptographic-doom-principle.html.

15. ERTAUL, Levent; KAUR, Manpreet; GUDISE, Venkata Arun Kumar R. Implementation and performance analysis of pbkdf2, bcrypt, scrypt algorithms. In: *Proceedings of the international conference on wireless networks (ICWN)*. The Steering Committee of The World Congress in Computer Science, Computer ..., 2016, p. 66.

16. OWASP FOUNDATION. *Password Storage Cheat Sheet* [https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html]. 2024. Accessed: 2026-05-15.

17. SARANTINOS, Nikolaos; BENZAÏD, Chafika; ARABIAT, Omar; AL-NEMRAT, Ameer. Forensic Malware Analysis: The Value of Fuzzy Hashing Algorithms in Identifying Similarities. In: *2016 IEEE Trustcom/BigDataSE/ISPA*. 2016, pp. 1782–1787. Available from DOI: 10.1109/TrustCom.2016.0274.

18. SEKANINA, Jan. *Fuzzy Hashing Project* [online]. 2025. [visited on 2026-03-12]. Term Project Report. Masaryk University, Faculty of Informatics. Available from: https://is.muni.cz/auth/el/fi/podzim2025/PA018/ode/term_project_reports/PA018_Semestral_Project.pdf. Course: PA018, Autumn 2025 Semester.

19. DAEMEN, Joan; RIJMEN, Vincent. *The Design of Rijndael: AES – The Advanced Encryption Standard*. Springer Science & Business Media, 2002. ISBN 978-3540425809.

20. JAYASINGHE, Rushika. *What is AEAD (Authenticated Encryption with Associated Data)?* [online]. 2024-02-23. [visited on 2026-03-12]. Available from: https://medium.com/@rushikajayasinghe/what-is-aead-authenticated-encryption-with-associated-data-17a5b2f42404.

21. BOZHKO, Andrey. *Properties of Authenticated Encryption with Associated Data (AEAD) Algorithms* [online]. 2024-02. [visited on 2025-03-12]. Internet-Draft, draft-irtf-cfrg-aead-properties-03. Internet Engineering Task Force. Available from: https://www.ietf.org/archive/id/draft-irtf-cfrg-aead-properties-03.html. Work in Progress.

22. MCGREW, David. *An Interface and Algorithms for Authenticated Encryption* [RFC 5116]. RFC Editor, 2008. Request for Comments, no. 5116. Available from DOI: 10.17487/RFC5116.

23. COMPUTERPHILE. *AES GCM (Advanced Encryption Standard in Galois Counter Mode) - Computerphile* [online]. YouTube, 2024-04-02. [visited on 2026-03-12]. Available from: https://www.youtube.com/watch?v=-fpVv_T4xwA.

24. MCGREW, David; VIEGA, John. The Galois/counter mode of operation (GCM). *submission to NIST Modes of Operation Process*. 2005, vol. 20, pp. 0278–0070.

25. CYBERSECURITY NONPROFIT (CSNP). *ML-KEM (Kyber) Explained: Complete Implementation Guide*. 2024. Available also from: https://qramm.org/learn/ml-kem-kyber-explained.html. Accessed: 2026-05-17.

26. NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. *Digital Signature Standard (DSS)*. 2013-07. Tech. rep., FIPS PUB 186-4. U.S. Department of Commerce. Available from DOI: 10.6028/NIST.FIPS.186-4.

27. LIU, Penghui; NIU, Yi; ZHONG, Xiaoxiong; WU, Jiahui; ZHANG, Weizhe; XUE, Kaiping; XIAO, Bin. *Efficient ML-DSA Public Key Management Method with Identity for PKI and Its Application*. 2026. Available from arXiv: 2603.25043 [cs.CR].

28. NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. *FN-DSA: Fast-Fourier Lattice-Based Digital Signature Algorithm*. 2024. Tech. rep., Draft FIPS PUB 206. U.S. Department of Commerce. Available also from: https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization.

29. NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. *Stateless Hash-Based Digital Signature Standard*. 2024-08. Tech. rep., FIPS PUB 205. U.S. Department of Commerce. Available from DOI: 10.6028/NIST.FIPS.205.

30. ARNOLD, Ken; GOSLING, James; HOLMES, David. *The Java programming language*. Addison Wesley Professional, 2005.

31. STATISTA. *Most used programming languages among developers worldwide as of 2024* [online]. Statista Inc., 2025. [visited on 2026-01-28]. Available from: https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/.

32. SUN MICROSYSTEMS. *Sun Announces New Family of Java Processors* [Press Release]. 1996. Available also from: https://web.archive.org/web/20070310235103/http://www.sun.com/smi/Press/sunflash/1996-01/sunflash.960123.10561.xml. Archived by the Wayback Machine on March 10, 2007.

33. JONG, Eduard de; HARTEL, Pieter; PEYRET, Patrice; CATTANEO, Peter. *Java Card: An analysis of the most successful smart card operating system to date* [online]. [visited on 2023-05-05]. Available from: https://ris.utwente.nl/ws/portalfiles/portal/5129057/java_card.pdf.

34. EVANS, Benjamin J.; FLANAGAN, David. *Java in a Nutshell: A Desktop Quick Reference*. 6th. Sebastopol, CA: O'Reilly Media, 2014. ISBN 978-1-449-37082-4. Available also from: http://www.r-5.org/files/books/computers/languages/java/main/Benjamin_Evans_David_Flanagan-Java_in_a_Nutshell_6th_ed-EN.pdf. Accessed: 28 January 2026.

35. ORACLE CORPORATION. *Java Cryptography Architecture (JCA) Oracle Providers Documentation*. Available also from: https://docs.oracle.com/en/java/javase/25/security/oracle-providers.html. Accessed: 2026-05-08.

36. THE LEGION OF THE BOUNCY CASTLE. *Bouncy Castle Cryptography APIs* [online]. 2026. [visited on 2026-05-08]. Available from: https://www.bouncycastle.org/.

37. THE CONSCRYPT AUTHORS. *Conscrypt: A Java Security Provider* [online]. 2026. [visited on 2026-05-08]. Available from: https://www.conscrypt.org/.

38. GOOGLE; CONSCRYPT CONTRIBUTORS. *Conscrypt: A Java Security Provider*. 2026. Available also from: https://github.com/google/conscrypt/. GitHub Repository.

39. SOTO BUENO, Alex. *What's new in Java 25 for developers* [online]. IBM Developer, 2025-09-24. [visited on 2026-05-18]. Available from: https://developer.ibm.com/articles/java-whats-new-java25/.

40. Available also from: https://tex.stackexchange.com/questions/114219/add-notes-to-latex-beamer.

41. THE APACHE SOFTWARE FOUNDATION. *Apache Maven* [online]. [visited on 2026-05-18]. Available from: https://maven.apache.org/.

42. CONSCRYPT PROJECT. *Conscrypt OpenJDK Uber, Version 2.5.2*. 2021. Available also from: `https://mvnrepository.com/artifact/org.conscrypt/conscrypt-openjdk-uber/2.5.2`. Maven Repository. Released March 30, 2021.

43. DEVANG JAIN. *Make a Self-Replicating Virus in Python* [`https://infosecwriteups.com/make-a-self-replicating-virus-in-python-bb29404e3f6b`]. [N.d.]. Accessed: 20 April 2026.

44. COLLECTIVE, levelblue. 2026. Available also from: `https://www.levelblue.com/blogs/spiderlabs-blog/why-we-should-probably-stop-visually-verifying-checksums`.

45. PERRIG, Adrian; SONG, Dawn. Hash visualization: A new technique to improve real-world security. In: *International Workshop on Cryptographic Techniques and E-Commerce*. 1999, vol. 25.

7 An appendix

In addition to this thesis, the presentation slides and their corresponding Overleaf project source files are appended. Furthermore, the digital repository (also available on GitLab), containing all source code described throughout this work, is included as an electronic attachment.