Static Verification of an Implementation of 5G-AKA

by

Negar Sabour

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2023

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Computer-aided cryptography offers a variety of tools that are essential for ensuring the security of cryptographic protocols. These tools can assist in designing the protocol, verifying its correctness during implementation, and detecting potential side-channel attacks. In the functional correctness branch of computer-aided cryptography, program specifications are compared against program behavior using a specification language. JML is a specification language for programs written in Java, and OpenJML is one of the verifiers that can check if a Java program meets its corresponding JML specifications.

In this study, we investigate the implementation of 5G-AKA and utilize the JML specification language to write specifications. We then use the OpenJML verifier to check whether the implementation satisfies these specifications, and make necessary changes to ensure that the implementation meets the specifications. Through this process, we perform a detailed analysis and uncover several flaws and bugs in the initial implementation. In other words, our study serves as a proof by construction that an implementation of the 5G-AKA specification can be automatically verified using the JML language.

Overall, this study serves as a case study for writing specifications in JML language from multiple documents written in English, highlighting the importance of using computer-aided cryptography tools to ensure the correctness of an implementation of cryptographic protocols.

Acknowledgements

I would like to thank my supervisors Professor Mahesh V. Tripunitara and Professor Werner Dietl, who generously provided knowledge and expertise.

I would also like to thank Professor Catherine Rosenberg who was involved in the validation survey for this research project.

Dedication

To my ...

Mom who gave me the gift of life,
Sister who gave me endless encouragement and
My husband who gave me endless love.

Table of Contents

Author's Declaration ii

Abstract iii

Acknowledgements iv

Dedication v

List of Figures xii

List of Tables xiv

List of Codes xv

List of Abbreviations xviii

List of Symbols xxi

1 Introduction 1

1.1 Computer-aided Cryptography 1

1.2 Functional correctness 2

1.3 JML 3

1.3.1 History 3

1.3.2 Where to begin . . . 4
1.3.3 JML tools . . . 4
1.3.3.1 Tools for parsing and type checking . . . 5
1.3.3.2 Tools for checking assertions at runtime . . . 5
1.3.3.3 Tools for checking assertions at compile time (static verification) . . . 5
1.3.3.4 Tools for generating specifications . . . 6
1.3.3.5 Tools for documentation . . . 6
1.4 OpenJML . . . 6
1.4.1 OpenJML architecture . . . 6
1.5 Outline . . . 7

2 Static verification of 5G-AKA **9**
2.1 Main entities in 5G-AKA . . . 9
2.2 Values stored in each entity before the initialization protocol begins . . . 10
2.3 5G-AKA initialization protocol . . . 11
2.3.1 SUCI structure . . . 12
2.3.1.1 SUPI Type . . . 12
2.3.1.2 Home Network Identifier . . . 13
2.3.1.3 Routing Indicator . . . 13
2.3.1.4 protection Scheme Id . . . 13
2.3.1.5 Home Network Public Key Id . . . 14
2.3.1.6 Scheme Output . . . 14
2.3.2 Scheme Output in SUCI . . . 16
2.3.2.1 Parameters used in ECIES encryption scheme . . . 17
2.3.2.2 Scheme Input for ECIES encryption . . . 18
2.3.2.3 Step 1 in ECIES encryption: Key Pair Generation . . . 19
2.3.2.3.1 Specification . . . 19

2.3.2.3.2 Bugs in implementation of ANSI-X9.63-KDF . . 19
2.3.2.3.3 Verification and refactoring 20
2.3.2.4 Step 2 in Elliptic Curve Integrated Encryption Scheme (ECIES) encryption: Key Agreement 28
2.3.2.4.1 Specification 28
2.3.2.4.2 Verification and refactoring 29
2.3.2.5 Step 3 in ECIES encryption: Key Derivation 36
2.3.2.5.1 Specification 36
2.3.2.5.2 Bugs in implementation of ANSI-X9.63-KDF . . 38
2.3.2.5.3 Verification and refactoring 40
2.3.2.6 Step 4 in ECIES encryption: Symmetric Encryption . . . 45
2.3.2.6.1 Specification 45
2.3.2.6.2 Verification and refactoring 45
2.3.2.7 Step 3 in ECIES decryption: Symmetric Decryption . . . 50
2.3.2.7.1 Specification 50
2.3.2.7.2 Verification and refactoring 50
2.3.2.8 Step 5 in ECIES encryption: Message Authentication Code (MAC) Function 51
2.3.2.8.1 Specification 51
2.3.2.8.2 Verification and refactoring 52
2.3.3 Static verification of edges of the process graph 55
2.3.3.1 First Solution 57
2.3.3.2 Second Solution 58
2.3.3.3 Third Solution 59
2.3.3.4 Fourth Solution 61
2.4 5G-AKA authentication protocol 64
2.4.1 Steps 1-4 of 5G-AKA authentication protocol 65
2.4.1.1 Random Challenge (RAND) 70

2.4.1.1.1 Specification . . . 70
2.4.1.1.2 Verification and Refactoring . . . 70
2.4.1.2 SQN . . . 71
2.4.1.3 Serving network name . . . 72
2.4.1.4 Authentication Management Field (AMF) . . . 72
2.4.1.5 K . . . 72
2.4.1.6 3GPP authentication and key generation functions f1, f1*, f2, f3, f4, f5 and f5* . . . 72
2.4.1.7 MAC . . . 73
2.4.1.7.1 Specification . . . 73
2.4.1.7.2 Verification and refactoring . . . 73
2.4.1.8 XRES . . . 76
2.4.1.8.1 Specification . . . 76
2.4.1.8.2 Verification and refactoring . . . 76
2.4.1.9 CK and AK . . . 77
2.4.1.9.1 Specification . . . 77
2.4.1.9.2 Verification and refactoring . . . 77
2.4.1.10 AK . . . 77
2.4.1.10.1 Specification . . . 77
2.4.1.10.2 Verification and refactoring . . . 78
2.4.1.11 Key Derivation Function (KDF) functions . . . 78
2.4.1.11.1 Converting an integer to bytes . . . 79
2.4.1.11.2 Converting an integer to two bytes . . . 81
2.4.1.12 (X)RES* . . . 82
2.4.1.12.1 Specification . . . 82
2.4.1.12.2 Verification and refactoring . . . 82
2.4.1.13 K_{AUSF} . . . 84
2.4.1.13.1 Specification . . . 84

2.4.1.13.2 Verification and refactoring 85
2.4.1.14 AUTN 89
2.4.1.14.1 Specification 89
2.4.1.14.2 Verification and refactoring 89
2.4.1.15 H(X)RES* 89
2.4.1.15.1 Specification 89
2.4.1.15.2 Verification and Refactoring 90
2.4.1.16 K_{SEAF} 91
2.4.1.16.1 Specification 91
2.4.1.16.2 Verification and refactoring 92
2.4.1.17 5G SE AV 92
2.4.1.17.1 Specification 92
2.4.1.17.2 Verification and Refactoring 92
2.4.2 Step 5 of 5G-AKA authentication protocol 93
2.4.3 Step 6 of 5G-AKA authentication protocol 93
2.4.4 Step 7 of 5G-AKA authentication protocol 94
2.4.4.1 Check AV freshness 94
2.4.4.2 AV Freshness Passed 96
2.4.4.3 Synchronization Failure 98
2.4.4.4 MAC Failure 100
2.4.5 Step 8 of 5G-AKA authentication protocol 100
2.4.6 Step 9 of 5G-AKA authentication protocol 100
2.4.7 Step 10 of 5G-AKA authentication protocol 101
2.4.8 Step 11 of 5G-AKA authentication protocol 101
2.4.9 Step 12 of 5G-AKA authentication protocol 101
2.4.10 Step 13 101

3 Software Improvements **102**

3.1 Naming . . . 102

3.1.1 Camel case . . . 102

3.1.2 Class names . . . 103

3.1.3 Method names . . . 103

3.1.4 Project name . . . 103

3.1.5 Package names . . . 104

3.2 Code Duplication . . . 104

3.3 Structure tests . . . 105

3.4 Using a build system . . . 106

3.4.1 Some benefits of a build system like gradle . . . 106

3.4.2 Gradle Wrapper . . . 107

3.4.3 Gradle in current project . . . 108

4 Conclusion **109**

5 Future Work and Improvements **110**

References **111**

List of Figures

1.1 Branches of Computer-aided Cryptography. Portions of this figure are based on the work of Barbosa et al. [26]. 1

1.2 Architecture of OpenJDK and OpenJML [63, pp. 5] 7

2.1 Main entities and their sub-entities in 5G authentication and key agreement (5G-AKA) 10

2.2 Initialization protocol: showing sub-entities [20, p. 39] 11

2.3 Initialization protocol: showing three main entities [27, p. 4] 11

2.4 SUCI structure [23, p. 20] 12

2.5 SUPI structure of type IMSI [23, p. 19] 13

2.6 Scheme Output structure for null scheme [23, p. 22] 15

2.7 Scheme Output structure for profile A [23, p. 23] 15

2.8 Scheme Output structure for profile B [23, p. 23] 15

2.9 ECIES encryption at User Equipment (UE) [20, p. 207] 16

2.10 ECIES decryption at Home Network (HN) [20, p. 208] 17

2.11 Scheme Output part of SUCI for SUPI type IMSI and null scheme [18, p. 540] 18

2.12 An example of converting decimal digits of Mobile Subscriber Identification Number (MSIN) to hex digits of Scheme Input 19

2.13 UML diagram showing relation between interface AlgorithmParameterSpec and two classes NamedParameterSpec and ECGenParameterSpec 22

2.14 ANSI-X9.63-KDF inputs and output 36

2.15 ANSI-X9.63-KDF inputs and output that is used for ECIES encryption scheme in [20, pp. 207] . . . 37
2.16 Example of a protocol section with two verified nodes . . . 56
2.17 Weakness of Just Verifying Nodes . . . 56
2.18 Solution1 for Verifying Edges . . . 57
2.19 Solution2 for Verifying Edges . . . 58
2.20 Solution3 for Verifying Edges . . . 60
2.21 simplified version of solution3 for Verifying Edges that is not acceptable . . . 61
2.22 Solution4 for Verifying Edges . . . 62
2.23 ECIES encryption for generating Scheme Output _ Data Flow diagram . . . 63
2.24 Authentication protocol: showing sub-entities [20, p. 43] . . . 64
2.25 Authentication protocol: showing main entities [69, p. 13] . . . 65
2.26 Authentication protocol: showing main entities and exceptional cases [27, p. 5] . . . 67
2.27 Steps One to Four in 5G-AKA Authentication protocol _ Inputs and Outputs 68
2.28 Steps One to Four in 5G-AKA Authentication protocol _ Data Flow diagram 69
2.29 Steps Seven Check AV freshness in 5G-AKA Authentication protocol _ Inputs and Outputs . . . 94
2.30 Steps Seven Check AV freshness in 5G-AKA Authentication protocol _ Data Flow diagram . . . 95
2.31 Continuation of step seven in the 5G-AKA authentication protocol when AV freshness validation is successful. _ Data Flow diagram . . . 97
2.32 Continuation of step seven in the 5G-AKA authentication protocol when AV Synchronization failure occurs _ Inputs and Output . . . 98
2.33 Continuation of step seven in the 5G-AKA authentication protocol when AV Synchronization failure occurs _ Data Flow diagram . . . 99
2.34 Step nine in 5G-AKA Authentication protocol _ Data Flow diagram . . . 100

List of Tables

1.1 specification languages 3
1.2 JML tools 4

2.1 Values stored in each entity before the protocol begins 10
2.2 Parameters used in ECIES encryption scheme for both profiles A and B. . 18
2.3 3GPP authentication and key generation functions f1, f1*, f2, f3, f4, f5 and f5* 72

3.1 Class names 103

List of Codes

2.1 complete java.security.spec.AlgorithmParameterSpec.jml: Adding one model field . . . 22

2.2 create java.security.spec.NamedParameterSpec.jml . . . 23

2.3 create java.security.spec.ECGenParameterSpec.jml . . . 23

2.4 complete java.security.KeyPairGenerator.jml: Adding one model field and two model methods . . . 24

2.5 complete java.security.KeyPairGenerator.jml: getInstance(java.lang.String algorithm) . . . 24

2.6 complete java.security.KeyPairGenerator.jml: initialize(AlgorithmParameterSpec alg) . . . 25

2.7 complete java.security.KeyPairGenerator.jml: generateKeyPair() . . . 26

2.8 implementation and verification of Step1 of ECIES encryption: Key Pair Generation . . . 27

2.9 complete javax.crypto.KeyAgreement.jml: getInstance(String s) . . . 29

2.10 complete javax.crypto.KeyAgreement.jml: getInstance (java.lang.String s, java.security.Provider provider) . . . 30

2.11 create java.security.Provider.jml . . . 31

2.12 create org.bouncycastle.jce.provider.BouncyCastleProvider.jml . . . 31

2.13 complete javax.crypto.KeyAgreement.jml: init(java.security.Key key) . . . 31

2.14 complete javax.crypto.KeyAgreement.jml: doPhase(java.security.Key key, boolean b) . . . 33

2.15 complete javax.crypto.KeyAgreement.jml: generateSecret() . . . 33

2.16 implementation and verification of Step 2 of ECIES encryption: Key Agreement . . . 34
2.17 Implementation of ANSI-X9.63-KDF algorithm from [69, pp. 27] (used for ECIES encryption scheme for both profiles A and B) . . . 38
2.18 steps 1 and 2 of ANSI-X9.63-KDF algorithm used in key derivation step in ECIES encryption scheme . . . 41
2.19 complete java.security.MessageDigest.jml: Adding one model field and two model methods . . . 42
2.20 complete java.security.MessageDigest.jml: getInstance(java.lang.String s) . . . 42
2.21 complete java.security.MessageDigest.jml: digest(byte[] bb) . . . 43
2.22 implementation and verification of Step3 of ECIES encryption: Key Derivation 43
2.23 temp . . . 45
2.24 complete javax.crypto.Cipher.jml: getInstance(java.lang.String s) . . . 46
2.25 rewrite javax.crypto.Cipher.jml: init(int n, Key key, AlgorithmParameterSpec spec) . . . 46
2.26 complete javax.crypto.Cipher.jml: doFinal(byte[] b) . . . 47
2.27 complete javax.crypto.spec.IvParameterSpec.jml . . . 49
2.28 implementation and verification of Step 4 of ECIES encryption: Symmetric Encryption . . . 49
2.29 implementation and verification of Step 3 of ECIES decryption: Symmetric Decryption . . . 51
2.30 complete javax.crypto.Mac.jml: getInstance(java.lang.String s) . . . 52
2.31 complete javax.crypto.Mac.jml: init(java.security.Key key) . . . 52
2.32 complete javax.crypto.Mac.jml: doFinal(byte[. . . 53
2.33 implementation and verification of Step 5 of ECIES encryption: MAC Function 55
2.34 implementation of the RAND generation from [69] . . . 70
2.35 complete java.util.Random.jml . . . 71
2.36 implementation and verification of RAND generation . . . 71
2.37 implementation and verification of f1 . . . 73

2.38 implementation and verification of the updated version of f1 74
2.39 implementation and verification of f2 76
2.40 implementation and verification of f3 and f4 77
2.41 implementation and verification of f5 78
2.42 implementation and verification of Converting an integer to bytes 79
2.43 implementation and verification of Converting an integer to two bytes . . . 81
2.44 implementation and verification of (X)RES* 83
2.45 implementation of xor function in [69] 85
2.46 implementation and verification of xor two byte arrays with same length . 86
2.47 implementation and verification of K_{AUSF} 87
2.48 implementation and verification of AUTN 89
2.49 implementation and verification of H(X)RES* 90
2.50 implementation and verification of K_{SEAF} 92
2.51 implementation and verification of 5G Serving Environment Authentication Vector (5G SE AV) 93
2.52 implementation and verification of checking the MSB bit of given byte . . . 96

List of Abbreviations

5G HE AV 5G Home Environment Authentication Vector 65

5G SE AV 5G Serving Environment Authentication Vector 65, 91, 92, 99

5G-AKA 5G authentication and key agreement 9, 10, 64, 65, 77, 78

5G-GUTI 5G Global Unique Temporary Identifier 11

5GC 5G Core 77

AK Anonymity Key 71, 75–77, 83–85, 87, 90, 93, 97

AMF Access and Mobility Management Function 92

AMF Authentication Management Field 71, 72, 87, 93

ARPF Authentication credential Repository and Processing Function 9, 64, 65

AUSF Authentication Server Function 9, 65, 92

AUTN Authentication Token 64, 65, 73, 87, 91–93

CAC Computer-aided Cryptography 1, 2

CK Cipher Key 71, 75, 76, 81, 83, 84, 90, 95

EAP-AKA Extensible Authentication Protocol authentication and key agreement 9

EC Elliptic Curve 18, 28

ECCDH Elliptic Curve (EC) Cofactor Diffie-Hellman 18, 28, 30

ECIES Elliptic Curve Integrated Encryption Scheme 16–21, 27, 28, 33, 36, 40, 41, 43, 45, 49–52, 54

GCI Global Cable Identifier 12

GLI Global Line Identifier 12

HN Home Network 9, 10, 16, 17, 20, 64, 65, 71, 92, 100, 102

IK Integrity Key 71, 81, 83, 84, 90, 95

IMSI International Mobile Subscription Identity 12, 13, 18

JML Java Modeling Language 3–6, 109

K Authentication Key 10, 71, 72, 75, 93, 97

KDF Key Derivation Function 18, 71, 75–78, 80, 83, 88, 90

MAC Message Authentication Code 18, 37, 45, 51, 54, 71, 72, 75, 87, 93

MCC Mobile Country Code 12, 13

MNC Mobile Network Code 12

MSIN Mobile Subscriber Identification Number 12, 18, 19

NID Network Identifier 12

$\mathbf{pk_{HN}}$ public key of HN 10

PLMN Public Land Mobile Network 12

$\mathbf{prik_{HN}}$ private key of HN 10

RAND Random Challenge 64, 65, 69–72, 75, 81, 88, 89, 91–93, 97, 99

SEAF Security Anchor Function 9, 65, 92, 99, 100

SIDF Subscription Identifier Deconcealing Function 9

SN Serving Network 9, 10, 12, 65, 92, 99, 100, 102

SNname Serving Network (SN) name 10, 81, 83, 84, 90

SNPN Stand-alone Non-Public Network 12

SQN Sequence Number 10, 70–72, 83–85, 87, 90, 93, 97

SQN_{HN} Sequence Number (SQN) stored at HN 10

SQN_{UE} SQN stored at UE 10

SUCI Subscription Concealed Identifier 11–13, 65, 100

SUPI Subscription Permanent Identifier 10, 12, 13, 18, 65, 92

UDM Unified Data Management 9, 64, 65

UE User Equipment 9–11, 16, 70, 71, 92, 102

UML Unified Modeling Language 21

USIM Universal Subscriber Identity Module 9

List of Symbols

|| Concatenation 37, 38, 71, 72, 78, 81, 84, 87, 88, 90, 102

$\oplus$ Exclusive or 84, 87, 93

Chapter 1

Introduction

1.1 Computer-aided Cryptography

Computer-aided Cryptography (CAC) offers machine-verifiable techniques to assist with various stages of cryptographic algorithm development. The key areas of focus in CAC are design-level security, functional correctness, efficiency, and side-channel resistance [26, pp. 1,13]. Figure 1.1 illustrates branches of CAC.

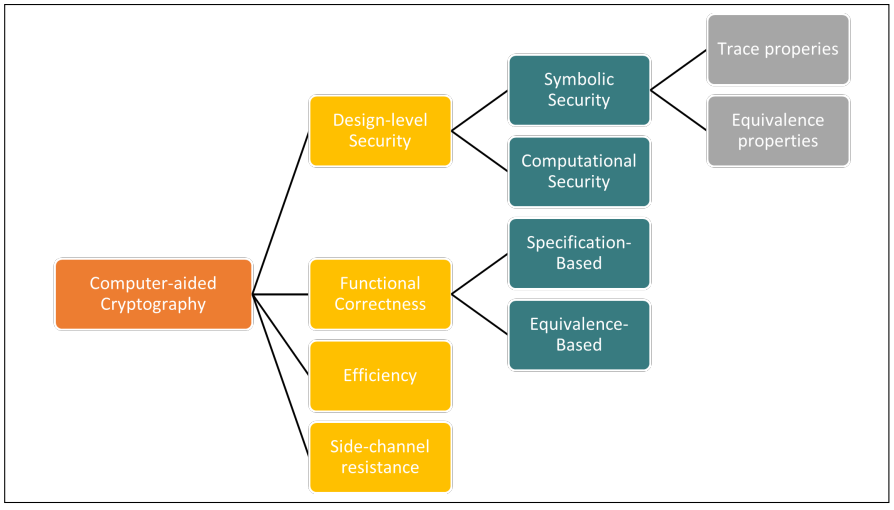


Figure 1.1: Branches of Computer-aided Cryptography. Portions of this figure are based on the work of Barbosa et al. [26].

At the design level, proving the security of cryptographic schemes through mathematical arguments using pen and paper may not always be ideal, as it can be error-prone and might simplify the system to a point where the actual powers of an attacker are not captured. To overcome these limitations, the concept of design-level security has emerged, providing a reliable evaluation of complex designs that are beyond the scope of the traditional pen-and-paper examination. The formal methods and cryptography communities have both contributed to the development of two areas in this field: symbolic security and computational security [26, pp. 1–6]. Tamarin, a symbolic security tool, has been applied to analyze actual protocols such as TLS 1.3 draft 10 [43] and 5G-AKA [27]. Tamarin's analysis of 5G-AKA in [27] exposed certain security issues, for which minor adjustments were proposed to enhance the protocol's security.

The efficiency aspect of CAC includes tools that analyze the performance of cryptographic algorithms. Even minor enhancements can have a significant impact on the system's effectiveness, particularly when the algorithm is used on a large scale. Therefore, efficiency becomes increasingly important in such scenarios.

At the implementation level, there are tools available to verify whether an implementation adheres to the "constant time" principles. Moreover, some of these tools have the ability to modify code that violates these principles [26, pp. 10].

Functional correctness will be discussed in Section 1.2.

1.2 Functional correctness

Functional correctness tools aim to verify whether an implementation satisfies its specification, which means ensuring that it performs as intended. Other techniques for ensuring software quality are auditing and testing, but these have limitations, such as the possibility of human errors and biases in auditing and the challenge of testing the entire input space [26, pp. 7]. While functional correctness does not have the same limitations, it assumes the correctness of libraries and relies on verification tools that themselves may contain bugs [26, pp. 7].

Functional correctness can be divided into two areas: comparing the implementation to a reference one for equivalence and analyzing whether it meets its specification. An example of a tool that evaluates specification satisfaction is Key, which exposed a bug in the Java standard library's TimSort implementation [44].

1.3 JML

Since 1998, JML has served as a Formal specification Language used to write program specifications in Java. A comprehensive list of specification languages and their corresponding programming languages is presented in Table 1.1.

Table 1.1: specification languages

specification languages	supported programming language	ref
JML	Java	[6]
SPARK	Ada	[63]
Dafny	Dafny	[61]
Stainless (previously Leon)	Scala	[60]
Larch	C++	[55]

1.3.1 History

The development of Java Modeling Language (JML) was initiated by Gary Leavens and his team of colleagues and students at Iowa State University [30, pp. 1]. Drawing from their prior experience with the Larch/C++ project, they began working on a specification language for Java [25, pp. 194]. Unlike C++, Java at the time was a more straightforward alternative due to its lack of pointers [60, pp. 314].

In 1998, Leavens worked with Albert Baker and Clyde Ruby from Iowa State University during the initial development of JML design [58].

Around the same period, a group led by Rustan Leino and Greg Nelson at the DEC/-Compaq research center created a formal specification language for Java that shares many similarities with JML [45].

During that time period, a group led by Bart Jacobss at the Verificard company in Nijmegen, Netherlands, created LOOP, a tool that utilized JML [60, pp. 315]. At the same time, Michael Ernst developed Daikon, a program that generated invariants for JML [60, pp. 315].

Patrice Chalin, a member of the faculty at Concordia University, played a major role in advancing RAC in JML [60, pp. 315]. Peter Muller, who has been a full professor at ETH Zurich since 2008, made significant contributions to enhancing the semantics of JML [60, pp. 315]. David Cok joined the JML project as an open-source developer in 2000 [60, pp. 315].

For a comprehensive list of contributors to JML, please refer to [56]. If you're interested in delving deeper into the history of JML, you may find [60, pp. 314–316] and [25, pp. 194–195] informative.

1.3.2 Where to begin

Information on how to use JML can be obtained from various sources. The most recent JML reference manual can be found in [6], and practical guidance on JML can be found in Chapter 16 of the Key book [25]. The JML project website [56] and the tutorial section of the OpenJMLs website [13] also offer helpful information on using JML. Additionally, [59] provides a tutorial for JML.

1.3.3 JML tools

Similar to programming languages, specification languages like JML require a range of tools. Table 1.2 provides an overview of the various JML tools.

Table 1.2: JML tools

Area	Tools
parsing and type checking	JML checker
checking assertions at runtime	OpenJML [66] jmlunit [72] jmlc [31]
checking assertions at compile time	OpenJML [66] KEY [25] VerCors [62] VeriFast [51] ESC/Java [30] ESC/Java2 [30] LOOP [30] JACK [30]
generating specifications	jmlspecs [30] Houdini [30] Daikon [30]
documentation	jmldoc [7]

1.3.3.1 Tools for parsing and type checking

The JML checker is in charge for parsing and type checking of both programs in Java and specifications in JML [30, pp. 3]. This ensures that there are no errors or typos in the JML annotations.

1.3.3.2 Tools for checking assertions at runtime

Tools that provide checking JML assertions at runtime, test the behaviour of the program for a specific run. The process of Runtime Assertion Checking involves compiling both the program and JML annotations into class files [66, pp. 74]. Then running those class files result in checking whether the assertions hold [30, pp. 3].

Examples of Tools that provide runtime assertion checking are OpenJML [66, pp. 74], jmlc [31] and jmlunit [72], the latter of which combines unit testing with runtime assertion checking [30, pp. 3].

1.3.3.3 Tools for checking assertions at compile time (static verification)

Compile-time assertion checking tools are used to verify the correctness of an implementation against its specification. Unlike runtime assertion checking, program verification evaluates the program's behaviour for all valid inputs, rather than a specific run. The level of automation and complexity that these tools can handle depends on the trade-off between them [30, pp. 4]. Several examples of tools that offer compile-time assertion checking include OpenJML [66], KEY [25], ESC/Java, ESC/Java2, LOOP, and JACK.

ESC/Java, developed at Compaq Research, works for a subset of JML notations and can detect some errors in the program, such as null dereferencing and index out of range [30, pp. 7]. Its input is a Java program with JML annotations, and its output is any error found in the specification or Java program [30, pp. 7]. ESC/Java2 aimed to enhance ESC/Java by supporting all JML annotations and was used in the Dutch parliament's voting system in 2004 [30, pp. 9,15].

LOOP, developed at the University of Nijmegen, was initially designed to investigate object-oriented semantics and later appeared as a static verification tool [30, pp. 9-11]. JACK, initially developed at the search lab of Gemplus and then at INRIA, aimed to achieve a high degree of automation and be prover-independent but is not publicly available [30, pp. 11–12].

1.3.3.4 Tools for generating specifications

Software developers can benefit greatly from using specification-generating tools since creating specifications manually can be time-consuming and prone to errors [30, pp. 12]. Examples of such tools include jmlspecs, Houdini, and Daikon. Jmlspecs takes Java source files as input and generates a skeleton specification [30, pp. 4], while Daikon observes a program's behavior during runtime to identify likely invariants that it automatically adds to the program [30, pp. 12]. Houdini generates annotations that are later verified by ESC/Java [30, pp. 13].

1.3.3.5 Tools for documentation

Jmldoc [7] is a software tool designed for documenting JML specifications in a similar way to Javadoc [30, pp. 14]. Unlike Javadoc, Jmldoc can interpret JML specifications and integrate them into the resulting HTML pages [7].

1.4 OpenJML

OpenJML is an open-source tool, based on the OpenJDK Java compiler, that verifies whether Java program implementations satisfy their corresponding JML specifications [6, pp. 2]. The tool's source code and releases can be found on its GitHub repository [65], which is accessible to the public. For those seeking to gain familiarity with OpenJML, OpenJML Reference Manual [6] and the tutorial section of the OpenJML website [13] are both recommended resources to consider.

1.4.1 OpenJML architecture

As illustrated in Figure 1.2, the architecture of OpenJML and OpenJDK involves several steps, including input scanning, parsing, name resolution, and type-checking. After generating an Abstract Syntax Tree (AST), JML specifications are incorporated into the AST as assertions and assumptions, resulting in a JML-enhanced AST that is used for both runtime assertion checking and static verification. To perform runtime assertion checking, the JML-enhanced AST is sent to the code generation phase to produce byte-code output [63, pp. 4]. The compiled code is then run as usual [63, pp. 3]. For static verification, the JML-enhanced AST is passed to the JML Local encoding process, which generates

an SMT-LIB equivalent of the AST embodying the semantics of Java [63, pp. 4]. The SMT-LIB equivalent of the AST is then input to an SMT Solver, which determines the validity of the assertions provided in the input [63, pp. 4].

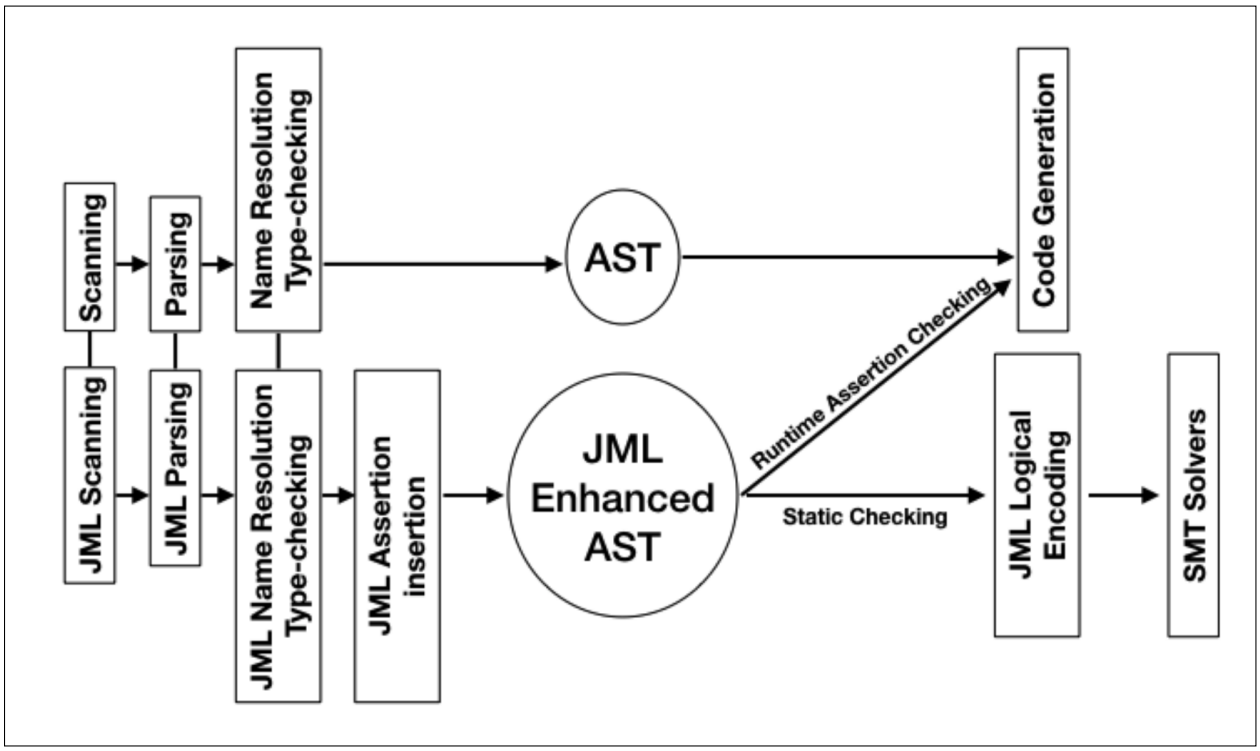


Figure 1.2: Architecture of OpenJDK and OpenJML [63, pp. 5]

1.5 Outline

In Chapter 2, we will conduct a static verification of the 5G-AKA protocol, examining both the initialization protocol (Section 2.3) and the authentication protocol (Section 2.4). In Section 2.3.3, we will propose several approaches for capturing edges in a process graph for static verification, and provide a comparison of these approaches.

In Chapter 3, we will provide details about software improvements that were made to the project. These improvements were designed to enhance the overall understandability and efficiency of the project.

We draw conclusions in Chapter 4 and provide a few suggestions for future works in Chapter 5.

Chapter 2

Static verification of 5G-AKA

2.1 Main entities in 5G-AKA

There are three main entities participating in 5G-AKA: Subscriber (combination of user's phone (also called UE) and Universal Subscriber Identity Module (USIM)), SN and HN. In [20] each SN and HN has several sub-entities, but for the sake of simplicity in [27, p. 3] and [69, pp. 10–12] Subscriber, SN and HN are chosen as three main logical entities.

Figure 2.1 shows three main entities and their sub-entities in 5G-AKA. Security Anchor Function (SEAF) is a sub-entity of SN and Authentication Server Function (AUSF), Unified Data Management (UDM), Authentication credential Repository and Processing Function (ARPF) and Subscription Identifier Deconcealing Function (SIDF) are sub-entities of HN. Subscriber and SN communicate through insecure channel. However, SN and HN communicate via authenticated channel [27, p. 3].

5G-AKA and Extensible Authentication Protocol authentication and key agreement (EAP-AKA) are two protocols that allow SNs and subscribers mutually authenticate each other [27, p. 3]. First an initialization protocol is run and at the end of that HN decides to choose 5G-AKA or EAP-AKA [27, p. 3].

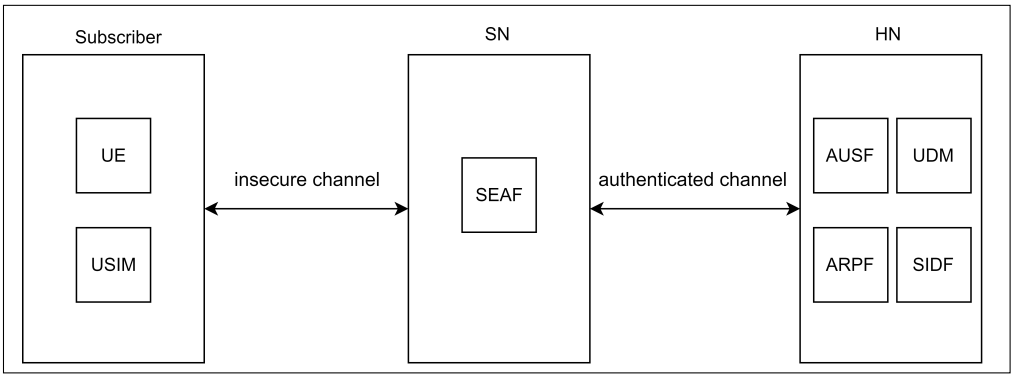


Figure 2.1: Main entities and their sub-entities in 5G-AKA

2.2 Values stored in each entity before the initialization protocol begins

Table 2.1 demonstrates a few values stored in each entity before the initialization protocol begins [27, pp. 3–4]. In [69], Authentication Key (K) and the public key of HN (pk_{HN}) were erroneously taken to be the same values. We separate them and set K to a randomly generated value of 32 bytes length stored in both HN and subscriber.

Table 2.1: Values stored in each entity before the protocol begins

Entity	Variable	Note
Subscriber	Subscription Permanent Identifier (SUPI)	SUPI is a unique and permanent identity for subscriber. It "also contains idHN and therefore identifies both a subscriber and its HN" [27, p. 3].
	K	long-term secret symmetric key, between subscribers and their HNs [27, p. 3])
	SQN stored at UE (SQN_{UE})	a counter
	pk_{HN}	public asymmetric key
SN	SN name (SNname)	SNs identity
HN	SUPI	For each subscriber, HN stores these three fields.
	K	
	SQN stored at HN (SQN_{HN})	
	private key of HN ($prik_{HN}$)	private key of HN

2.3 5G-AKA initialization protocol

The initialization protocol is the beginning of an authentication process and involves selecting the authentication method [20, p. 39]. Figure 2.2 depicts the initialization protocol, showing sub-entities, while Figure 2.3 shows three main entities involved in the initialization protocol.

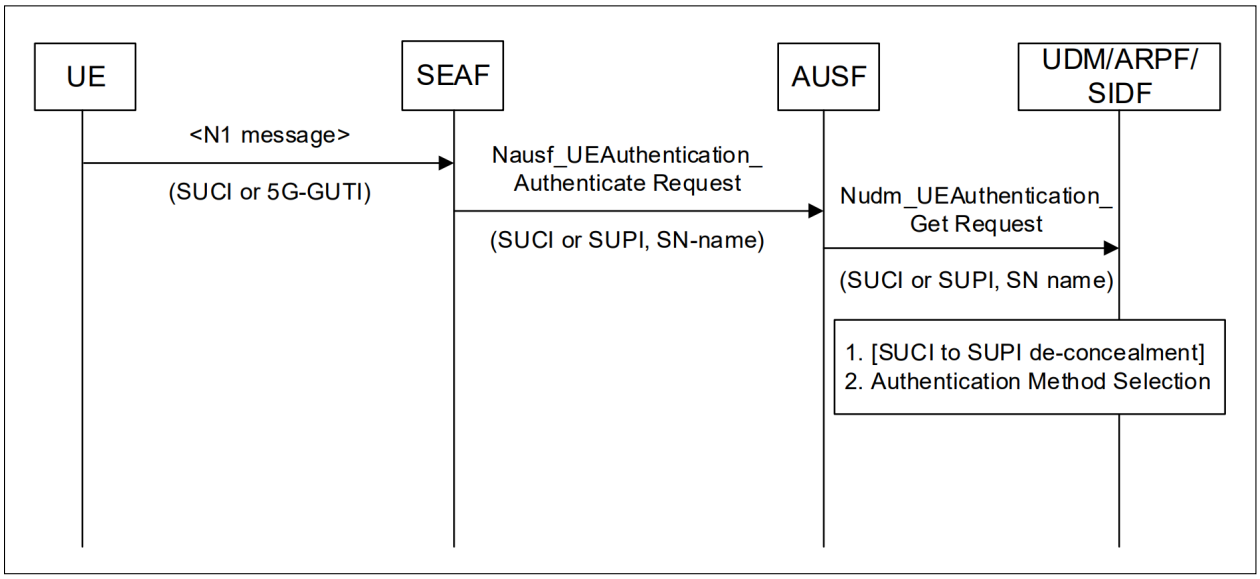


Figure 2.2: Initialization protocol: showing sub-entities [20, p. 39]

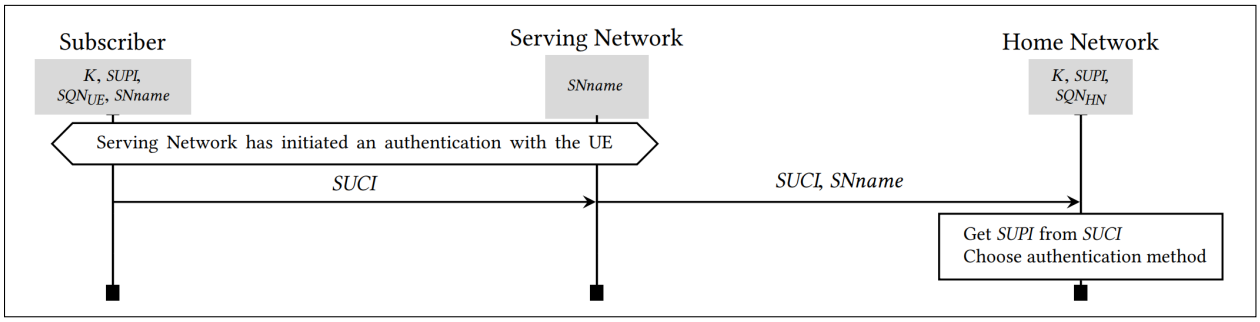


Figure 2.3: Initialization protocol: showing three main entities [27, p. 4]

As Figure 2.2 indicates, first, UE sends Subscription Concealed Identifier (SUCI) or 5G Global Unique Temporary Identifier (5G-GUTI) to SN in N1 message. To implement the N1 message [69, p. 39] chose a NAS-like message (see [69, p. 39] for details of the structure).

2.3.1 SUCI structure

Figure 2.4 illustrates structure of SUCI [23, p. 20].

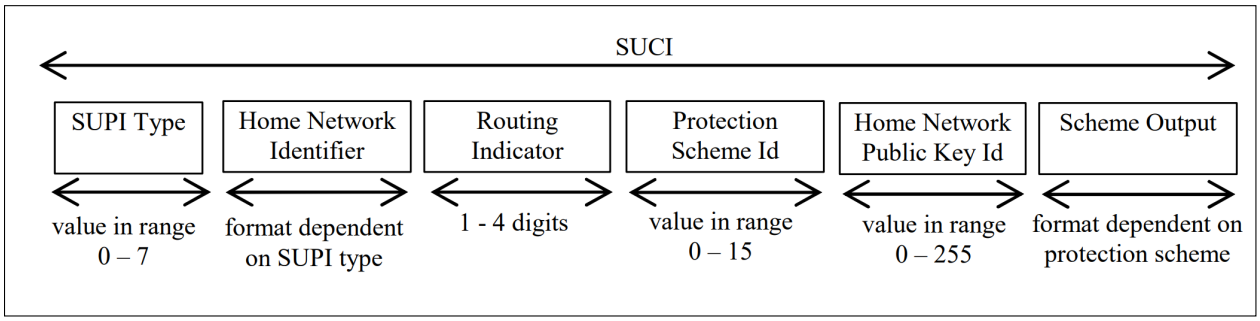


Figure 2.4: SUCI structure [23, p. 20]

2.3.1.1 SUPI Type

First Component of SUCI is SUPI Type. SUPI Type supported in implementation in [69] is International Mobile Subscription Identity (IMSI) (corresponds to the value 0 [23, p. 20]). Other SUPI Types are Network Specific Identifier, Global Line Identifier (GLI) and Global Cable Identifier (GCI).

Figure 2.5 displays SUPI structure of type IMSI [23, p. 19]. The IMSI is a string of decimal digits with a maximum length of 15 digits. Three fields that make the IMSI are Mobile Country Code (MCC), Mobile Network Code (MNC), and MSIN.

MCC is a three digit decimal value [23, p. 21]), uniquely identifies the mobile subscription's country [23, p. 20]). MNC is a two or three decimal digit value [23, p. 21]).The length of MNC is based on the MCC value [23, p. 20]). MNC either specifies the mobile subscription's home Public Land Mobile Network (PLMN) or, in conjunction with the MCC and Network Identifier (NID), the Stand-alone Non-Public Network (SNPN) [23, p. 20]). MSIN is a string of decimal digits with a maximum length 10, indicates a mobile subscription of PLMN or SNPN.

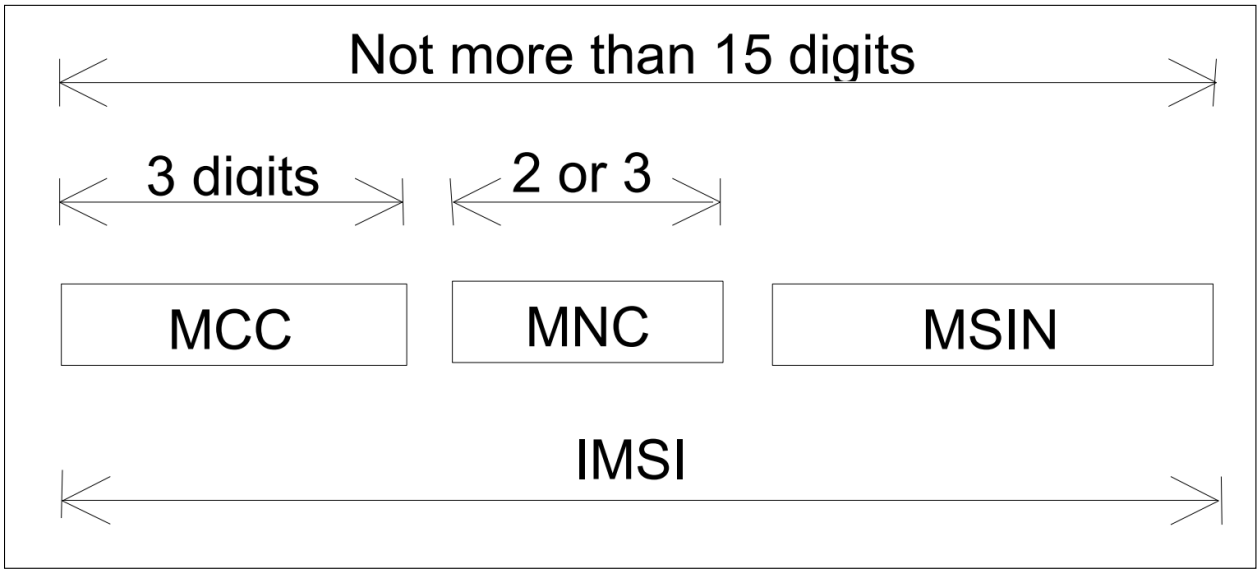


Figure 2.5: SUPI structure of type IMSI [23, p. 19]

2.3.1.2 Home Network Identifier

Second component of SUCI is Home Network Identifier. The format of the Home Network Identifier depends on the type of SUPI. For SUPI of type IMSI, Home Network Identifier consists of two parts MCC and glsMNC [23, p. 21].

2.3.1.3 Routing Indicator

.

Routing Indicator is a one to four-digit value stored in USIM. If the Routing Indicator is not specified on the USIM, it must be set to 0 (only one decimal digit of 0 in it) [23, p. 21].

2.3.1.4 protection Scheme Id

The Protection Scheme ID is a value ranging from 0 to 15 [23, p. 21]. There are three Protection Scheme IDs available [20, p. 206]:

- Null scheme: 0x0
- Profile A: 0x1
- Profile B: 0x2

Values between 0x3 and 0xB are reserved for future use, while values between 0xC and 0xF are for proprietary protection schemes [20, p. 206].

The Scheme Output length for the three protection schemes is as follows [20, p. 206]:

- Null scheme: size of input (MSIN in case of IMSI)
- Profile A: 256-bit public key + 64-bit MAC + size of input
- Profile B: 264-bit public key + 64-bit MAC + size of input

2.3.1.5 Home Network Public Key Id

Home Network Public Key Id is "an identifier used to indicate which public/private key pair is used for SUPI protection and de-concealment of the SUCI" [20, p. 20]. Home Network Public Key Id, protection Scheme Id and Home Network Public Key are stored in USIM [20, p. 27]. Home Network Public Key Id takes a value between 0 and 255 and it is set to 0 if and only if null scheme is selected [23, p. 21].

2.3.1.6 Scheme Output

The structure of Scheme Output depends on Protection Scheme(see Figure 2.6, Figure 2.7 and Figure 2.8).

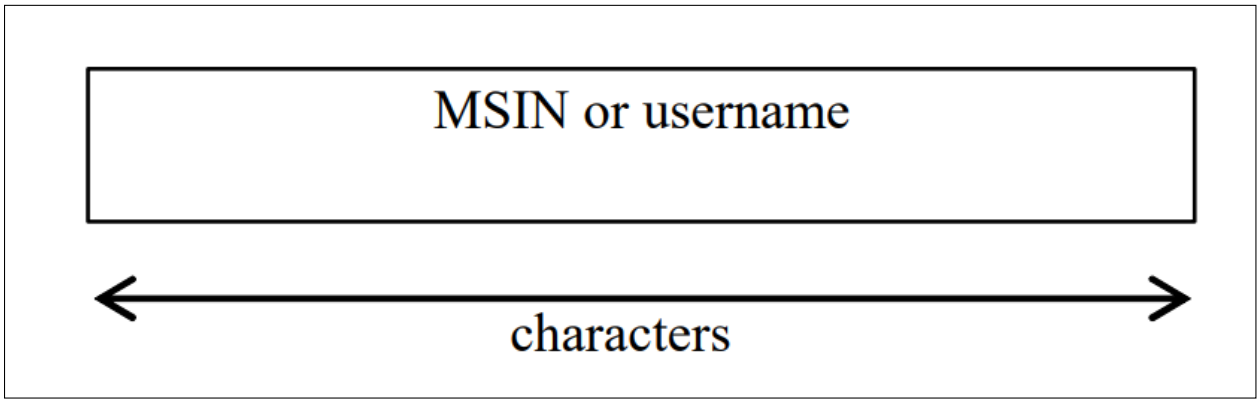


Figure 2.6: Scheme Output structure for null scheme [23, p. 22]

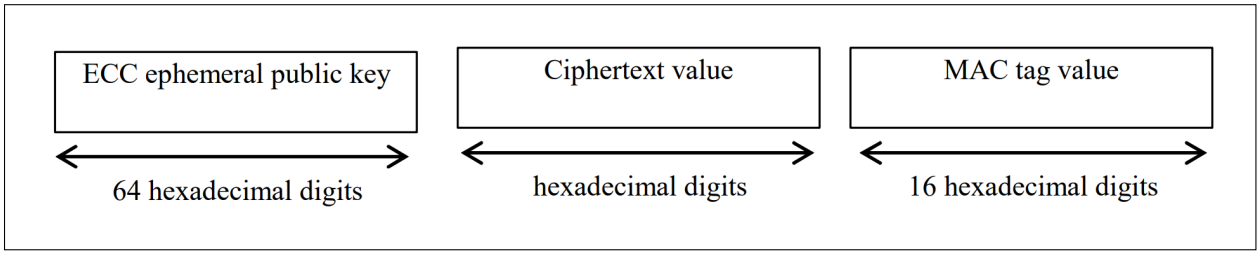


Figure 2.7: Scheme Output structure for profile A [23, p. 23]

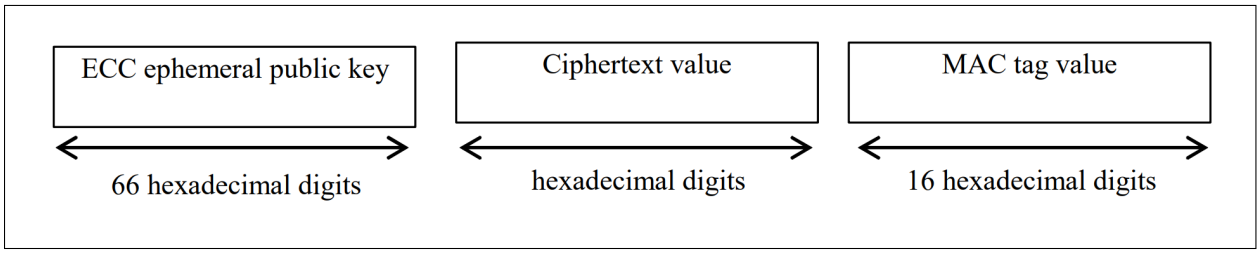


Figure 2.8: Scheme Output structure for profile B [23, p. 23]

2.3.2 Scheme Output in SUCI

In Section 2.3.1.6 we saw the structure of Scheme Output for profiles A, B and null scheme. Here we will see how Scheme Output is generated for profiles A and B by using ECIES encryption at UE from values including Scheme Input (also called plaintext block). Moreover, we will go through the ECIES decryption process which is used to extract plaintext block from Scheme Output. Figure 2.9 demonstrates ECIES encryption at UE while Figure 2.10 illustrates ECIES decryption at HN.

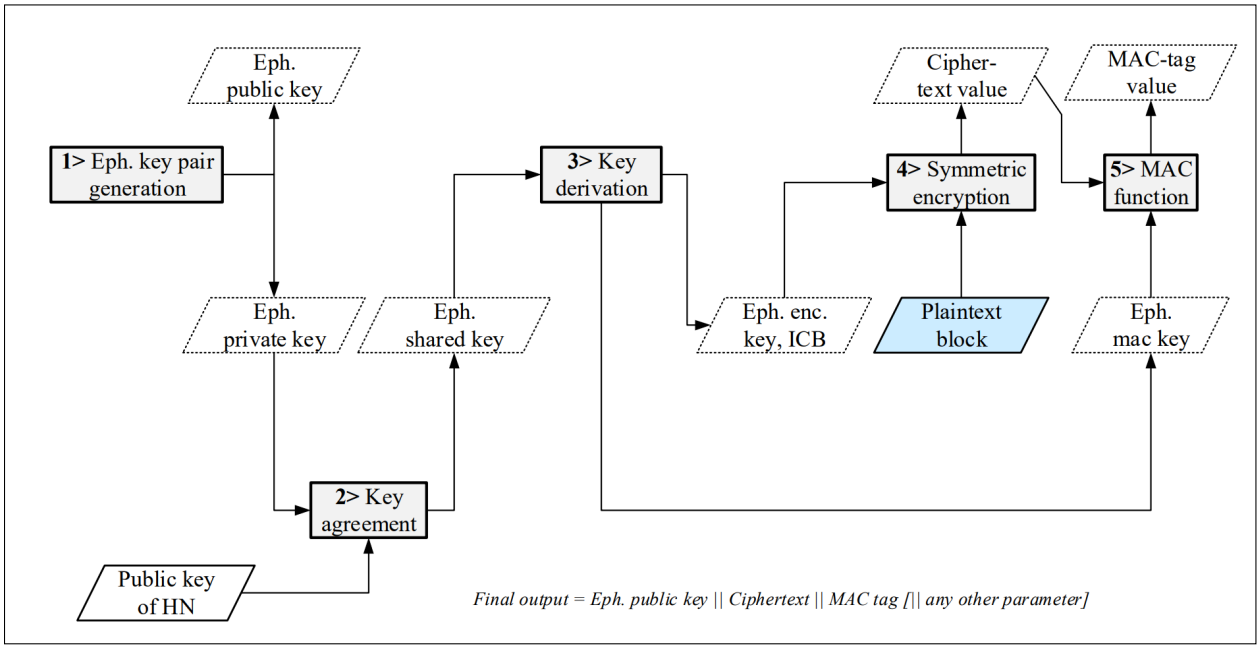


Figure 2.9: ECIES encryption at UE [20, p. 207]

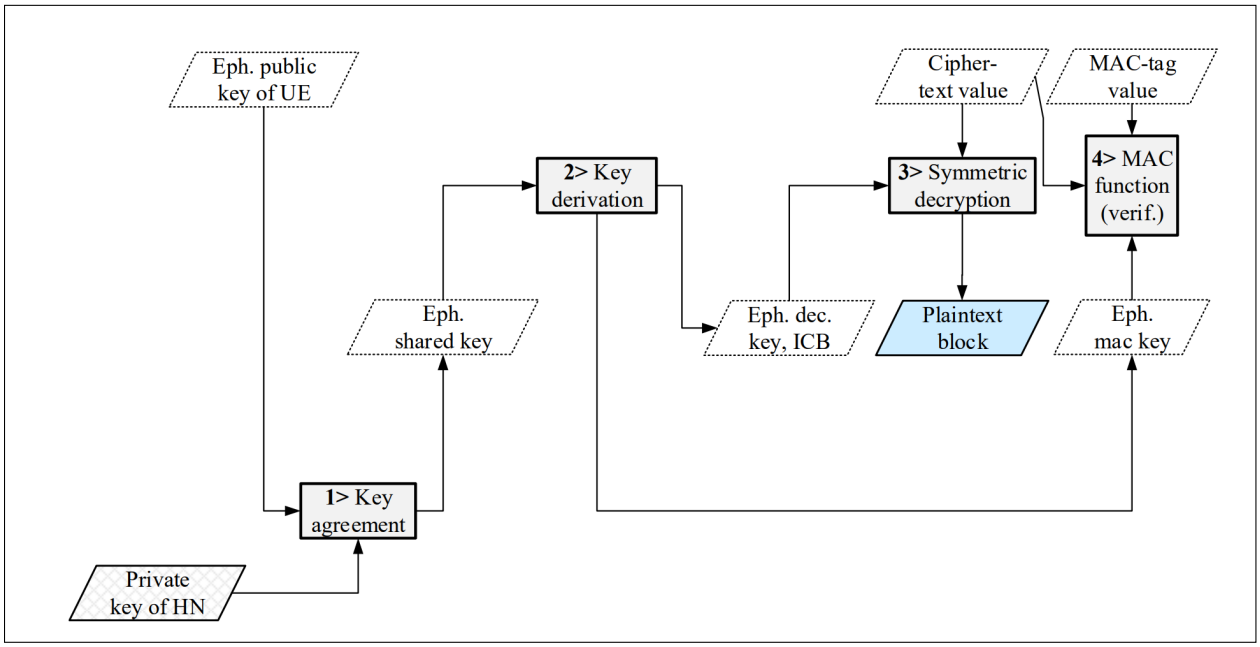


Figure 2.10: ECIES decryption at HN [20, p. 208]

2.3.2.1 Parameters used in ECIES encryption scheme

Table 2.2 shows parameters used in ECIES encryption scheme for both profiles A and B [20, pp. 209–210] [70, pp. 52–53].

Both sharedinfo1 and sharedinfo2 were incorrectly set to null in the implementation described in [69]. Additionally, in that same implementation, sharedinfo1 was mistakenly labeled as sharedinfo2. As a result, sharedinfo2 was fed into the key derivation step instead of sharedinfo1.

Table 2.2: Parameters used in ECIES encryption scheme for both profiles A and B.

step in ECIES encryption scheme	parameter name	parameter value for profile A	parameter value for profile B
Ephemeral key pair generation (at UE)	EC domain parameters	Curve25519	secp256r1
	point compression for ephemeral public key	N/A	true
Key Agreement	EC Diffie-Hellman primitive	X25519	EC Cofactor Diffie-Hellman (ECCDH)
Key Derivation	KDF	ANSI-X9.63-KDF	
	SharedInfo1	R: octet representation of Ephemeral public Key which is an EC point	
	Hash function	SHA-256	
Symmetric encryption scheme	Encryption algorithm	AES128 in CTR mode	
	enckeylen	16 octets (128 bits)	
	icblen	16 octets (128 bits)	
MAC	MAC function	HMACSHA-256	
	mackeylen	32 octets (256 bits)	
	maclen	8 octets (64 bits)	
	SharedInfo2	the empty string	

2.3.2.2 Scheme Input for ECIES encryption

If the SUPI type is IMSI, the scheme input for ECIES encryption is the MSIN part, which is coded as hexadecimal digits using packed BCD coding [20, p. 207]. Packed BCD puts two decimal digits in a byte. The order of digits in bytes is as shown in Figure 2.11 [20, p. 207] [18, p. 540]. For instance, the most significant byte consists of the two most significant digits, but in reverse order (first the second-most significant digit and then the most significant digit). As we saw in 2.3.1.1, the MSIN is a string of decimal digits with a maximum length of 10. If the MSIN consists of an odd number of decimal digits, the bits 5 to 8 of the final octet must be coded as '1111' (0xF), as specified in [20, p. 207]

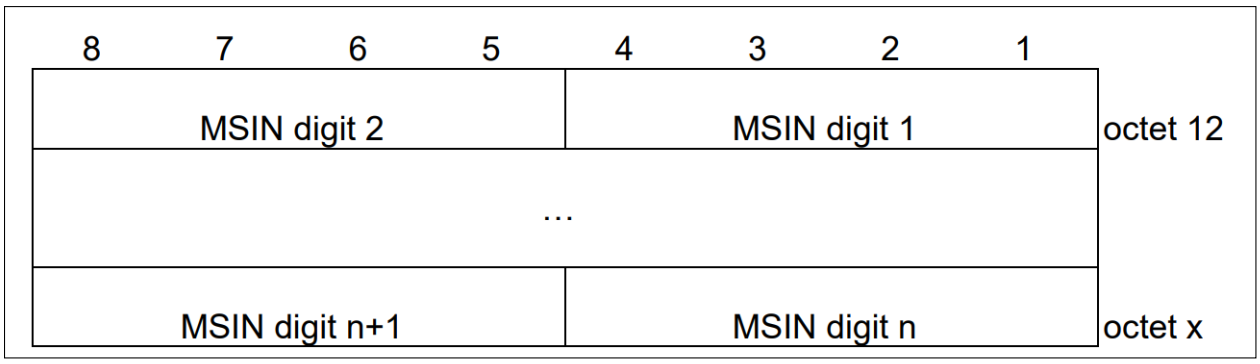


Figure 2.11: Scheme Output part of SUCI for SUPI type IMSI and null scheme [18, p. 540]

Figure 2.12 provides an example of converting decimal digits of the MSIN to hexadecimal digits for the scheme input. The order of digits per byte is based on Figure 2.11. Packed BCD is utilized for this conversion.

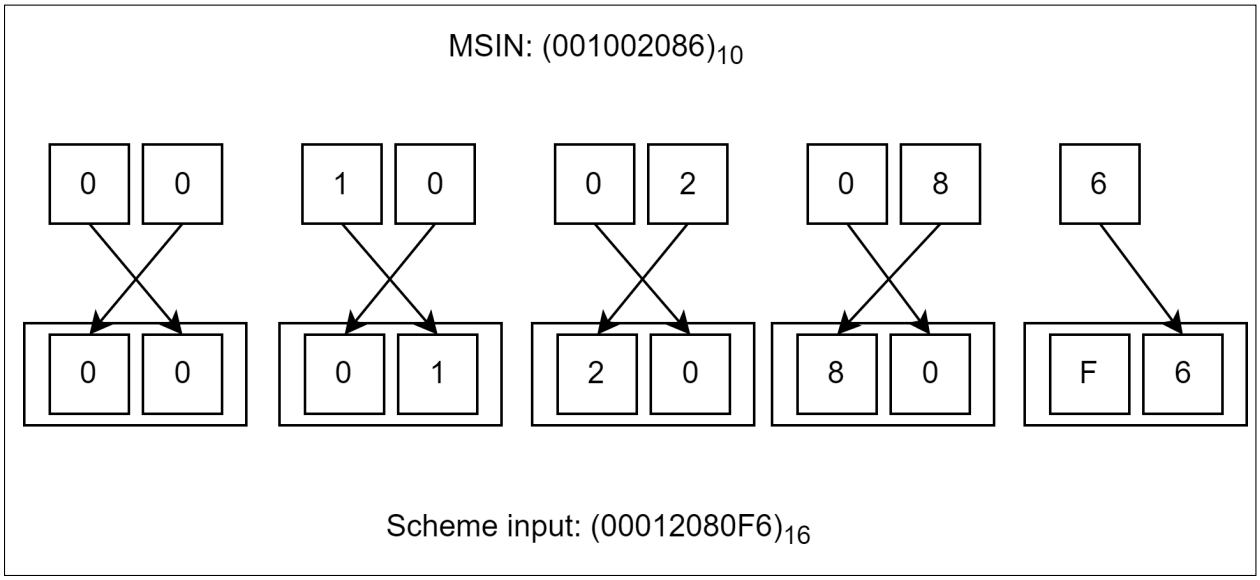


Figure 2.12: An example of converting decimal digits of MSIN to hex digits of Scheme Input

2.3.2.3 Step 1 in ECIES encryption: Key Pair Generation

2.3.2.3.1 Specification The key pair generation step of ECIES is based on section 3.2.1 of [70]. The only input for key pair generation is the domain parameter, and the output is a key pair corresponding to the input [70, p. 23]. Therefore, when the profile is A, the generated key pair is based on the domain parameter of Profile A, which is Curve25519 (see Table 2.2). Similarly, when the profile is B, the generated key pair is based on the domain parameter of Profile B, which is secp256rl (see Table 2.2).

2.3.2.3.2 Bugs in implementation of ANSI-X9.63-KDF Public and private Keys when Curve25519 is used (profile A), each should have a length of 32 bytes [29, p. 1]. Public and Private Keys when secp256r1 is used (profile A), should have lengths of 33 (in compressed form) and 32 bytes respectively [14]. However, in the implementation in

[69] their lengths are different from what we desire. [69] First compute key pair using the procedure in [69, p. 23] that returns a key pair of type java.security.KeyPair. Then [69] call getPublic() and getPrivate() on the object of type KeyPair to get the private key (of type java.security.PrivateKey) and public key (of type java.security.PublicKey). Later in [69] java.security.Key.getEncoded() method was used to get public and private keys in a byte array and these values were used whenever the raw value of keys was expected like the ephemeral public key that is part of a scheme output of ECIES encryption.

The reason for this mismatch between implementation in [69] that discussed above and specification is that in the implementation java.security.Key.getEncoded() method is used to obtain bytes of private and public keys, which do not return the raw key values. For public keys, getEncoded() returns the public keys in "X.509" format and for private keys, getEncoded() returns the private keys in "PKCS#8" format [67].

We need to extract raw values from java.security.PublicKey, which will be used, for instance, in constructing Scheme Output in ECIES encryption, and vice versa, which will be used, for instance, in the key agreement step in ECIES decryption at HN. For java.security.PrivateKey, we need to convert it to raw, as we just want to use this raw value (for printing and testing purposes).

2.3.2.3.3 Verification and refactoring For profile A in key pair generation step, the following was used in [69, pp. 23]:

- KeyPairGenerator keyPairGeneratorEphem = KeyPairGenerator.getInstance("X25519");

First we change it to its equivalence as suggested in [5] to make it like profile B implementation which make verification step simpler:

- KeyPairGenerator keyPairGeneratorEphem = KeyPairGenerator.getInstance("XDH");
- NamedParameterSpec paramSpec = new NamedParameterSpec("X25519");
- keyPairGeneratorEphem.initialize(paramSpec);

Based on [5], by setting algorithm to "XDH" for getting instance of KeyPairGenerator class and "X25519" as algorithm parameters name in the constructor of class java.security.spec- .NamedParameterSpec (as shown in the three lines above) we can then generate a "Curve25519" key pair. Similarly based on [28] by setting algorithm to "EC

for getting instance of KeyPairGenerator class and "secp256r1" as algorithm parameters name in constructor of class java.security.spec.ECGenParameterSpec, we can then generate a "secp256r1" key pair.

For verification of Key Pair Generation (step 1 of ECIES encryption in 2.9) first, we complete java.security.spec.AlgorithmParameterSpec.jml (see Listing 2.1), java.security.spec-.NamedParameterSpec.jml (see Listing 2.2), java.security.spec.ECGenParameterSpec (see Listing 2.3) and java.security.KeyPairGenerator.jml (see Listing 2.4, Listing 2.5, Listing 2.6 and Listing 2.7).

Figure 2.13 demonstrates the relation between interface AlgorithmParameterSpec and two classes NamedParameterSpec and ECGenParameterSpec in a Unified Modeling Language (UML) diagram [2] [12]. The model field curve_name is defined in AlgorithmParameterSpec (see Listing 2.1) for representing algorithm parameters' name in constructor of classes NamedParameterSpec and ECGenParameterSpec. Line 9 will be used in step 4 of ECIES (Symmetric Encryption).

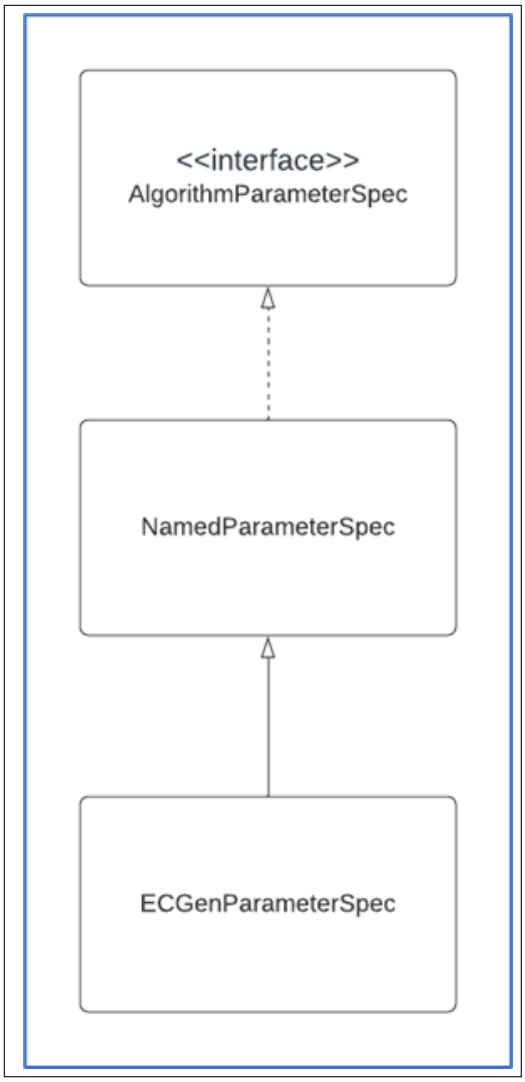


Figure 2.13: UML diagram showing relation between interface AlgorithmParameterSpec and two classes NamedParameterSpec and ECGenParameterSpec

Listing 2.1: complete java.security.spec.AlgorithmParameterSpec.jml: Adding one model field

```
1 public interface AlgorithmParameterSpec {
2     //@ model public instance int n;
3    //------------------------------ Author: Negar -----------------------
4     //@ model public instance String curve_name;
```

```
5
6       //@ pure
7       //@ model public byte[] get_Param();
8
9       //@ model public instance byte[] ivb;
10
11      //---------------------------------- end -----------------------------
12      // no members
13  }
```

In Listing 2.2 we create java.security.spec.NamedParameterSpec.jml and write the specification for its constructor (The constructor throws an exception of NullPointerException only when its input is null [12]). Similarly, in Listing 2.3 we add java.security.spec.ECGenParameterSpec.jml and specify the specification for the constructor [2].

Listing 2.2: create java.security.spec.NamedParameterSpec.jml

```
1  //------------------------------ Author: Negar ----------------------
2  public class NamedParameterSpec implements AlgorithmParameterSpec{
3
4     //@ public normal_behavior
5     //@ requires stdName != null;
6     //@ ensures curve_name==stdName;
7     //@ also public exceptional_behavior
8     //@ requires stdName == null;
9     //@ signals_only NullPointerException;
10    //@ pure
11    public NamedParameterSpec(java.lang.String stdName)throws
          NullPointerException;
12 }
13 //---------------------------------- end -----------------------------
```

Listing 2.3: create java.security.spec.ECGenParameterSpec.jml

```
1  //------------------------------ Author: Negar ----------------------
2  public class ECGenParameterSpec extends NamedParameterSpec{
3
4     //@ public normal_behavior
5     //@ requires stdName != null;
6     //@ ensures curve_name==stdName;
7     //@ also public exceptional_behavior
```

```
8     //@ requires stdName == null;
9     //@ signals_only NullPointerException;
10    //@ pure
11    public ECGenParameterSpec(java.lang.String stdName)throws
          NullPointerException;
12 }
13 //----------------------------------- end ------------------------------
```

For java.security.KeyPairGenerator.jml, first we define two model functions, isKeyPair_Curve25519 and isKeyPair_secp256r1, which serve as oracles for us (see Listing 2.4). We also define model field curve_selected that is responsible for curve name (algorithm parameters name). There is no need to add a model field like "algorithmSelected" since there is already a model field "algorithm" which is responsible for storing algorithm name. The generator in Listing 2.4 is for tracking the generator of public and private parts of a key pair (in step2 we will elaborate more on it).

Listing 2.4: complete java.security.KeyPairGenerator.jml: Adding one model field and two model methods

```
1     //@ ensures \result<==>(((kp.getPublic().generator) ==
          "Curve25519")&&((kp.getPrivate().generator) == "Curve25519"));
2     //@ pure
3     //@ model public static boolean isKeyPair_Curve25519(KeyPair kp );
4
5     //@ ensures \result<==>(((kp.getPublic().generator) ==
          "secp256r1")&&((kp.getPrivate().generator) == "secp256r1"));
6     //@ pure
7     //@ model public static boolean isKeyPair_secp256r1(KeyPair kp );
8
9     //@ model public instance java.lang.String curve_selected;
```

In java.security.KeyPairGenerator.jml The behavior of method KeyPairGenerator-.getInstance(java.lang.String algorithm) depends on whether algorithmIsSupported(algorithm) will be evaluated to true or not (see Listing 2.5). By the time of writing this document, the specification of the function algorithmIsSupported is not complete and it is not mentioned which algorithms are supported. Therefore, we have added a list of supported algorithms as presented in [15].

Listing 2.5: complete java.security.KeyPairGenerator.jml: getInstance(java.lang.String al-

gorithm)

```
1     //@ public normal_behavior
2     //------------------- Author: Negar ------------------------
3     //@ ensures
          (((algorithm=="DiffieHellman")||(algorithm=="DSA")||(algorithm=="RSA")
          ||(algorithm=="RSASSA-PSS")||(algorithm=="EC")||(algorithm=="XDH")
          ||(algorithm=="X25519")||(algorithm=="X448"))<==> \result);
4     //--------------------end Negar-------------------------
5     //@ pure heap_free
6     //@ model public static boolean algorithmIsSupported(String algorithm);
7
8
9     //@ public normal_behavior
10    //@  requires algorithmIsSupported(algorithm);
11    //@  ensures \fresh(\result);
12    //@  ensures \result.algorithm == algorithm;
13    //@ also public exceptional_behavior
14    //@  requires !algorithmIsSupported(algorithm);
15    //@  signals_only NoSuchAlgorithmException;
16    //@ pure
17    public static KeyPairGenerator getInstance(java.lang.String algorithm)
          throws NoSuchAlgorithmException;
```

In a similar fashion, as Listing 2.6 shows, KeyPairGenerator.initialize(java.security.spec-.AlgorithmParameterSpec alg) depends on validParameterSpec(alg, alg.n). Similar to the function algorithmIsSupported, we need to complete specification for validParameterSpec [12] [2] (see Listing 2.6). For setting model field curve_selected we add lines 16 and 17 in Listing 2.6.

Listing 2.6: complete java.security.KeyPairGenerator.jml: initialize(AlgorithmParameterSpec alg)

```
1     //@ public normal_behavior
2     //@  ensures \result ==> alg.n == n;
3     //--------------Author: Nega-----------------------------
4     //@ ensures ((alg instanceof
          java.security.spec.NamedParameterSpec)&&(alg.curve_name ==
          "X25519"))==> (\result ==true );
5     //@ ensures ((alg instanceof
          java.security.spec.ECGenParameterSpec)&&(alg.curve_name ==
```

```
        "secp256r1"))==> (\result ==true );
6       //--------------end Negar------------------------------
7
8
9       //@ public normal_behavior
10      //@   requires validParameterSpec(alg, alg.n);
11      //@   assignable this.*;
12      //@   ensures this.algorithm == \old(this.algorithm);
13      //@   ensures this.n == alg.n;
14      //-------------- Author: Negar------------------------------
15      //@ ensures ((alg instanceof
            java.security.spec.NamedParameterSpec)&&(alg.curve_name ==
            "X25519"))==> (curve_selected == "X25519" );
16      //@ ensures ((alg instanceof
            java.security.spec.ECGenParameterSpec)&&(alg.curve_name ==
            "secp256r1"))==> (curve_selected == "secp256r1" );
17      //--------------end Negar------------------------------
18      //@ also public exceptional_behavior
19      //@   requires !validParameterSpec(alg, alg.n);
20      //@   signals_only InvalidAlgorithmParameterException;
21      public void initialize(java.security.spec.AlgorithmParameterSpec alg)
            throws InvalidAlgorithmParameterException;
```

Finally, we add two lines to generateKeyPair: see Listing 2.7 lines 6 and 7. These lines use the oracle functions isKeyPair_Curve25519 and isKeyPair_secp256r1.

Listing 2.7: complete java.security.KeyPairGenerator.jml: generateKeyPair()

```
1       //@ also public normal_behavior
2       //@   ensures \fresh(\result);
3       //@   ensures \result.getPublic() instanceof DHPublicKey ==> 0 <=
            ((DHPublicKey)\result.getPublic()).getY().value <
            java.math.BigInteger.pow256(this.n);
4       //@   ensures \result.getPublic().getEncoded() != null;
5       //--------------Author: Negar------------------------------
6       //@ ensures ((algorithm == "XDH") &&(curve_selected == "X25519"))==>
            isKeyPair_Curve25519(\result );
7       //@ ensures ((algorithm == "EC") &&(curve_selected == "secp256r1"))==>
            isKeyPair_secp256r1(\result );
8       //--------------end Negar------------------------------
9       //@ pure
```

```
10      public KeyPair generateKeyPair();
```

Listing 2.8 demonstrates implementation and verification of Step1 of ECIES encryption, Key Pair Generation.

Listing 2.8: implementation and verification of Step1 of ECIES encryption: Key Pair Generation

```
1       //Never throws exception as we don't have requires statement for
            normal_behavior
2     //@ private normal_behavior
3     //@ ensures (KeyPairGenerator.isKeyPair_Curve25519(\result));
4     //@ ensures \result !=null; //even if we remove this line, by defualt return
        value of a function is != null
5     //@ pure
6     private static KeyPair generateEphemeralKeyPairProfileA() {
7        try {
8           KeyPairGenerator keyPairGeneratorEphem =
                KeyPairGenerator.getInstance("XDH");
9           NamedParameterSpec paramSpec = new NamedParameterSpec("X25519");
10          keyPairGeneratorEphem.initialize(paramSpec);
11          KeyPair keyPairEphem = keyPairGeneratorEphem.generateKeyPair();
12          return keyPairEphem;
13       }
14
15       catch (NoSuchAlgorithmException | InvalidAlgorithmParameterException e) {
16          return null;
17       }
18    }
19
20    //-------------------------------------------------------
21    //Never throws exception as we don't have requires statement for
        normal_behavior
22    //@ private normal_behavior
23    //@ ensures (KeyPairGenerator.isKeyPair_secp256r1(\result));
24    //@ ensures \result !=null; //even if we remove this line, by defualt return
        value of a function is != null
25    //@ pure
26    private static KeyPair generateEphemeralKeyPairProfileB() {
27       try {
28          KeyPairGenerator keyPairGeneratorEphem =
```

```
            KeyPairGenerator.getInstance("EC");
29          ECGenParameterSpec ECDomainParamSecp256r1 = new
                ECGenParameterSpec("secp256r1"); //
                https://docs.oracle.com/en/java/javase/14/docs/specs/security/standard-names.html
30          keyPairGeneratorEphem.initialize(ECDomainParamSecp256r1);
31          KeyPair keyPairEphem = keyPairGeneratorEphem.generateKeyPair();
32          return keyPairEphem;
33       } catch (NoSuchAlgorithmException | InvalidAlgorithmParameterException e)
             {
34          return null;
35       }
36    }
37
38    //-------------------------------------------------------
39       //@ public normal_behavior
40    //@ requires (profileSelectionStr == "A") || (profileSelectionStr == "B");
41    //@ ensures (profileSelectionStr == "A") ==>
          (KeyPairGenerator.isKeyPair_Curve25519(\result));
42    //@ ensures (profileSelectionStr == "B") ==>
          (KeyPairGenerator.isKeyPair_secp256r1(\result));
43    //@ pure
44    public static KeyPair generateEphemeralKeyPair(String profileSelectionStr) {
45
46       if (profileSelectionStr == "A") {
47          return generateEphemeralKeyPairProfileA();
48       }
49
50       else if (profileSelectionStr == "B") {
51          return generateEphemeralKeyPairProfileB();
52       }
53       return null;
54    }
```

2.3.2.4 Step 2 in ECIES encryption: Key Agreement

2.3.2.4.1 Specification Key Agreement step of ECIES is stated in section 3.3 of [70]. The Key Agreement for profile A uses the X25519 EC Diffie-Hellman primitive, while the Key Agreement for profile B uses the ECCDH EC Diffie-Hellman primitive (see Table 2.2).

2.3.2.4.2 Verification and refactoring For verification of Key Agreement (step 1 of ECIES encryption in Figure 2.9) first we complete java.security.Key.jml, java.security-.KeyPairGenerator.jml (see Listing 2.4), javax.crypto.KeyAgreement.jml (see Listing 2.9, Listing 2.10, Listing 2.13, Listing 2.14 and Listing 2.15), java.security.Provider.jml (see Listing 2.11) and org.bouncycastle.jce.provider.BouncyCastleProvider.jml (see Listing 2.12).

So far in Step 1 (key pair generation), we have determined that in profile A, for generated key pair kpa, (KeyPairGenerator.isKeyPair_Curve25519(kpa)) is true and similarly in profile B for key pair kpb, KeyPairGenerator.isKeyPair_secp256r1(kpb) is true. Now we need to also know this property is held for public and private keys that are extracted from key pair. For that, we add the model field generator in java.security.Key.jml (//@ model public instance java.lang.String generator;). As java.security.PublicKey and java.security.PrivateKey are subinterfaces of java.security.Key [8] by putting this model field in Key, PrivateKey and PublicKey will also have this field.

Listing 2.4 shows in lines 1 and 5 that the generator field is set corresponding to the return value of the functions KeyPairGenerator.isKeyPair_Curve25519 and isKeyPair_secp256r1 in java.security.KeyPairGenerator.jml. More specifically line 1 says, isKeyPair_Curve25519-(keyPair kp) is true if and only if the private and public key fields of kp have generator equals to Curve25519. Similarly, line 5 says isKeyPair_secp256r1 (keyPair kp) is true if and only if the private and public key fields of kp have generator equals to secp256r1.

By looking at javax.crypto.KeyAgreement.jml we see that the behavior of the function getInstance(String s) depends on the model method validAlgorithm(String algorithm). First we need to tell the prover that which algorithms are valid for input of getInstance(String s). In Listing 2.9 we provide valid options [9] [16]. in line 18 of Listing 2.9 we say that the field algorithm needs to be set to the value of the input of getInstance(String s).

Listing 2.9: complete javax.crypto.KeyAgreement.jml: getInstance(String s)

```
1  //------------------------------ Author: Negar ----------------------
2  //@ spec_public
3  //---------------------------------- end -----------------------------
4   private final java.lang.String algorithm;
5
6
7   //@ public normal_behavior
8   //------------------------------ Author: Negar ----------------------
9   //@ ensures (((algorithm == "DiffieHellman")||(algorithm ==
        "ECDH")||(algorithm == "ECMQV")||(algorithm == "XDH")||(algorithm ==
        "X25519")||(algorithm == "X448")) <==> \result);
```

```
10      //---------------------------------- end ----------------------------
11      //@ model public static final pure heap_free boolean
            validAlgorithm(java.lang.String algorithm);
12
13
14      //@ public normal_behavior
15      //@  requires validAlgorithm(s);
16      //@  ensures \fresh(\result);
17      //------------------------------ Author: Negar ----------------------
18      //@  ensures \result.algorithm == s;
19      //---------------------------------- end ----------------------------
20      //@ also public exceptional_behavior
21      //@  requires !validAlgorithm(s);
22      //@  signals_only java.security.NoSuchAlgorithmException;
23      //@ pure
24      public static final KeyAgreement getInstance(java.lang.String s) throws
            java.security.NoSuchAlgorithmException;
```

In the implementation we want to call javax.crypto.KeyAgreement.getInstance(java.lang.String s, java.security.Provider provider) function using bouncy Castle as a provider and algorithm ECCDH so we need to enhance the specification to tell the prover that getInstance will have normal behavior once calling with these inputs. Like getInstance(java.lang.String s) that its behavior depends on the model method "validAlgorithm" here we need to evaluate a pair of an algorithm and a provider to see whether the algorithm is supported by the provider (see Listing 2.10).

We want to give the responsibility for checking whether the algorithm is supported by the provider, to the provider side. For that, we define the model method isValidAlgorithm in java.security.Provider.jml (see Listing 2.11 [42]). We will also let the prover know isValidAlgorithm("ECCDH") will return true for BouncyCastleProvider (see Listing 2.12 [1])

Listing 2.10: complete javax.crypto.KeyAgreement.jml: getInstance (java.lang.String s, java.security.Provider provider)

```
1       //------------------------------ Author: Negar ----------------------
2       //@ public normal_behavior
3       //@ requires provider.isValidAlgorithm(s);
4       //@ ensures \fresh(\result);
5       //@ ensures \result.algorithm == s;
6       //@ pure
```

```
7     //---------------------------------- end ----------------------------
8      public static final KeyAgreement getInstance(java.lang.String s,
           java.security.Provider provider) throws
           java.security.NoSuchAlgorithmException;
```

Listing 2.11: create java.security.Provider.jml

```
1  //------------------------------ Author: Negar ----------------------
2
3  public abstract class Provider extends java.util.Properties
4  {
5  //@ pure
6  //@ model public boolean isValidAlgorithm(java.lang.String s);
7
8  protected  Provider (String name,String versionStr,String info);
9  }
10 //------------------------------ end ----------------------
```

Listing 2.12: create org.bouncycastle.jce.provider.BouncyCastleProvider.jml

```
1  //------------------------------ Author: Negar ----------------------
2  public final class BouncyCastleProvider
3  extends java.security.Provider{
4
5  //@ ensures this.isValidAlgorithm("ECCDH");
6  //@ public normal_behavior
7  //@ pure
8  public BouncyCastleProvider();
9
10 //------------------------------ end ----------------------
```

The function init(java.security.Key key) in javax.crypto.KeyAgreement initializes key agreement with private information (private key) and in case of given input that is not appropriate for this key agreement it will throw InvalidKeyException [10]. Thus we need to add a requires statement in the normal behavior (see Listing 2.13) as it will not execute without exception for all inputs. We also add @ensures (privateKeyIsSet) to specify that this step is done. Lines 1 to 20 in Listing 2.13 shows some model fields and methods that will be used later.

Listing 2.13: complete javax.crypto.KeyAgreement.jml: init(java.security.Key key)

```
1     //------------------------------ Author: Negar -----------------------
2
3      //@ model public instance boolean privateKeyIsSet;
4      //@ model public instance java.security.PrivateKey priKey;
5      //@ model public instance java.security.PublicKey pubKey;
6
7      //@ pure
8      //@ model public static boolean isKeyAgreement_X25519(KeyAgreement ka );
9
10     //@ pure
11     //@ model public static boolean isKeyAgreement_ECCDH(KeyAgreement ka );
12
13
14    //@ pure
15     //@ model public static boolean isKey_X25519(byte[] k );
16
17     //@ pure
18     //@ model public static boolean isKey_ECCDH(byte[] k );
19
20     //---------------------------------- end ----------------------------
21
22       //@ public normal_behavior
23    //------------------------------ Author: Negar -----------------------
24     //in our case we want to add below line specification to make sure "key"
          is a appropriate one
25     //@ requires((algorithm == "X25519")==>(key.generator == "Curve25519"));
26    //@ requires((algorithm == "ECCDH")==>(key.generator == "secp256r1"));
27     //@ ensures (privateKeyIsSet);
28     //@ ensures priKey == key;
29     //---------------------------------- end ----------------------------
30     //@   assignable this.*;
31     //@   ensures true; // FIXME - ignoring exception
32     public final void init(java.security.Key key) throws
          java.security.InvalidKeyException;
```

The function doPhase(java.security.Key key, boolean b) in javax.crypto.KeyAgreement will throw exceptions in two cases. It will throw InvalidKeyException if it receives an inappropriate key and will throw IllegalStateException in case the key agreement was not

initialized previously [11]. So, for normal behavior we need to consider these two cases. In Listing 2.14, lines 3 and 4 are added for the first case and line 5 is added for the second case.

Listing 2.14: complete javax.crypto.KeyAgreement.jml: doPhase(java.security.Key key, boolean b)

```
1       //@ public normal_behavior
2       //------------------------------ Author: Negar -----------------------
3       //@ requires((algorithm == "X25519")==>(key.generator == "Curve25519"));
4       //@ requires((algorithm == "ECCDH")==>(key.generator == "secp256r1"));
5       //@ requires privateKeyIsSet;
6       //@ ensures ((algorithm == "X25519")==>(isKeyAgreement_X25519(this)));
7       //@ ensures ((algorithm == "ECCDH")==>(isKeyAgreement_ECCDH(this)));
8       //@ ensures pubKey == key;
9       //@ ensures priKey == \old(priKey);
10      //---------------------------------- end -----------------------------
11      //@  assignable this.*;
12      //@  ensures \fresh(\result); // FIXME - ignoring exception
13      public final java.security.Key doPhase(java.security.Key key, boolean b)
            throws java.security.InvalidKeyException,
            java.lang.IllegalStateException;
```

Finally we complete javax.crypto.KeyAgreement.generateSecret() in Listing 2.15.

Listing 2.15: complete javax.crypto.KeyAgreement.jml: generateSecret()

```
1       //@ public normal_behavior
2       //------------------------------ Author: Negar -----------------------
3       //@ ensures (isKeyAgreement_X25519(this))==> (isKey_X25519(\result));
4       //@ ensures (isKeyAgreement_ECCDH(this))==> (isKey_ECCDH(\result));
5       //@ ensures (isKey_X25519(\result)||isKey_ECCDH(\result))==>
            (\result.length == 32);
6       //---------------------------------- end -----------------------------
7       //@  ensures \fresh(\result);
8       //@  ensures \result.length == _secret.length;
9       //@ pure
10      public final byte[] generateSecret() throws
            java.lang.IllegalStateException;
```

Listing 2.16 demonstrates implementation and verification of Step 2 of ECIES encryp-

tion, Key Agreement.

Listing 2.16: implementation and verification of Step 2 of ECIES encryption: Key Agreement

```
1   //@ private normal_behavior
2   //@ requires (( privKeyVal.generator == "Curve25519" )&&(
        pubKeyVal.generator == "Curve25519" ));
3   //@ ensures KeyAgreement.isKeyAgreement_X25519(\result);
4   //@ ensures (\result).priKey == privKeyVal;
5   //@ ensures (\result).pubKey == pubKeyVal;
6   //@ pure
7   private static KeyAgreement keyAgreementFnProfileA(PrivateKey privKeyVal,
        PublicKey pubKeyVal) {
8      try {
9         KeyAgreement profileAKeyAgreement = KeyAgreement.getInstance("X25519");
10        profileAKeyAgreement.init(privKeyVal); // presumably uses
              eph_privateKey for Profile A
11        profileAKeyAgreement.doPhase(pubKeyVal, true); // presumably uses
              HN_publicKey for Profile A
12        //@ assert KeyAgreement.isKeyAgreement_X25519(profileAKeyAgreement);
13        return profileAKeyAgreement;
14     } catch (NoSuchAlgorithmException | InvalidKeyException e) {
15        return null;
16     }
17  }
18  //-------------------------------------------------------------
19  //@ private normal_behavior
20  //@ requires (( privKeyVal.generator == "secp256r1" )&&( pubKeyVal.generator
        == "secp256r1" ));
21  //@ ensures KeyAgreement.isKeyAgreement_ECCDH(\result);
22  //@ ensures (\result).priKey == privKeyVal;
23  //@ ensures (\result).pubKey == pubKeyVal;
24  //@ pure
25
26  private static KeyAgreement keyAgreementFnProfileB(PrivateKey privKeyVal,
        PublicKey pubKeyVal) {
27     try {
28        Provider BC = new BouncyCastleProvider();
29        KeyAgreement profileBKeyAgreement = KeyAgreement.getInstance("ECCDH",
              BC);
```

```
30
31          profileBKeyAgreement.init(privKeyVal);
32          profileBKeyAgreement.doPhase(pubKeyVal, true);
33
34          return profileBKeyAgreement;
35       } catch (NoSuchAlgorithmException | InvalidKeyException e) {
36          return null;
37       }
38    }
39
40 //-------------------------------------------------------------------
41    //@ public normal_behavior
42    //@ requires (profile== "A") || (profile == "B");
43    //@ requires (profile == "A") ==>(( privKey.generator == "Curve25519" )&&(
          pubKey.generator == "Curve25519" ));
44    //@ requires (profile == "B")==>(( privKey.generator == "secp256r1" )&&(
          pubKey.generator == "secp256r1" ));
45
46    //@ ensures (profile == "A") ==> (KeyAgreement.isKey_X25519(\result));
47    //@ ensures (profile == "B") ==> (KeyAgreement.isKey_ECCDH(\result));
48    //@ ensures \result.length == 32;
49    //@ pure
50    public static byte[] keyAgreement(String profile, PrivateKey privKey,
          PublicKey pubKey) {
51       KeyAgreement keyAgreement = (profile == "A") ?
             keyAgreementFnProfileA(privKey, pubKey)
52             : keyAgreementFnProfileB(privKey, pubKey);
53
54       try {
55
56          byte[] key = keyAgreement.generateSecret();
57
58          return key;
59
60       } catch (IllegalStateException e) {
61          return null;
62
63       }
64
65    }
```

2.3.2.5 Step 3 in ECIES encryption: Key Derivation

2.3.2.5.1 Specification Key Derivation algorithm for both profiles A and B as Table 2.2 shows, is ANSI-X9.63-KDF.

Inputs and output of ANSI-X9.63-KDF are [70, pp. 32] (see Figure 2.14):

- inputs
  1. Z: a shared secret value represented in octet string
  2. $keydatalen$: length in octets of the output K
  3. (Optional) $SharedInfo$: a shared value represented in octet string
- output
  - K: Key data represented in octet string of length $keydatalen$ octets, or invalid

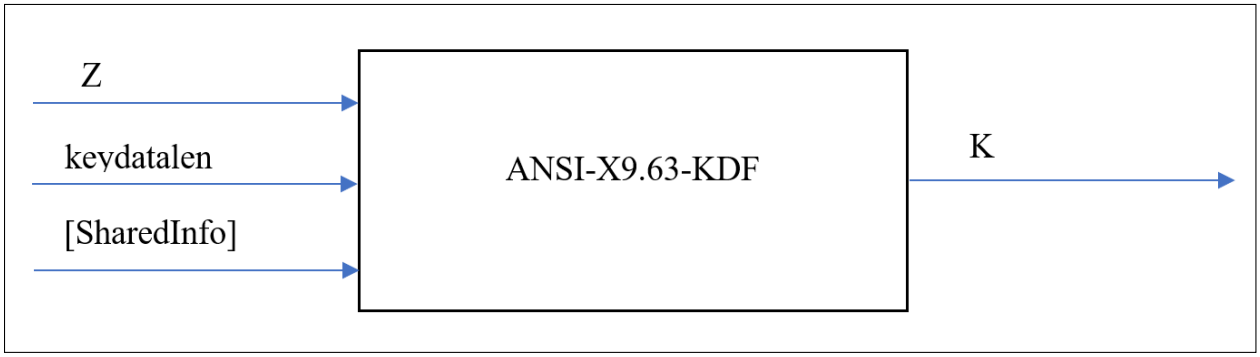


Figure 2.14: ANSI-X9.63-KDF inputs and output

Inputs and output of ANSI-X9.63-KDF that is used for ECIES encryption scheme in [20, pp. 207] are (see Figure 2.15, Figure 2.9 and Table 2.2):

- inputs
  1. Z: Ephemeral Shared Key
  2. $keydatalen$: $(enckeylen + icblen + mackeylen) = 16 + 16 + 32 = 64$ bytes
  3. (Optional) $SharedInfo$: SharedInfo1

- output
  - K: (Ephemeral Encryption Key || ICB || Ephemeral MAC Key) with length 64 bytes

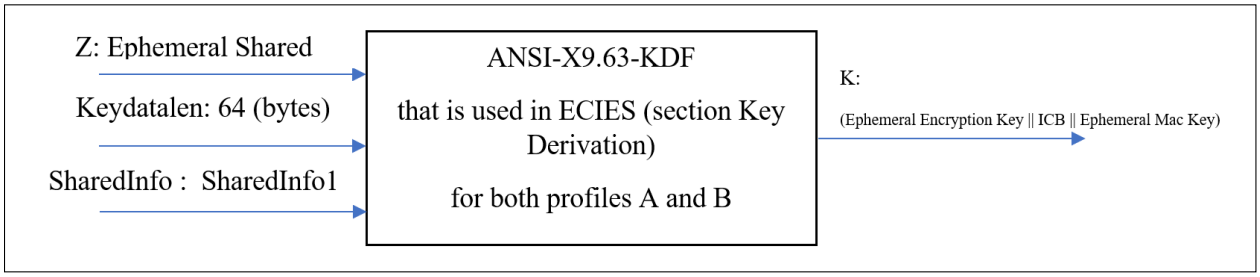


Figure 2.15: ANSI-X9.63-KDF inputs and output that is used for ECIES encryption scheme in [20, pp. 207]

In ANSI-X9.63-KDF $Hash$ is one of the [70, pp. 30]:

- SHA-1
- SHA-224
- SHA-256
- SHA-384
- SHA-512

The $hashmaxlen$ parameter specifies the maximum length of an input message for the $Hash$ function in octets. The $hashlen$ parameter, on the other hand, represents the length of the output of the $Hash$ function in octets.

ANSI-X9.63-KDF algorithm is as below [70, pp. 32–33].

1. If $|Z| + |SahredInfo| + 4 \geq hashmaxlen$ then return "invalid".
2. If $Keydatalen \geq hashlen \times (2^{23} - 1)$ then return "invalid".
3. Set $Counter$ to 4 octet value 0x00000001

4. For $i = 1$ to $\lceil \frac{Keydatalen}{hashlen} \rceil$
   (a) $K_i = Hash\,(Z||Counter||SharedInfo)$
   (b) $Counter = Counter + 1$
   (c) $i = i + 1$
5. K = left most $keydatalen$ of $(K_1||K_2||K_{\lceil \frac{Keydatalen}{hashlen} \rceil})$
6. return K

2.3.2.5.2 Bugs in implementation of ANSI-X9.63-KDF There are few bugs in implementation of ANSI-X9.63-KDF algorithm in [69, pp. 27] used for ECIES encryption scheme for both profiles A and B (see Listing 2.17).

- In line 8 sharedInfoValLen is set to 0 instead of sharedInfoVal's length.
- In line 9, instead of 4 there shoud be (4×8) as other length values in this line are capture number of bits instead of bytes.
- *Counter* doesn't match with specification (see lines 17-26): $Integer.toHexString(1).getBytes(StandardCharsets.UTF-8)$ equals to byte array "[49]" so it is not 0x00000001 (byte array [0,0,0,1]).
- converting byte [] to String and then vice versa (for instance see lines 1,22 and 31).

Listing 2.17: Implementation of ANSI-X9.63-KDF algorithm from [69, pp. 27] (used for ECIES encryption scheme for both profiles A and B)

```
1  public static byte[] keyDerivationFn(byte[] kdfEphSharedSecretKeyVal, byte[]
       sharedInfoVal) {
2
3        int keyDataLen = 128 + 128 + 256;
4        double hashMaxLen = ((Math.pow(2, 64)) - 1) / 8;
5        int hashLen = 256;
6
7        int kdfEphSharedSecretKeyValLen = 8 * kdfEphSharedSecretKeyVal.length;
8        int sharedInfoValLen = 0;
```

```
9       boolean step1 = kdfEphSharedSecretKeyValLen + sharedInfoValLen + 4 <
            (hashMaxLen * 8);
10      boolean step2 = keyDataLen < hashLen * ((Math.pow(2, 32)) - 1);
11      if (step1 && step2) {
12
13         double ceilRatio = Math.ceil(keyDataLen / hashLen);
14
15         ByteArrayOutputStream outputK = new ByteArrayOutputStream();
16
17         for (int i = 1; i <= ceilRatio; i++) {
18
19            String counterHexStrVal = Integer.toHexString(i);
20
21            String concatStrZCounterSharedInfo1 = "";
22            String ZStr = new String(kdfEphSharedSecretKeyVal,
                  StandardCharsets.UTF_8);
23
24            String SharedInfoStr = new String(sharedInfoVal,
                  StandardCharsets.UTF_8);
25
26            concatStrZCounterSharedInfo1 = ZStr + counterHexStrVal +
                  SharedInfoStr;
27
28            try {
29
30               MessageDigest sha256_md = MessageDigest.getInstance("SHA-256");
31               byte[] K_i = sha256_md.digest(concatStrZCounterSharedInfo1
                     .getBytes(StandardCharsets.UTF_8));
32
33               outputK.write(K_i);
34
35            } catch (NoSuchAlgorithmException | IOException e) {
36               e.printStackTrace();
37            }
38
39         }
40
41         byte[] output_K_FINAL = outputK.toByteArray();
42
43         String output_K_FINAL_str = new String(output_K_FINAL,
```

```
            StandardCharsets.UTF_8);
44          System.out.println("keyDerivationFn: output_K_FINAL_str... " +
                output_K_FINAL_str);
45
46          return output_K_FINAL;
47
48      }
49
50      else {
51          System.err.println("INVALID");
52          return null;
53      }
54
55  }
```

2.3.2.5.3 Verification and refactoring We can rewrite ANSI-X9.63-KDF algorithm used in ECIES encryption scheme for both profiles A and B as below [70, pp. 32–33]. $Hash$ for both profiles A and B is SHA-256 (see 2.2), so $hashlen$ is 32 bytes and $hashmaxlen$ is $(2^{64} - 1)$ bits. $keydatalen$ is 64 bytes.

1. check $|EphemeralSharedKey| * 8 + |SahredInfo1| * 8 + 4 * 8 \geq (2^{64} - 1)$ which is always false once putting SahredInfo1 and Ephemeral Shared Key in Java byte[] as maximum length of an array in Java is $2^{31} - 1$.
2. check $64 \geq 32 \times (2^{23} - 1)$, which is always false.
3. Compute K_1 and K_2
   - $K_1 = SHA - 256\,(phemeralSharedKey \,||\, 0x00000001 \,||\, SharedInfo1)$
   - $K_2 = SHA - 256\,(phemeralSharedKey \,||\, 0x00000002 \,||\, SharedInfo1)$
4. return $k_1 \| k_2$

In other words, the inputs of Key Derivation (step 3 of ECIES encryption in 2.9) are $EphemeralSharedKey$, $SharedInfo1$ and the output has 64 bytes length. The left half of the output is $SHA - 256\,(phemeralSharedKey \,||\, 0x00000001 \,||\, SharedInfo1)$ and the right half of the output is $SHA-256\,(phemeralSharedKey \,||\, 0x00000002 \,||\, SharedInfo1)$.

Listing 2.18 demonstrates that with static verification we can show steps 1 and 2 of ANSI-X9.63-KDF algorithm used in key derivation step in ECIES encryption scheme, are always passed.

Listing 2.18: steps 1 and 2 of ANSI-X9.63-KDF algorithm used in key derivation step in ECIES encryption scheme

```
1   //@ private normal_behavior
2   //@ ensures \result == true;
3   //@ pure
4   private static boolean keyDerivationFnStepsOneToTwo(int a1_byteLength, int
        a2_byteLength) {
5      int keyData_BitLen = 128 + 128 + 256; // length of the output of
           keyDerivationFn function
6      double hashMax_BitLen = 1.8446744073709552E19;// ((Math.pow(2, 64)) - 1) ;
7
8        int hash_BitLen = 256; // in bits
9
10     long kdfEphSharedSecretKeyVal_BitLen = Long.valueOf(8) * a1_byteLength;
11     long sharedInfoVal_BitLen = Long.valueOf(8) * a2_byteLength;
12
13     // step 1:
14     boolean step1 = ((((kdfEphSharedSecretKeyVal_BitLen) +
           (sharedInfoVal_BitLen)) + (4 * 8)) < (hashMax_BitLen));
15     // step 2:
16     boolean step2 = keyData_BitLen < hash_BitLen * (4.294967295E9);
17
18     assert (step1) : "Step 1 doesn't hold in keyDerivationFn method";
19     assert (step2) : "Step 2 doesn't hold in keyDerivationFn method";
20     return (step1 && step2);
21  }
```

For verification of Key Derivation (Step 3 of ECIES encryption in 2.9) first we complete java.security.MessageDigest.jml (see Listing 2.19, Listing 2.20 and Listing 2.21).

The model field SHA256_selected in Listing 2.19 is a flag that become true once the algorithm is SHA256. The model method isHash256 in Listing 2.19 is a method that return true once its second argument is SHA-256 of the first argument. The model method validAlgorithm in Listing 2.19 is defined to tell the verifier "SHA-256" is a valid algorithm for java.security.MessageDigest.getInstance(java.lang.String s) i.e. java.security.MessageDigest.getInstance("SHA-256") does not throw NoSuchAlgorithmException [4][41].

Listing 2.19: complete java.security.MessageDigest.jml: Adding one model field and two model methods

```
1  //@ model public static boolean SHA256_selected;
2
3
4    //@ ensures (isHash256( input, output ))==>(\forall byte []inputCopy;
       ((inputCopy!=null)&&(inputCopy.length == input.length)&&(\forall int
       index; 0<=index< inputCopy.length;
       inputCopy[index]==\old(input)[index])); (\forall byte []outputCopy;
       ((outputCopy!=null)&&(outputCopy.length == output.length)&&(\forall int
       index2; 0<=index2< outputCopy.length;
       outputCopy[index2]==\old(output)[index2]));
       isHash256(inputCopy,outputCopy)) );
5
6    //below line is needed as we expect determinism: computing sha-256 (input,
       key) always get us same result
7    //@ ensures (\forall byte[] outputCopy; isHash256(input, outputCopy ) &&
       isHash256(input, output );java.util.Arrays.equals(outputCopy,output));
8
9    //@ assignable \nothing;
10   //@ pure
11   //@ model public static boolean isHash256(byte[] input, byte[] output );
12
13
14   //@ public normal_behavior
15   //@ ensures ((algorithm =="SHA-256") ==> \result);
16   //@ pure
17   //@ model public static boolean validAlgorithm(String algorithm);
```

In lines 2 to 6 of Listing 2.20 we indicate that for expecting termination of getInstance (java.lang.String s) the input should be a valid algorithm and not equal to null. Also we say that once the input algorithm is "SHA-256" the length field of the returned object is equal to SHA256_SIZE ,which is a model field defined in MessageDigest.jml (model public static final int SHA256_SIZE = 32), and the flag SHA256_selected becomes true.

Listing 2.20: complete java.security.MessageDigest.jml: getInstance(java.lang.String s)

```
1  //@ public normal_behavior
2    //------------------------------ Author: Negar -----------------------
3    //@ requires validAlgorithm(s); //make sure that it doesn't throw
```

```
    NoSuchAlgorithmException
4   //@ requires (s !=null); //make sure that it doesn't throw
    NullPointerException
5   //@ ensures s.equals("SHA-256")==>(\result.length == SHA256_SIZE);
6   //@ ensures s.equals("SHA-256")==>(SHA256_selected == true);
7   //-------------------------------------------------------------------
8   //@  ensures \fresh(\result);
9   //@ pure
10  public static MessageDigest getInstance(java.lang.String s) throws
    NoSuchAlgorithmException;
```

In line 2 of Listing 2.21 we indicate that once algorithm is SHA-256, the output of the digest method is SHA-256 of its input.

Listing 2.21: complete java.security.MessageDigest.jml: digest(byte[] bb)

```
1   //------------------------------ Author: Negar ----------------------
2   //@ ensures (SHA256_selected) ==> isHash256(\old(bb),\result);
3   //---------------------------------- end -----------------------------
4   public byte[] digest(byte[] bb);
```

Listing 2.22 demonstrates implementation and verification of Step 3 of ECIES encryption: Key Derivation.

Listing 2.22: implementation and verification of Step3 of ECIES encryption: Key Derivation

```
1   // (hash(a1||oneIn4Bytes||a2) || hash(a1||twoIn4Bytes||a2))
2
3   //@ public normal_behavior
4   //@ requires (a1 != null)&&(a2 != null);
5   //@ requires (a1.length +4 + a2.length)<= Integer.MAX_VALUE ;
6   //@ ensures \result.length == 64;
7   //@ ensures MessageDigest.isHash256( \old(Concatination.concat2byteArray(
        a1,
        Concatination.concat2byteArray(IntToByteArrayConverter.returnOneIn4Bytes(),
        a2))),java.util.Arrays.copyOfRange(\result,0,32));
8   //@ ensures MessageDigest.isHash256( \old(Concatination.concat2byteArray(
        a1,
        Concatination.concat2byteArray(IntToByteArrayConverter.returnTwoIn4Bytes(),
        a2))),java.util.Arrays.copyOfRange(\result,32,64));
```

```
9     //@ pure
10    public static byte[] keyDerivationFn(byte[] a1, byte[] a2) {
11       // a1 ==z
12       // a2 ==sharedInfo
13       // ---------------------------------step 1 & 2
14       boolean step1_2 = keyDerivationFnStepsOneToTwo(a1.length, a2.length);
15       //@ assert step1_2;
16       // ---------------------------------step 3
17       // use returnOneIn4Bytes() and returnTwoIn4Bytes()
18       // ---------------------------------step 4 & 5
19       final byte[] K_1 =
             Cryptography.computeSha256(Concatination.concat2byteArray(a1,
20             Concatination.concat2byteArray(IntToByteArrayConverter.returnOneIn4Bytes(),
                  a2)));
21       final byte[] K_2 =
             Cryptography.computeSha256(Concatination.concat2byteArray(a1,
22             Concatination.concat2byteArray(IntToByteArrayConverter.returnTwoIn4Bytes(),
                  a2)));
23
24       // concat K_1 and K_2
25       byte[] output_K_FINAL = Concatination.concat2byteArray(K_1, K_2);
26       return output_K_FINAL;
27    }
28    //-------------------------------------------------
29    //@ public normal_behavior
30    //@ requires bb!=null;
31    //@ ensures \result.length == 32;
32    //@ ensures MessageDigest.isHash256(bb,\result);
33    //@ ensures \result !=null;
34    //@ pure
35    public static byte[] computeSha256(byte[] bb) {
36
37       try {
38          MessageDigest sha256_md = MessageDigest.getInstance("SHA-256");
39          final byte[] hashedMessage = sha256_md.digest(bb);
40          return hashedMessage;
41       } catch (NoSuchAlgorithmException e) {
42          return null;
43       }
44    }
```

Listing 2.23: temp

```
1   //@ public normal_behavior
2   //@ requires a1!= null;
3   //@ requires a2!= null;
4   //@ requires (a1.length + a2.length)<= Integer.MAX_VALUE ;
5   //@ ensures \result.length == a1.length + a2.length;
6   //@ ensures (\forall int i; 0<=i <a1.length ; \result[i] == \old(a1[i]));
7   //@ ensures (\forall int j; 0<=j <a2.length ; \result[a1.length+j] ==
        \old(a2[j]));
8   //@ ensures \result != null;
9   //@ pure
10  public static byte[] concat2byteArray(byte[] a1, byte[] a2) {
11     byte[] res = new byte[a1.length + a2.length];
12     System.arraycopy(a1, 0, res, 0, a1.length);
13     System.arraycopy(a2, 0, res, a1.length, a2.length);
14     return res;
15  }
```

2.3.2.6 Step 4 in ECIES encryption: Symmetric Encryption

2.3.2.6.1 Specification Symmetric Encryption step of ECIES for both profiles A and B use the AES128 encryption algorithm in CTR mode (see Table 2.2). In Step 3 of ECIES we saw that the output of Key Derivation has length of 64 bytes. The most significant "enckeylen" bytes (16 bytes) of that is Ephemeral Encryption Key, the least significant mackeylen bytes are Ephemeral MAC Key (32 bytes) and the remaining icblen bytes (16 bytes) is ICB [20, pp. 207].

2.3.2.6.2 Verification and refactoring For verification of Symmetric Encryption (Step 4 of ECIES encryption in Figure 2.9 first we complete javax.crypto.Cipher.jml (see Listing 2.24, Listing 2.25 and Listing 2.26), javax.crypto.spec.IvParameterSpec.jml (see Listing 2.27) and java.security.spec.AlgorithmParameterSpec (see Listing 2.1).

In javax.crypto.Cipher.jml, the model method validAlgorithm(String algorithm) is used in the specification of method getInstance(String s) (see Listing 2.24). The specification of validAlgorithm(String algorithm) is completed based on [34] and [32] in a way that now the prover realizes that AES/CTR/NoPadding is a valid input for getInstance(java.lang.String s).

Listing 2.24: complete javax.crypto.Cipher.jml: getInstance(java.lang.String s)

```
1    //------------------------------ Author: Negar ----------------------
2    //@ model public final instance java.lang.String algorithm;
3
4    //@ public normal_behavior
5    //@ ensures (((algorithm == "AES/CTR/NoPadding")) ==> \result);
6     // pure
7    //----------------------------------- end -----------------------------
8     //@ model public static pure heap_free boolean validAlgorithm(String
         algorithm);
9
10
11
12    //@ public normal_behavior
13   //------------------------------ Author: Negar ----------------------
14   //@   requires validAlgorithm(s);
15    //@   ensures \result.algorithm == s;
16    //----------------------------------- end -----------------------------
17    //@   ensures \fresh(\result);
18   //------------------------------ Author: Negar ----------------------
19    //@ also public exceptional_behavior
20    //@   requires !validAlgorithm(s);
21    //@   signals_only java.security.NoSuchAlgorithmException,
         javax.crypto.NoSuchPaddingException;
22    //----------------------------------- end -----------------------------
23    //@ pure
24    public static final javax.crypto.Cipher getInstance(java.lang.String s)
         throws java.security.NoSuchAlgorithmException,
         javax.crypto.NoSuchPaddingException;
```

In Listing 2.25 for javax.crypto.Cipher.jml, we rewrite specification of init(int n, Key key, AlgorithmParameterSpec spec) [33]. In Listing 2.26 we complete specification of method doFinal(byte[] b).

Listing 2.25: rewrite javax.crypto.Cipher.jml: init(int n, Key key, AlgorithmParameterSpec spec)

```
1  //------------------------------ Author: Negar ----------------------
2    //@ model public instance byte[] keyByteArray;
3    //@ model public instance byte[] ivbByteArray;
```

```
4    //@ model public instance int modeOperation;
5
6
7    //@ public normal_behavior
8     //https://docs.oracle.com/en/java/javase/14/docs/api/java.base/javax/crypto/Cipher.html#i
9    //below 3 lines added for normal behavior requirements inorder to have
        proper key for "AES/CTR/NoPadding" (not throwing
        java.security.InvalidKeyException)
10    //@ requires(algorithm == "AES/CTR/NoPadding")==> (key instanceof
        javax.crypto.spec.SecretKeySpec ivspec );
11   //@ requires(algorithm == "AES/CTR/NoPadding")==> (key.getAlgorithm() ==
        "AES");
12   //@ requires(algorithm == "AES/CTR/NoPadding")==> ((key.getEncoded().length
        == (128/8))||(key.getEncoded().length ==
        (192/8))||(key.getEncoded().length == (256/8)));
13
14   //below line is to make sure there is no
        ava.security.InvalidAlgorithmParameterException
15   //@ requires (n == Cipher.ENCRYPT_MODE) || (n == Cipher.DECRYPT_MODE);
16
17   //below line is to make sure there is no
        java.security.InvalidAlgorithmParameterException
18   //@  requires (algorithm == "AES/CTR/NoPadding")==> (spec instanceof
        javax.crypto.spec.IvParameterSpec );
19
20    //@   assignable this.content, this.keyByteArray, this.ivbByteArray;
21    //@   ensures isInitialized;
22   //@   ensures keyByteArray == \old(key.getEncoded());
23   //@   ensures ivbByteArray == \old(spec.ivb);
24   //@ ensures modeOperation == n;
25   // ensures (spec instanceof javax.crypto.spec.IvParameterSpec) ==>
        (ivbByteArray == .getIV());
26    //---------------------------------- end -----------------------------
27    public final void init(int n, java.security.Key key,
        java.security.spec.AlgorithmParameterSpec spec) throws
        java.security.InvalidKeyException,
        java.security.InvalidAlgorithmParameterException;
```

Listing 2.26: complete javax.crypto.Cipher.jml: doFinal(byte[] b)

```
1  //------------------------------ Author: Negar -----------------------
2  //@ ensures (\forall byte[] keyCopy; (keyCopy.length ==
       key.length)&&(java.util.Arrays.equalArrays(keyCopy,0,key,0,key.length));
       isAES_CTR_ENC(output , input,key , ivb ) <==> isAES_CTR_ENC(output ,
       input,keyCopy , ivb ) );
3  //@ ensures (\forall byte[] ivbCopy; (ivbCopy.length ==
       ivb.length)&&(java.util.Arrays.equalArrays(ivbCopy,0,ivb,0,ivb.length));
       isAES_CTR_ENC(output , input,key , ivb ) <==> isAES_CTR_ENC(output ,
       input,key , ivbCopy ) );
4   //ciphertext and plaintext have the same size in AES CTR NoPadding mode
5  //@ ensures \result ==> input.length == output.length;
6  //@ assignable \nothing;
7  //@ pure
8  //@ model public static boolean isAES_CTR_ENC( byte[] output , byte[]
       input,byte[] key , byte[] ivb );
9  //@ ensures (\forall byte[] keyCopy; (keyCopy.length ==
       key.length)&&(java.util.Arrays.equalArrays(keyCopy,0,key,0,key.length));
       isAES_CTR_DEC(output , input,key , ivb ) <==> isAES_CTR_DEC(output ,
       input,keyCopy , ivb ) );
10 //@ ensures (\forall byte[] ivbCopy; (ivbCopy.length ==
       ivb.length)&&(java.util.Arrays.equalArrays(ivbCopy,0,ivb,0,ivb.length));
       isAES_CTR_DEC(output , input,key , ivb ) <==> isAES_CTR_DEC(output ,
       input,key , ivbCopy ) );
11 //@ assignable \nothing;
12 //@ pure
13 //@ model public static boolean isAES_CTR_DEC( byte[] output , byte[]
       input,byte[] key , byte[] ivb );
14 //---------------------------------- end ----------------------------


16

17  //@ public normal_behavior
18 //------------------------------ Author: Negar -----------------------
19 //in normal behavior it is needed that object was initialized before :
       https://docs.oracle.com/en/java/javase/14/docs/api/java.base/javax/crypto/Cipher.html#d
20  //@ requires isInitialized;
21 //@ ensures ((modeOperation == Cipher.ENCRYPT_MODE)&&(algorithm ==
       "AES/CTR/NoPadding"))==> isAES_CTR_ENC( \result , b ,keyByteArray ,
       ivbByteArray );
22  //@ ensures ((modeOperation == Cipher.DECRYPT_MODE)&&(algorithm ==
        "AES/CTR/NoPadding"))==> isAES_CTR_DEC( \result , b ,keyByteArray ,
```

```
        ivbByteArray );
23    //---------------------------------- end -----------------------------
24     //@   assignable this.content;
25     //@   ensures \fresh(\result);
26     //@   ensures \result.length == b.length; // FIXME - is this so? perhaps
          just when there have been no prvious calls?
27     //@   ensures (* FIXME OPENJML - length of output? is it really fresh? *);
28     public final byte[] doFinal(byte[] b) throws
          javax.crypto.IllegalBlockSizeException, javax.crypto.BadPaddingException;
```

Finally we complete javax.crypto.spec.IvParameterSpec.jml (see Listing 2.27) [35].

Listing 2.27: complete javax.crypto.spec.IvParameterSpec.jml

```
1     //@ spec_public
2     //---------------------------------- end -----------------------------
3       private byte[] iv;
4
5     //@ public normal_behavior
6     //------------------------------ Author: Negar ----------------------
7     //@   ensures this.iv != buf;
8     //@   ensures this.iv.length == buf.length;
9     //@   ensures java.util.Arrays.equalArrays(this.iv,0,buf,0,buf.length);
10    //@ ensures this.ivb == this.iv;
11    //---------------------------------- end -----------------------------
12    //@   ensures true;
13    //@ pure
14      public IvParameterSpec(byte[] buf);
15
16    //------------------------------ Author: Negar ----------------------
17    //@ public normal_behavior
18    //@ ensures \result == iv;
19    //@ pure
20    //---------------------------------- end -----------------------------
21    public byte[] getIV();
```

Listing 2.28 demonstrates implementation and verification of Step 4 of ECIES encryption: symmetric encryption (AES-128 in CTR mode with no padding).

Listing 2.28: implementation and verification of Step 4 of ECIES encryption: Symmetric

Encryption

```
1  //@ public normal_behavior
2     //@ requires ((key!=null) && (key.length ==(128/8)));
3     //@ requires ((icb!=null)&&(icb.length ==(128/8)));
4     //@ ensures Cipher.isAES_CTR_ENC( \result , plainText ,key , icb );
5     //@ pure
6     public static byte[] symmetricEncryption(byte[] key, byte[] icb, byte[]
          plainText) {
7        try {
8           Key keySpec = new SecretKeySpec(key, "AES");
9
10          AlgorithmParameterSpec ivspec = new IvParameterSpec(icb);
11
12          Cipher cipher = Cipher.getInstance("AES/CTR/NoPadding");
13
14          cipher.init(Cipher.ENCRYPT_MODE, keySpec, ivspec);
15
16          byte[] result = cipher.doFinal(plainText);
17          return result;
18       } catch (NoSuchAlgorithmException | NoSuchPaddingException |
             InvalidKeyException
19             | InvalidAlgorithmParameterException | IllegalBlockSizeException |
                 BadPaddingException e) {
20          return null;
21       }
22
23    }
```

2.3.2.7 Step 3 in ECIES decryption: Symmetric Decryption

2.3.2.7.1 Specification Symmetric Decryption step of ECIES for both profiles A and B use AES128 decryption algorithm in CTR mod (see Table 2.2).

2.3.2.7.2 Verification and refactoring For verification of Symmetric Decryption (Step 3 of ECIES decryption in Figure 2.10) first we complete javax.crypto.Cipher.jml by adding lines 9-13 and 23 in Listing 2.26.

Listing 2.29 demonstrates implementation and verification of Step 3 of ECIES decryption, Symmetric Decryption (AES-128 in CTR mode with no padding).

Listing 2.29: implementation and verification of Step 3 of ECIES decryption: Symmetric Decryption

```
1   //@ public normal_behavior
2   //@ requires ((key!=null) && (key.length ==(128/8)));
3   //@ requires ((icb!=null)&&(icb.length ==(128/8)));
4   //@ ensures Cipher.isAES_CTR_DEC( \result , cipherText ,key , icb );
5   //@ pure
6   public static byte[] symmetricDecryption(byte[] key, byte[] icb, byte[]
        cipherText) {
7      //ciphertext_SUCI
8
9      try {
10
11        Key keySpec = new SecretKeySpec(key, "AES");
12
13        AlgorithmParameterSpec ivspec = new IvParameterSpec(icb);
14
15        Cipher cipher = Cipher.getInstance("AES/CTR/NoPadding");
16        cipher.init(Cipher.DECRYPT_MODE, keySpec, ivspec);
17        return cipher.doFinal(cipherText);
18     }
19
20     catch (NoSuchAlgorithmException | NoSuchPaddingException |
           InvalidKeyException
21           | InvalidAlgorithmParameterException | IllegalBlockSizeException |
               BadPaddingException e) {
22        return null;
23     }
24
25  }
```

2.3.2.8 Step 5 in ECIES encryption: MAC Function

2.3.2.8.1 Specification The MAC Function for both profiles A and B is HMACSHA-256 (see Table 2.2). The output of HMACSHA-256 has a length of 256 bits (32 bytes) [64].

The MAC-tag value generated in step 5 of ECIES has length maclen, which is 8 bytes for either profile A or B. MAC-tag in ECIES described as the maclen most significant octets of the output of the HMAC function [20, pp. 208]. For an example see [20, pp. 211]. By mistake, the entire 32 bytes output of HMACSHA-256 in [69] was mistakenly regarded as the MAC-tag.

2.3.2.8.2 Verification and refactoring For verification of MAC Function (Step 5 of ECIES encryption in Figure 2.9) first we complete javax.crypto.Cipher.jml (see Listing 2.30).

In javax.crypto.Mac.jml, the model method validAlgorithm is used in specification of getInstance(String s) method. In Listing 2.30, we complete the specification of the method based on [38] and [36] so that it now recognizes that "HmacSHA256" as a valid input for getInstance(java.lang.String s).

Listing 2.30: complete javax.crypto.Mac.jml: getInstance(java.lang.String s)

```
1     //@ public normal_behavior
2     //------------------------------ Author: Negar -----------------------
3     //@ ensures ((algorithm =="HmacSHA256") ==> \result);
4     //------------------------------------ end -----------------------------
5      //@ model public static pure heap_free boolean validAlgorithm(String
          algorithm);
6
7      //@ public normal_behavior
8      //@   requires validAlgorithm(algorithm);
9      //@   ensures \fresh(\result);
10     //@   ensures \result.algorithm == algorithm;
11     //@ also public exceptional_behavior
12     //@   requires !validAlgorithm(algorithm);
13     //@   signals_only java.security.NoSuchAlgorithmException;
14     //@ pure
15     public static final Mac getInstance(java.lang.String algorithm) throws
          java.security.NoSuchAlgorithmException;
```

For javax.crypto.Mac.jml, in Listing 2.31, we also complete specification of the function init(java.security.Key key) iin such a way that the prover could get that the init function terminates normally (without error) once the input key is valid for the algorithm specified in previous steps (using getInstance method) [39].

Listing 2.31: complete javax.crypto.Mac.jml: init(java.security.Key key)

```
1   //------------------------------ Author: Negar ----------------------
2   //@ model public instance byte[] keyByteArray;
3   //---------------------------------- end --------------------------
4
5
6   //@ public normal_behavior
7   //@   assignable this.*;
8   //@   ensures this.initialized == true;
9   //@   ensures this.isReset == true;
10  //@   ensures this.savedInfo == null;
11  //@// TODO: Model InvalidKeyException
12  //------------------------------ Author: Negar ----------------------
13  //@ public normal_behavior
14  //@ requires(algorithm == "HmacSHA256")==> (key instanceof
        javax.crypto.spec.SecretKeySpec );
15  //@ requires(algorithm == "HmacSHA256")==> (key.getAlgorithm() ==
        "HmacSHA256");
16  //@   assignable this.initialized, this.isReset, this.savedInfo, this.
        keyByteArray;
17  //@   ensures keyByteArray == \old(key.getEncoded());
18  //@   ensures this.initialized == true;
19  //@   ensures this.isReset == true;
20  //@   ensures this.savedInfo == null;
21  //---------------------------------- end --------------------------
22  public final void init(java.security.Key key) throws
        java.security.InvalidKeyException;
```

Listing 2.32 demonstrates the completion of the specification for the method doFinal(byte[] input). In order for the function to behave normally when called for an object of the class javax.crypto.Mac, the object must be initialized beforehand [37] (see line 22 of Listing 2.32).

Listing 2.32: input)]complete javax.crypto.Mac.jml: doFinal(byte[] input)

```
1   //------------------------------ Author: Negar ----------------------
2   //@ ensures (\forall byte[] keyCopy; (keyCopy.length ==
        key.length)&&(java.util.Arrays.equalArrays(keyCopy,0,key,0,key.length));
        isHmacSHA256(output , input,key ) <==> isHmacSHA256(output ,
        input,keyCopy ) );
```

```
3       //@ ensures (\forall byte[] keyCopy; (keyCopy.length ==
            key.length)&&(\forall int index; 0<=index< keyCopy.length;
            keyCopy[index]==\old(key)[index]); isHmacSHA256(output , input,key )
            <==> isHmacSHA256(output , input,keyCopy ) );
4
5       //@ ensures (\forall byte[] inputCopy; (inputCopy.length ==
            input.length)&&(java.util.Arrays.equalArrays(inputCopy,0,input,0,input.length));
            isHmacSHA256(output , input,key ) <==> isHmacSHA256(output ,
            inputCopy,key ) );
6       //@ ensures (\forall byte[] inputCopy; (inputCopy.length ==
            input.length)&&(\forall int index; 0<=index< inputCopy.length;
            inputCopy[index]==\old(input)[index]); isHmacSHA256(output , input,key
            ) <==> isHmacSHA256(output , inputCopy,key ) );
7
8       //@ ensures \result ==> output.length == 32;
9
10      //below line is needed as we expect (determinism)computing HmacSHA256
            (input, key) always get us same result
11      //@ ensures (\forall byte[] outputCopy; isHmacSHA256(outputCopy ,
            input,key ) && isHmacSHA256(output , input,key
            );Arrays.equals(outputCopy,output));
12
13      //@ assignable \nothing;
14      //@ ensures \result ==> (output.length==32);
15      //@ pure
16      //@ model public static boolean isHmacSHA256( byte[] output , byte[]
            input,byte[] key );
17      //---------------------------------- end -----------------------------
18
19
20      //@ public normal_behavior
21      //------------------------------ Author: Negar ----------------------
22      //@ requires this.initialized;
23      //@ ensures (algorithm == "HmacSHA256")==> isHmacSHA256( \result , input
            ,keyByteArray );
24      //---------------------------------- end -----------------------------
25      //@   { update(input,0,(int)input.length); return doFinal(); }
26      public final byte[] doFinal(byte[] input) throws
            java.lang.IllegalStateException;
```

Listing 2.33 demonstrates implementation and verification of Step 5 of ECIES Encryption, MAC Function.

Listing 2.33: implementation and verification of Step 5 of ECIES encryption: MAC Function

```
1   //@ public normal_behavior
2   //@ requires (key!=null);
3   //@ requires (inputTextVal!=null);
4   //@ ensures Mac.isHmacSHA256( \result , inputTextVal ,key );
5   //@ pure
6   public static byte[] macFn(byte[] inputTextVal, byte[] key) {
7
8      try {
9         Mac sha256Hmac = Mac.getInstance("HmacSHA256");
10        SecretKeySpec keySpec = new SecretKeySpec(key, "HmacSHA256");
11        sha256Hmac.init(keySpec);
12        byte[] macData = sha256Hmac.doFinal(inputTextVal);
13        return macData;
14     } catch (NoSuchAlgorithmException | InvalidKeyException e) {
15        return null;
16     }
17  }
18  //-------------------------------------------------
19
20  //@ public normal_behavior
21  //@ requires (MacValue32byte.length ==32);
22  //@ ensures ((\result.length == 8) && (\forall int i; 0 <= i < 8;
        \result[i-0] == MacValue32byte[i]));
23  //@ pure
24  public static byte[] stepFiveMacFn8byte(byte[] MacValue32byte) {
25     byte[] macTag8byte = Arrays.copyOfRange(MacValue32byte, 0, 8);
26     return macTag8byte;
27  }
```

2.3.3 Static verification of edges of the process graph

We can represent a part of the protocol as a graph in which each node performs a specific process on its inputs and generates an output. Thus far, we have verified that each node

performs its designated task. However, it is not sufficient to stop at verifying each node; we must also ensure that the nodes are connected as they should be. For instance, consider a scenario where all nodes are verified, but there is no connection between them. To illustrate, let us assume that the portion of the protocol consists of only two nodes (① and ②) and an edge from node ① to node ② (we assume the output of ① is a byte array). If we have already verified the two steps as shown in Figure 2.16, and there exists another node (node ③) such that $post③ \rightarrow pre②$, then we can use the output of node ③ as the input to node ②, as shown in Figure 2.17.

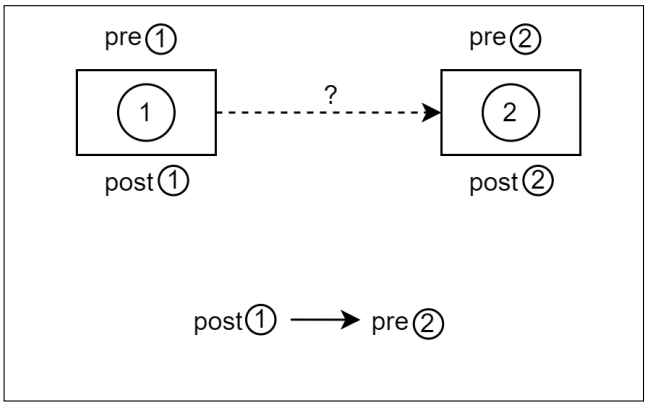


Figure 2.16: Example of a protocol section with two verified nodes

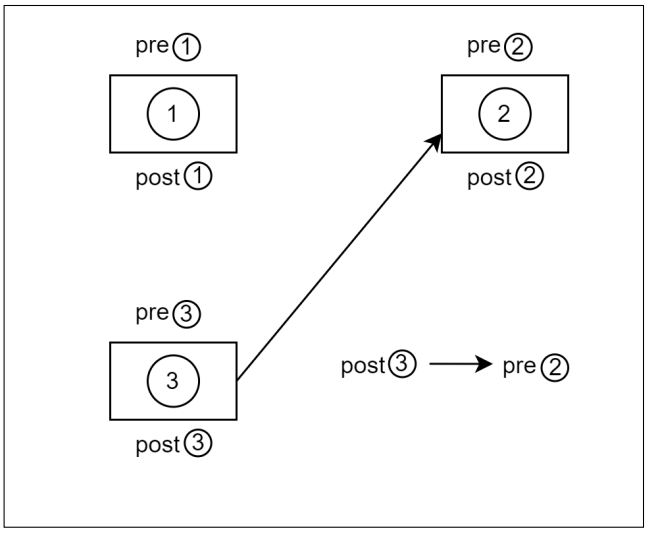


Figure 2.17: Weakness of Just Verifying Nodes

The next step is to verify that the edges are also captured. In other words, we want to ensure that we call ② given outputs of ① as its input. We will explore a few possible solutions for this and compare them.

2.3.3.1 First Solution

The idea here is to wrap existing methods ①, and ② and enhanced pre and post-condition of them as Figure 2.18. Here we don't put any specifications for the model method.

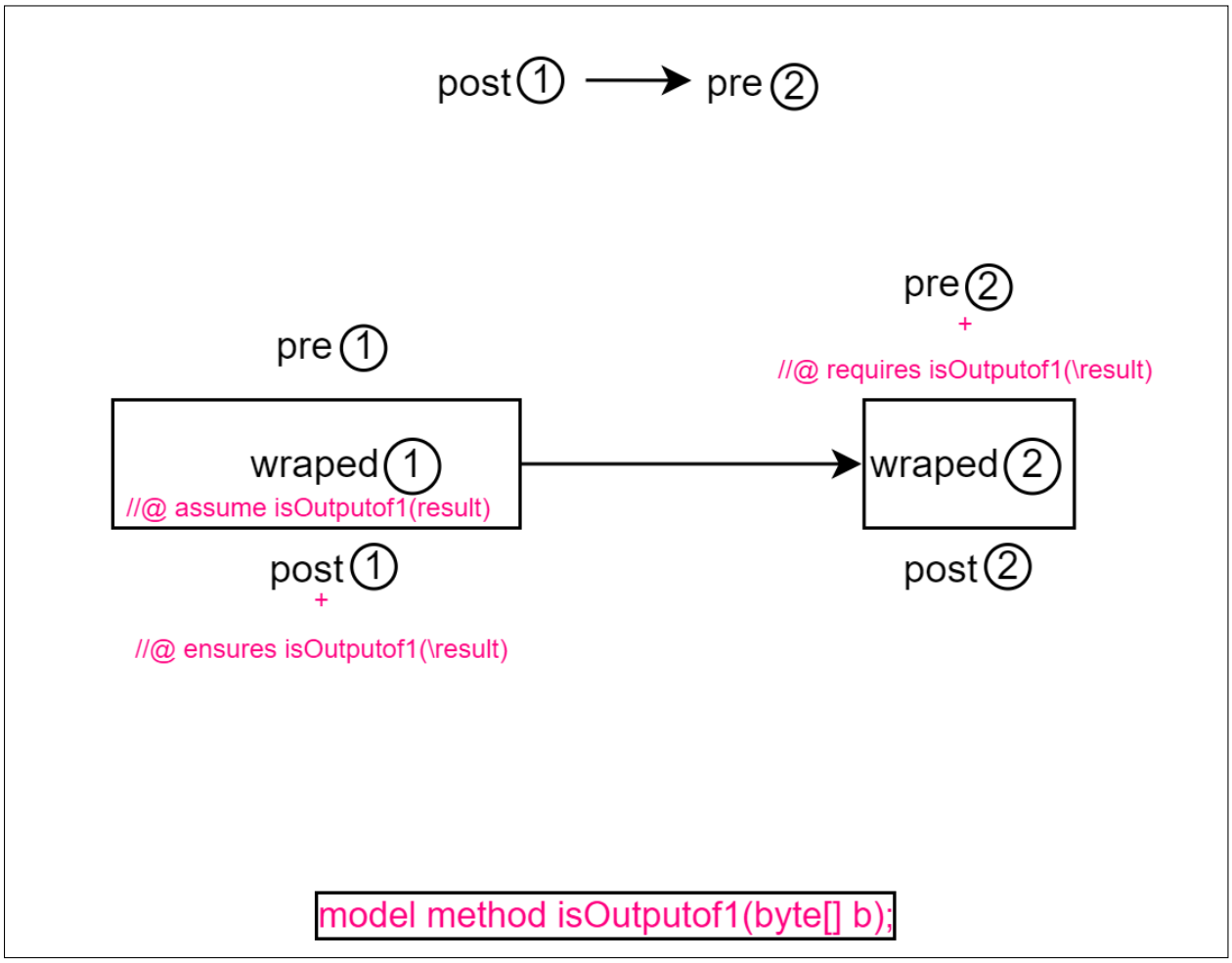


Figure 2.18: Solution1 for Verifying Edges

2.3.3.2 Second Solution

In the second solution, we apply a similar approach as in the first solution by wrapping the existing methods ① and ② and enhancing their pre- and post-conditions as shown in Figure 2.19. Additionally, we introduce a new postcondition for the model method.

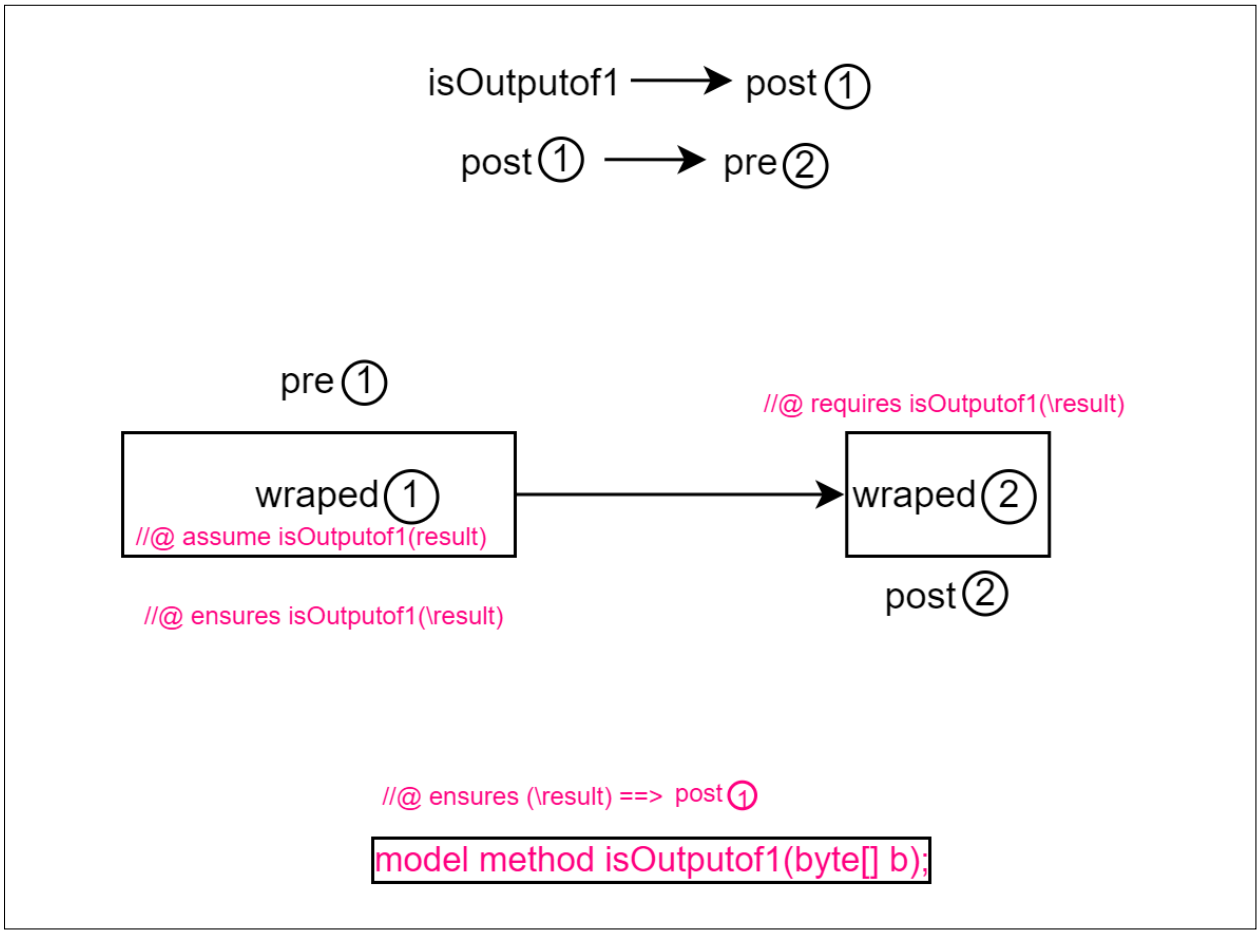


Figure 2.19: Solution2 for Verifying Edges

- Note 1: Can we remove the "assume" in 2.19? No. If we remove it, we would need to change $\Rightarrow$ to $\Longleftrightarrow$, which would essentially rename $post$① to $isOutputof1$. We do not want to do this because the result of any other function, such as ③, that has post $post$③ $\Rightarrow$ $post$① could still be fed into wrapped ②. However, we want to

guarantee that wrapped ② can only be called by providing the output of wrapped ① as its input.

- Note 2: If we use the "assume" in Figure 2.19, we are claiming too much as humans because isOutputof1 has a postcondition itself. Ultimately, using the second solution will result in several of these "assumes", making it highly probable that a human will make a mistake in at least one of them.

2.3.3.3 Third Solution

The third solution eliminates the weaknesses of the second solution, which relied on a strong assumption. Like the first solution in the third solution, we do not need to include post① in the post-condition of wrapped①. The third solution is presented in Figure 2.20.

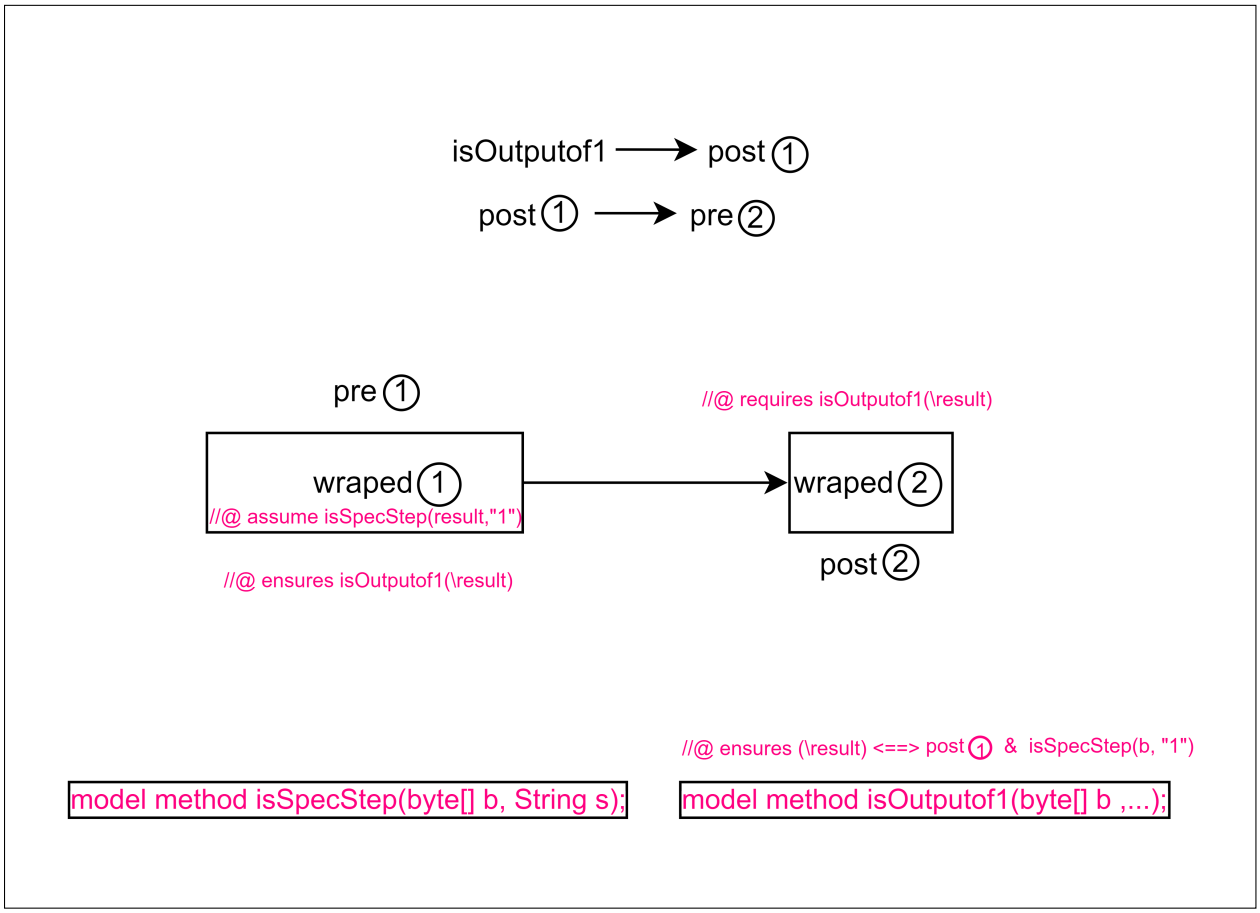


Figure 2.20: Solution3 for Verifying Edges

It may be a question whether the function isOutputof1 can only have one input of type string, as shown in Figure 2.21. Consider the scenario where wrapped① is called and isSpecStep("1") is true. If another function like ③ has $post③ \Rightarrow post①$, it could still be fed into wrapped②. However, we want to ensure that wrapped② can only be called by giving the output of wrapped① as its input. Therefore, the simplified version of the third solution shown in Figure 2.21 is not acceptable.

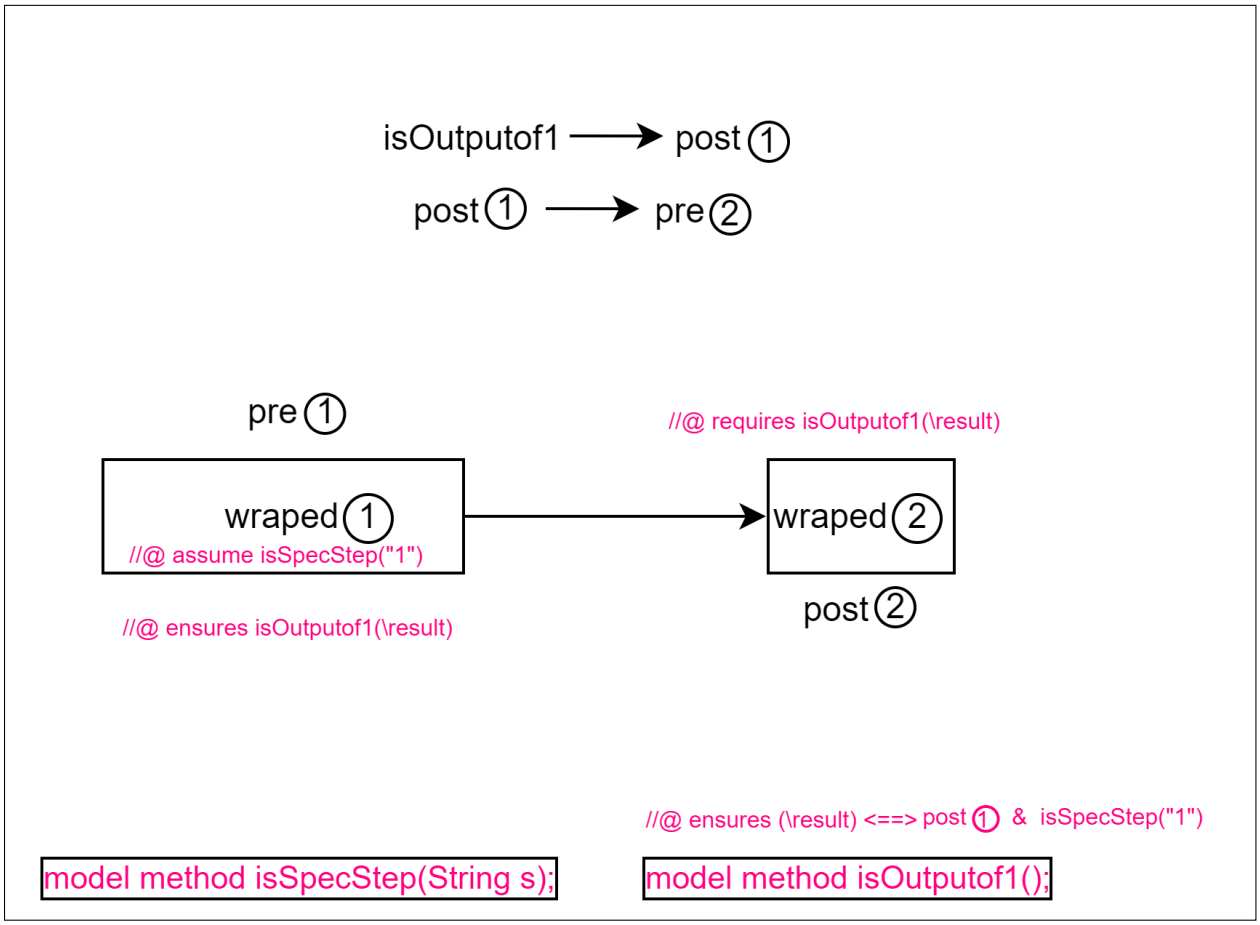


Figure 2.21: simplified version of solution3 for Verifying Edges that is not acceptable

2.3.3.4 Fourth Solution

In the fourth solution shown in Figure 2.22, we only need one model method to capture an edge, as in the first solution. Similar to the third solution, we use a model method with two inputs that can be reused for capturing other edges.

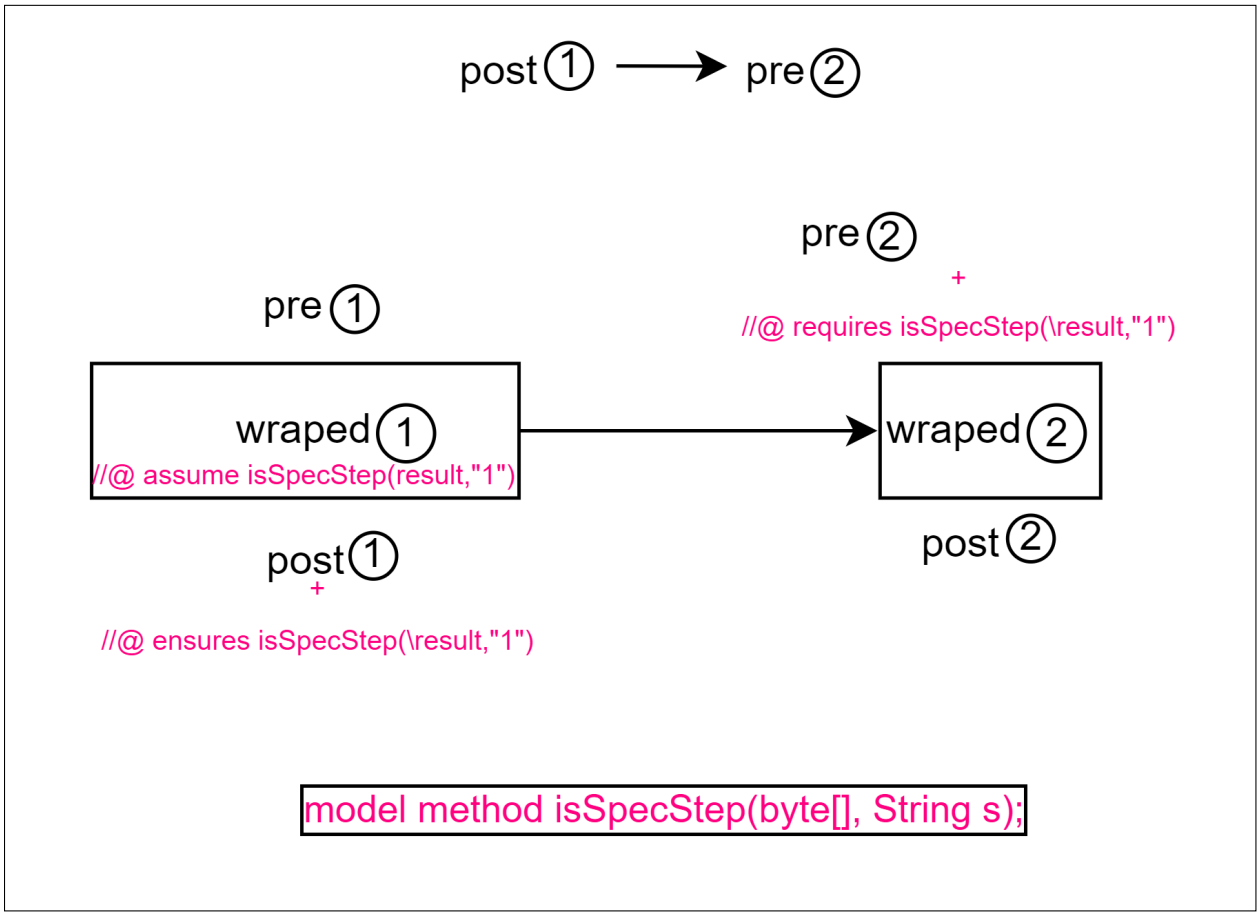


Figure 2.22: Solution4 for Verifying Edges

We applied the fourth solution for capturing edges to ECIES encryption for generating Scheme Output. This process is depicted in Figure 2.23, which shows the resulting encrypted data along with the corresponding edges captured from the input data.

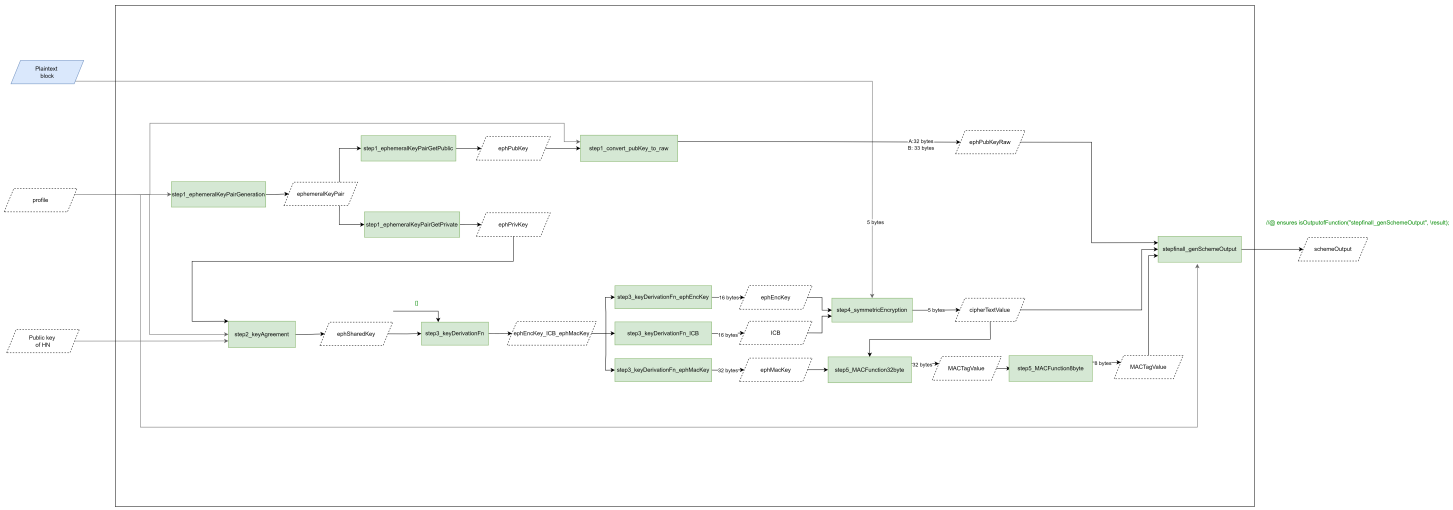


Figure 2.23: ECIES encryption for generating Scheme Output _ Data Flow diagram

2.4 5G-AKA authentication protocol

Figure 2.24 illustrates authentication protocol showing sub-entities and Figure 2.25 demonstrates authentication protocol showing three main entities.

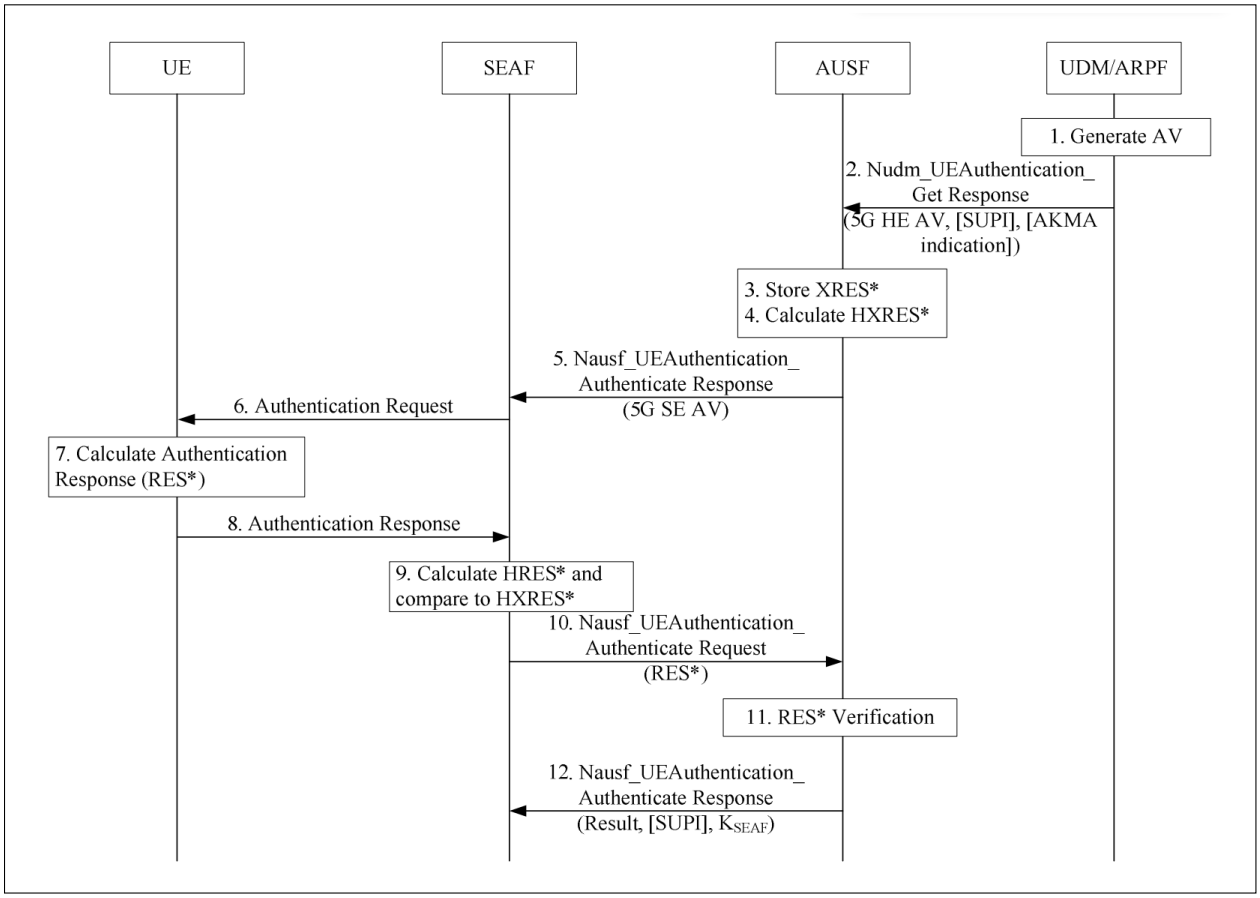


Figure 2.24: Authentication protocol: showing sub-entities [20, p. 43]

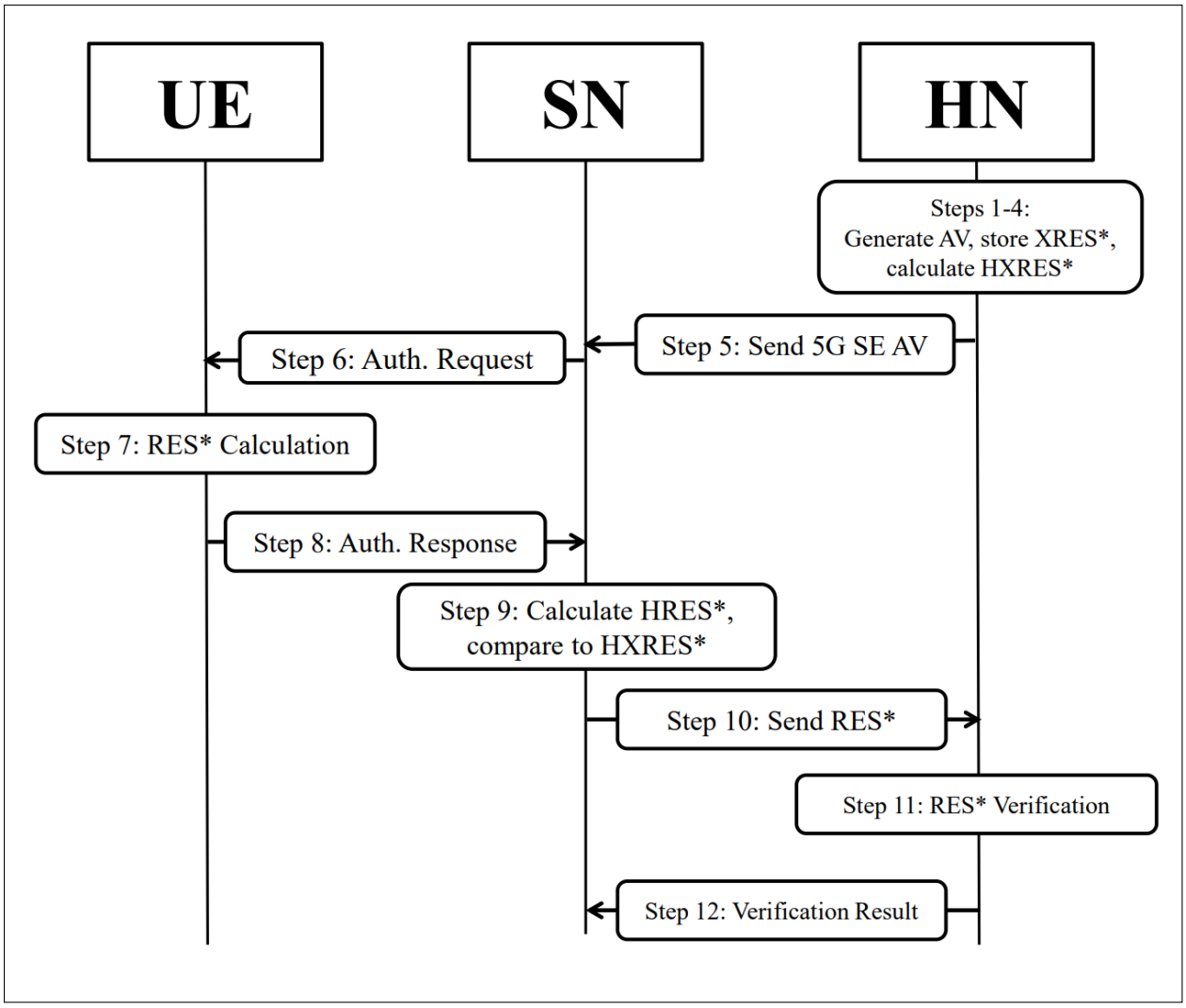


Figure 2.25: Authentication protocol: showing main entities [69, p. 13]

2.4.1 Steps 1-4 of 5G-AKA authentication protocol

In the first step of 5G-AKA authentication protocol, sub-entity UDM/ARPF at HN computes 5G HE AV [20, pp. 43]. 5G HE AV is concatination of RAND, Authentication Token (AUTN), $XRES^*$, and K_{AUSF} (RAND|| $XRES^*$|| K_{AUSF}|| AUTN) [20, pp. 43].

In step 2 of 5G-AKA authentication protocol,UDM/ARPF sends 5G HE AV (and SUPI in case HN received SUCI in initialization step) to sub-entity AUSF at HN [20, pp. 43].

In step 3 of 5G-AKA authentication protocol, AUSF temporally stores $XRES^*$ (part of a received 5G HE AV) with SUPI (if it was received in the previous step) [20, pp. 44].

In step 4 of 5G-AKA authentication protocol, AUSF computes $HXRES^*$ from $XRES^*$ (part of a received 5G HE AV) [20, pp. 44]. AUSF also computes K_{SEAF} from K_{AUSF} (part of a received 5G HE AV) [20, pp. 44]. AUSF then computes 5G AV from the 5G HE AV by replacing the KAUSF with KSEAF and $XRES^*$ with the $HXRES^*$ in the 5G HE AV [20, pp. 44]. So the formula for 5G AV would be (RAND|| $HXRES^*$|| K_{SEAF}|| AUTN).

In step 5 of 5G-AKA authentication protocol, AUSF computes 5G SE AV from 5G AV by removing K_{SEAF} [20, pp. 44]. So the formula for 5G SE AV would be (RAND|| $HXRES^*$|| AUTN). AUSF sends the 5G SE AV to SN (sub-entity SEAF) [20, pp. 44].

If we look at main entities, in steps 1 to 5, HN generate 5G SE AV (RAND|| $HXRES^*$|| AUTN) and sends it to SN. Once HN computes RAND, $HXRES^*$ and AUTN, it can compute 5G SE AV by concatenating them and there is no need to compute 5G Home Environment Authentication Vector (5G HE AV) and 5G AV. This could also be seen in Figure 2.26.

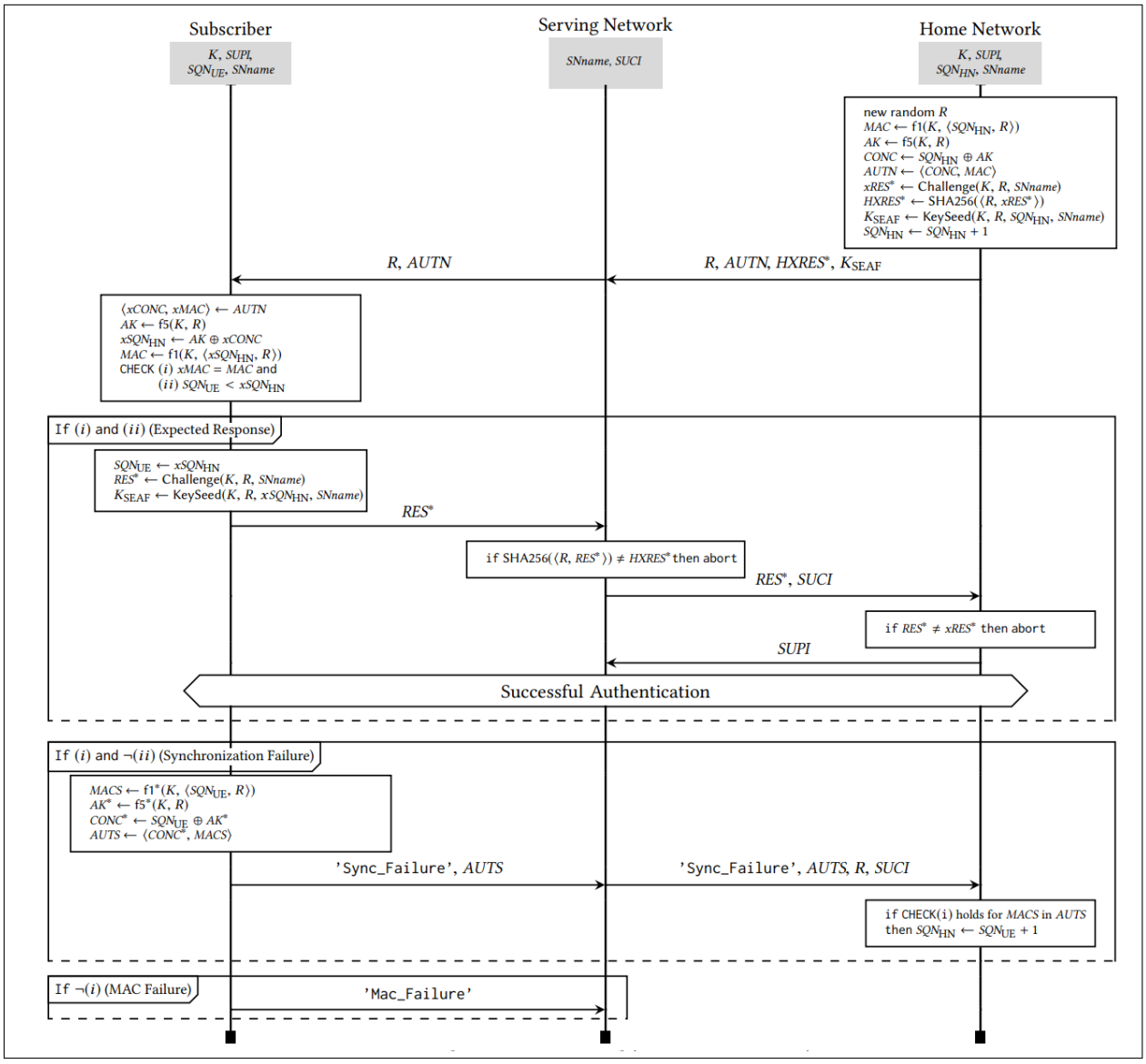


Figure 2.26: Authentication protocol: showing main entities and exceptional cases [27, p. 5]

Inputs and outputs of Steps One to Four in the 5G-AKA Authentication protocol are illustrated in Figure 2.27. The data flow diagram for these steps is shown in Figure 2.28.

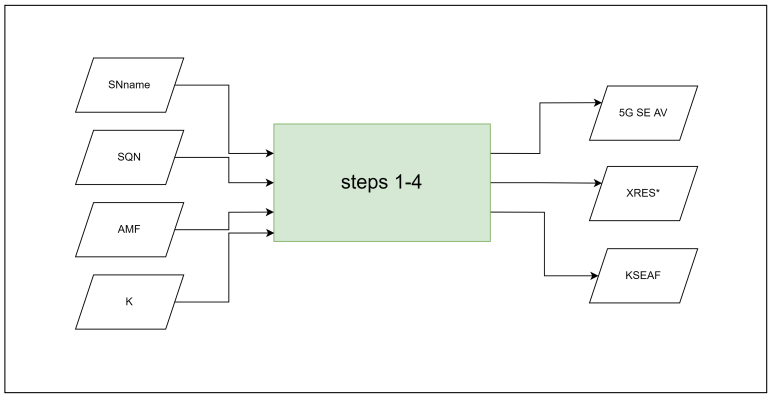


Figure 2.27: Steps One to Four in 5G-AKA Authentication protocol _ Inputs and Outputs

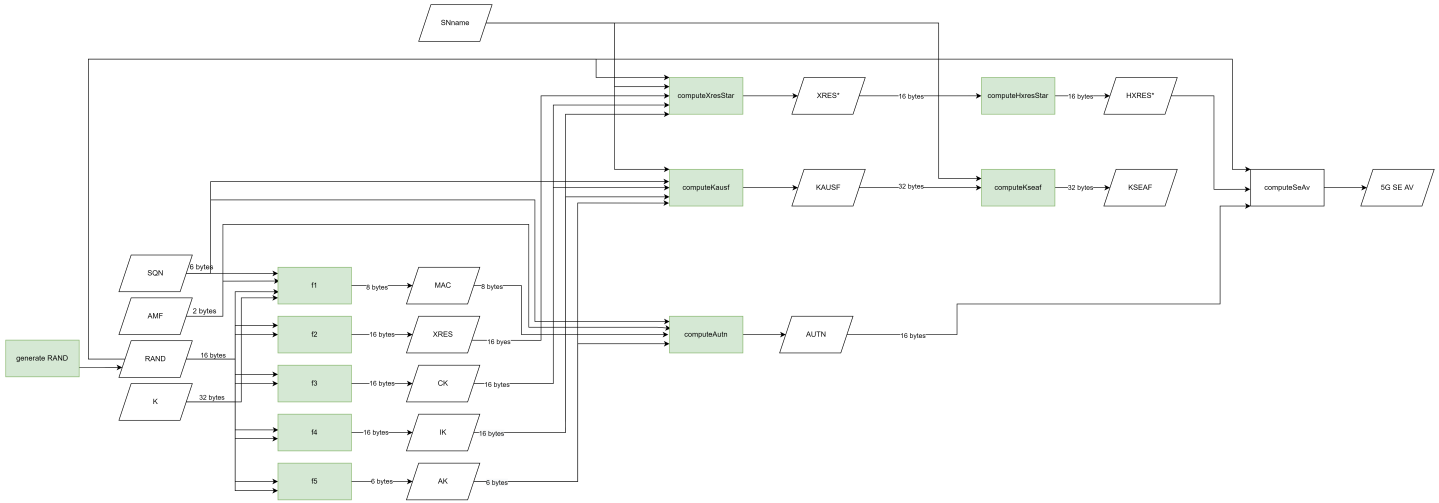


Figure 2.28: Steps One to Four in 5G-AKA Authentication protocol _ Data Flow diagram

2.4.1.1 RAND

2.4.1.1.1 Specification The RAND has a length of 16 bytes [19].

2.4.1.1.2 Verification and Refactoring In [69, p. 34] was decided to be set to a value between 0 and 147483637. Listing 2.34 demonstrates implementation of the RAND generation from [69] which is clearly does not even meet producing a value between 0 and 147483637.

Listing 2.34: implementation of the RAND generation from [69]

```
1  Random randObj = new Random();
2  int maxRandVal = 147483637;
3  int maxLengthofRandVal = Integer.toString(maxRandVal).length();
4  int generatedRANDval = randObj.nextInt(maxRandVal);
5  String generatedRANDval_str = Integer.toString(generatedRANDval);
6
7  if(maxLengthofRandVal>Integer.toString(generatedRANDval).length())
8  {
9
10    int diffVal = maxLengthofRandVal -
          Integer.toString(generatedRANDval).length();
11
12    for (int x = 0; x < diffVal; x++) {
13
14       generatedRANDval_str += "0";
15
16    }
17 }
18
19 generatedRANDval=Integer.parseInt(generatedRANDval_str);
```

In the rest of this section we go thought the refactored version of RAND generation and verify that it meets the specification. For verification of RAND generation first we complete javax.crypto.Cipher.jml.

In lines 2 to 4 of Listing 2.35 we define the model method isRandBytes(byte[] input) which is reponsible to return true only in case its input was generated by a random process. Next, we add line 9 to indicate after calling the function nextBytes with the input bytes (nextBytes(bytes)), "bytes" is filled randomly [40].

Listing 2.35: complete java.util.Random.jml

```
//------------------------------ Author: Negar ----------------------
//@ assignable \nothing;
//@ pure
//@ model public static boolean isRandBytes(byte[] input);
//------------------------------ end Negar ----------------------

//@ public normal_behavior
//------------------------------ Author: Negar ----------------------
//@ ensures isRandBytes(bytes);
//------------------------------ end Negar ----------------------
//@  assignable this.content, bytes[*];
public void nextBytes(byte[] bytes);
```

Listing 2.36 demonstrates implementation and verification of RAND generation

Listing 2.36: implementation and verification of RAND generation

```
//@ public normal_behavior
//@ ensures \result.length == 16;
//@ ensures Random.isRandBytes(\result);
//@ pure
public static byte[] generateRand() {
   Random rd = new Random();
   byte[] rand = new byte[16];
   rd.nextBytes(rand);
   return rand;
}
```

2.4.1.2 SQN

SQN is a 6-byte value [19, p. 29]. At HN, SQN is denoted by SQN_{HN} and is strictly increasing [27, p. 10]. On the other hand, at UE, SQN is denoted by SQN_{UE} and represents the highest SQN that has been accepted by USIM [71, p. 4].

2.4.1.3 Serving network name

In 5G-AKA, Serving network name is (("5G")||":"||(SN Id)) [20, p. 29]. With the exception of non-public standalone networks, the SN Id is referred to as the SNN-network-identifier [20, p. 29], as defined in Table 9.12.1.1 of [18, p. 650].

2.4.1.4 AMF

The AMF is a 16 bits value [19, p. 73]. The AMF's 16 bits are numbered from 0 to 15, with bit 0 being the most significant and bit 15 being the least significant [19, p. 73]. The bit 0 is called the AMF separation bit [19, p. 73]. For step 1 of 5G-AKA we have in [20, p. 43] that AMF's separation bit set to 1. Bits 1 through 7 are reserved for future standardization [19, p. 73]. While not yet specified for a specific use, bits 1 to 7 must be set to 0 [19, p. 73]. Bits 8 to 15 are reserved for proprietary use [19, p. 73]. Three examples of these proprietary usages can be found in Annex F [19, p. 71]. Like in [69, p. 33] we use the value $(10000000\ 00000000)_2$ for AMF.

2.4.1.5 K

K is a secret key shared between UE and HN [27, p. 3]. The length of K is 16 bytes or 32 bytes [19, p. 29]. We use a random 32-byte value in the implementation for K.

2.4.1.6 3GPP authentication and key generation functions f1, f1*, f2, f3, f4, f5 and f5*

Table 2.3 shows type, inputs and output of 3GPP authentication and key generation functions f1, f1*, f2, f3, f4, f5 and f5*.

Table 2.3: 3GPP authentication and key generation functions f1, f1*, f2, f3, f4, f5 and f5*

Fucntion	Type [19, pp. 23] [21, pp. 9–10]	Inputs [19, pp. 22–25] [17, pp. 7]	Output [19, pp. 11]	Length of Output in bytes [19, pp. 29]
f1	MAC function	K, RAND, SQN and AMF	MAC	8
f2	MAC function	K and RAND	RES and XRES	4 to 16
f3	KDF function	K and RAND	Cipher Key (CK)	16
f4	KDF function	K and RAND	Integrity Key (IK)	16
f5	KDF function	K and RAND	Anonymity Key (AK)	6
f1*	KDF function	K, RAND, SQN and AMF	MAC_S	8
f5*	KDF function	K and RAND	AK in re-synchronisation procedures	6

Two examples of algorithm set for the 3GPP authentication and key generation functions f1, f1*, f2, f3, f4, f5 and f5*, are MILENAGE [21] [17] and TUAK [22].

HMAC-SHA-256 was chosen for the functions f1, f1*, and f2, and ANSI-X9.63-KDF (as described in Section 2.3.2.5) for f2, f3, f4, f5, and f5* in [69]. While these choices satisfy the types of f1 to f5 listed in Table 2.3, the generated outputs do not have the expected lengths. Specifically, the output lengths of f1, f1*, and f2 are 32 bytes, while the output lengths of f3, f4, f5, and f5* are 64 bytes in [69]. Nonetheless, we will adopt the implementation choices of [69] and truncate the output by selecting the rightmost bytes of the desired length as listed in Table 2.3.

2.4.1.7 MAC

2.4.1.7.1 Specification As explained in Section 2.4.1.6, the f1 function is utilized for the computation of MAC. Table 2.3 illustrates that f1 is a MAC function that produces an output with a length of 8.

2.4.1.7.2 Verification and refactoring In [69], HMAC-SHA-256((AMF || SQN || RAND), K) is used for the f1 functions. We made two modifications. First, we extract the 8 right-most bytes of the output of HMAC-SHA-256 to ensure that the length of MAC matches the requirements specified in [19, pp. 29]. Second we change the order of cancatinating AMF, SQN and RAND to (SQN || RAND || AMF) to match the requirements specified in [19, pp. 23].

The function "f1Mac" in Listing 2.37 demonstrates implementation and verification of f1.

Listing 2.37: implementation and verification of f1

```
1  //@ public normal_behavior
2  //@ requires (AMFval!=null)&& (SQNval!=null)&& (RANDval!=null) ;
3  //@ requires (AMFval.length + SQNval.length+ RANDval.length)<=
       Integer.MAX_VALUE ;
4   //@ ensures Mac.isHmacSHA256( \result ,
        Concatination.concat2byteArray(SQNval,Concatination.concat2byteArray
        (RANDval,AMFval)), sharedSecretKeyK );
5  //@ pure
6  public static byte[] f1Mac32ByteOut(byte[] sharedSecretKeyK, byte[] AMFval,
       byte[] SQNval, byte[] RANDval) {
7
8     byte[] s = Concatination.concat2byteArray(SQNval,
          Concatination.concat2byteArray(RANDval, AMFval));
```

```
9       byte[] result32Bytes = Cryptography.macFn(s, sharedSecretKeyK);
10      return result32Bytes;
11
12  }
13
14  //@ public normal_behavior
15  //@ requires (AMFval!=null)&& (SQNval!=null)&& (RANDval!=null) ;
16  //@ requires (AMFval.length + SQNval.length+ RANDval.length)<=
        Integer.MAX_VALUE ;
17  //@ ensures \result.length == 8;
18  // bellow ensures is an implementation choice and its not specified in the
        specification: using HMAC_SHA_256 and then truncating it
19  //@ ensures Arrays.equals(\result,
        return8RightMostByteOf32(f1Mac32ByteOut(sharedSecretKeyK, AMFval,
        SQNval, RANDval)));
20  //@ pure
21  public static byte[] f1Mac(byte[] sharedSecretKeyK, byte[] AMFval, byte[]
        SQNval, byte[] RANDval) {
22
23      byte[] result32Bytes = f1Mac32ByteOut(sharedSecretKeyK, AMFval, SQNval,
            RANDval);
24      byte[] result8Bytes = return8RightMostByteOf32(result32Bytes);
25      return result8Bytes;
26
27  }
```

5G-AKA initially lacked Key Confirmation, despite its importance, as discussed in Section 5.2.2 of [27]. To address this, two proposed solutions were suggested to eliminate the need for Key Confirmation [27, pp. 12]. The first solution involves binding AUTN to SNname by including SNname as an input in f1, which computes MAC. The second solution requires adding a few additional steps to the protocol's end [27, pp. 12].

In Listing 2.38, the function "f1MacBasian" showcases the implementation and verification of the updated version of "f1" proposed in [27, pp. 12].

Listing 2.38: implementation and verification of the updated version of f1

```
1   //@ public normal_behavior
2   //@ requires (AMFval!=null)&& (SQNval!=null)&& (RANDval!=null) &&
        (SNname!=null) ;
3   //@ requires (AMFval.length + SQNval.length+ RANDval.length +
```

```
        SNname.length)<= Integer.MAX_VALUE ;
    // bellow ensures is an implementation choice and its not specified in the
        specification: using HMAC_SHA_256 and then truncate it
//@ ensures Mac.isHmacSHA256( \result ,
    Concatination.concat2byteArray(SQNval,Concatination.concat3byteArray(RANDval,AMFval,SNnam
    sharedSecretKeyK );
    //@ pure
    public static byte[] f1MacBasin32ByteOut(byte[] sharedSecretKeyK, byte[]
        AMFval, byte[] SQNval, byte[] RANDval,
          byte[] SNname) {

       byte[] s = Concatination.concat2byteArray(SQNval,
           Concatination.concat3byteArray(RANDval, AMFval, SNname));
       byte[] result32Bytes = Cryptography.macFn(s, sharedSecretKeyK);
       return result32Bytes;

    }


    //@ public normal_behavior
    //@ requires (AMFval!=null)&& (SQNval!=null)&& (RANDval!=null)&&
        (SNname!=null) ;
    //@ requires (AMFval.length + SQNval.length+ RANDval.length +
        SNname.length)<= Integer.MAX_VALUE ;
    //@ ensures \result.length == 8;
    // bellow ensures is an implementation choice and its not specified in the
        specification: using HMAC_SHA_256 and then truncate it
    //@ ensures Arrays.equals(\result,
        return8RightMostByteOf32(f1MacBasin32ByteOut(sharedSecretKeyK, AMFval,
        SQNval, RANDval, SNname)));
    //@ pure
    public static byte[] f1MACBasin(byte[] sharedSecretKeyK, byte[] AMFval,
        byte[] SQNval, byte[] RANDval,
          byte[] SNname) {

       byte[] result32Bytes = f1MacBasin32ByteOut(sharedSecretKeyK, AMFval,
           SQNval, RANDval, SNname);
       byte[] result8Bytes = return8RightMostByteOf32(result32Bytes);
       return result8Bytes;

```

```
31    }
```

2.4.1.8 XRES

2.4.1.8.1 Specification As explained in Section 2.4.1.6, the f2 function is utilized for the computation of RES and XRES. Table 2.3 illustrates that f2 is a MAC function that produces an output with a length of 4 to 16.

2.4.1.8.2 Verification and refactoring In [69], HMAC-SHA-256 is used for the f2 functions. For f1, we extract the 16 right-most bytes of the output of HMAC-SHA-256(RAND, K). This approach ensures that the length of XRES matches the requirements specified in [19, pp. 29].

In Listing 2.39, the function f1MacBasian showcases the implementation and verification of f2.

Listing 2.39: implementation and verification of f2

```
1  //@ public normal_behavior
2     //@ requires (sharedSecretKeyK!=null);
3     //@ requires (RANDval!=null);
4     //@ ensures \result.length == 16;
5     // bellow ensures is an implementation choice and its not specified in the
          specification: using HMAC_SHA_256 and then truncate it
6     //@ ensures Arrays.equals(\result,
          return16RightMostByteOf32(Cryptography.macFn(
          RANDval,sharedSecretKeyK)));
7     //@ pure
8     public static byte[] f2ResAndXres(byte[] sharedSecretKeyK, byte[] RANDval) {
9
10       byte[] result32Bytes = Cryptography.macFn(RANDval, sharedSecretKeyK);
11       byte[] result16Bytes = Authn.return16RightMostByteOf32(result32Bytes);
12       return result16Bytes;
13
14    }
```

2.4.1.9 CK and AK

2.4.1.9.1 Specification As explained in Section 2.4.1.6, the f3 and f4 function is utilized for the computation of CK and AK respectively. Table 2.3 illustrates that f3 and f4 are KDF functions that produce an output with a length of 16.

2.4.1.9.2 Verification and refactoring In [69], ANSI-X9.63-KDF with the output length of 64 bytes and using SHA-256 as a hash function is used for the f3 and f4 functions. Now for f3 and f4, we extract the 16 right-most bytes of the output of ANSI-X9.63-KDF with the output length of 32 bytes and use SHA-256 as a hash function. This approach ensures that the length of CK and AK matches the requirements specified in [19, pp. 29].

In Listing 2.40, the function f3CkAndf4Ik showcases the implementation and verification of f3 and f4.

Listing 2.40: implementation and verification of f3 and f4

```
1  //@ public normal_behavior
2  //@ requires (a1 != null)&&(a2 != null);
3  //@ requires (a1.length +4 + a2.length)<= Integer.MAX_VALUE ;
4  //@ ensures \result.length == 16;
5  //bellow ensures is an implementation choice and its not specified in the
       specification: using ANSI_KDF and then truncating it
6  //@ ensures Arrays.equals(\result, return16RightMostByteOf32(
       ansiKdf32ByteOut(a1,a2)));
7  //@ pure
8  public static byte[] f3CkAndf4Ik(byte[] a1, byte[] a2) {
9     byte[] res_32 = Authn.ansiKdf32ByteOut(a1, a2);
10    byte[] res_16 = Authn.return16RightMostByteOf32(res_32);
11    return res_16;
12 }
```

2.4.1.10 AK

2.4.1.10.1 Specification As explained in Section 2.4.1.6, the f5 function is utilized for the computation of AK. Table 2.3 illustrates that f5 is a KDF function that produces an output with a length of 6.

2.4.1.10.2 Verification and refactoring In [69], ANSI-X9.63-KDF with the output length of 64 bytes and using SHA-256 as a hash function is used for the f5 function. For f5, we extract the 6 right-most bytes of the output of ANSI-X9.63-KDF with the output length of 32 bytes and use SHA-256 as a hash function. This approach ensures that the length of AK matches the requirements specified in [19, pp. 29].

In Listing 2.41, the function f5Akf5StarAkStar showcases the implementation and verification of f5. like in [69] the same function is used for both f5 and f5*.

In [19, pp. 24–35], the output of both f5 and f5* functions is referred to as AK, as also noted in [17, pp. 29]. In contrast, other sources such as [27, pp. 5] use AK and AK* as the outputs of f5 and f5*, respectively.

Listing 2.41: implementation and verification of f5

```
1  //@ public normal_behavior
2  //@ requires (a1 != null)&&(a2 != null);
3  //@ requires (a1.length +4 + a2.length)<= Integer.MAX_VALUE ;
4  //@ ensures \result.length == 6;
5  //bellow ensures is an implementation choice and its not specified in the
       specification: using ANSI_KDF and then truncate it
6  //@ ensures Arrays.equals(\result, return6RightMostByteOf32(
       ansiKdf32ByteOut(a1,a2)));
7  //@ pure
8  public static byte[] f5Akf5StarAkStar(byte[] a1, byte[] a2) {
9     byte[] AK_32 = Authn.ansiKdf32ByteOut(a1, a2);
10    byte[] AK_16 = Authn.return6RightMostByteOf32(AK_32);
11    return AK_16;
12 }
```

2.4.1.11 KDF functions

There are a few rules that are hold for KDF functions in 5G Core (5GC) [20, p. 191]. In 5G-AKA authentication protocol the KDF functions that are used are K_{AUSF}, (X)RES* and K_{AUSF}. Below, these rules are listed:

- Each input parameter is an octet strings, not arbitrary length bit strings [24, p. 38].
- For an input parameter that is a character string UTF-8 encoding is used [24, p. 39].

- For an input parameter that is a non negative integer, first we encode the parameter in base 2. If this parameter's length is specified in 3GPP specifications, we put zeros to the left of the value representing the parameter in base 2. Otherwise, if the length (number of bits) of base-2 encoding is not a multiple of 8, we add minimum required zeroes to the left of the parameter value in base 2 to make the result have a number of bits that is a multiple of 8 [24, p. 39].
- Each input parameter has a length of no more than 65535 bytes [24, p. 38].
- The length of each input parameter Pi (encoded in octet string) is a two bytes value representing number of bytes of that parameter called Li [24, p. 38].
- The derived key is HMAC-SHA-256 (KEY , S) [24, p. 38].
  - KEY: The key used in KDF
  - S = FC || P0 || L0 || P1 || L1 || P2 || L2 ... || Pn || Ln
    - FC: FC values used in 5G-AKA authentication protocol are in the range 0x69 to 0x76 [20, p. 191].
    - P0...Pn: n+1 octet string inputs
    - L0..Ln: Li ($0 \leq i \leq n$) represents number of octets of Pi in two bytes.

2.4.1.11.1 Converting an integer to bytes In Section 2.4.1.11 we saw that for encoding an integer parameter in KDF function, in case the parameters length is not specified in 3GPP specifications, it may be necessary to add zeros to the left of the parameter value in base 2 to ensure that the resulting value has a number of bits that is a multiple of 8.

To accomplish this task, "intToVariableLengthByteArray" in Listing 2.42 is responsible for converting an integer input to a byte array (with minimum possible length).

Listing 2.42: implementation and verification of Converting an integer to bytes

```
1  //@ public normal_behavior
2  //@ requires n >= 0;
3
4  //@ ensures ((0<=n )&& (n <=255)) ==> \result.length ==1;
5  //@ ensures ((256<=n) && (n <=65535)) ==> \result.length ==2;
6  //@ ensures ((65536<=n) && (n <=16777215)) ==> \result.length ==3;
7  //@ ensures ((n >=16777216)&& (n<= Integer.MAX_VALUE)) ==> \result.length
       ==4;
8
```

```
9     //@ ensures (\result.length ==1) ==> (\result[0] ==
          Integer.valueOf(n).byteValue());
10
11    //@ ensures (\result.length ==2) ==> (\result[0] ==
          Integer.valueOf(n>>>8).byteValue());
12    //@ ensures (\result.length ==2) ==> (\result[1] ==
          Integer.valueOf(n).byteValue());
13
14    //@ ensures (\result.length ==3) ==> (\result[0] ==
          Integer.valueOf(n>>>16).byteValue());
15    //@ ensures (\result.length ==3) ==> (\result[1] ==
          Integer.valueOf(n>>>8).byteValue());
16    //@ ensures (\result.length ==3) ==> (\result[2] ==
          Integer.valueOf(n).byteValue());
17
18    //@ ensures (\result.length ==4) ==> (\result[0] ==
          Integer.valueOf(n>>>24).byteValue());
19    //@ ensures (\result.length ==4) ==> (\result[1] ==
          Integer.valueOf(n>>>16).byteValue());
20    //@ ensures (\result.length ==4) ==> (\result[2] ==
          Integer.valueOf(n>>>8).byteValue());
21    //@ ensures (\result.length ==4) ==> (\result[3] ==
          Integer.valueOf(n).byteValue());
22
23    //@ pure
24    public static byte[] intToVariableLengthByteArray(int n) {
25       byte[] result;
26
27       if ((0 <= n) && (n <= 255)) {
28          result = new byte[1];
29          result[0] = Integer.valueOf(n).byteValue();
30          return result;
31
32       } else if ((256 <= n) && (n <= 65535)) {
33          result = new byte[2];
34          result[0] = Integer.valueOf(n >>> 8).byteValue();
35          result[1] = Integer.valueOf(n).byteValue();
36          return result;
37       } else if ((65536 <= n) && (n <= 16777215)) {
38          result = new byte[3];
```

```
39          result[0] = Integer.valueOf(n >>> 16).byteValue();
40          result[1] = Integer.valueOf(n >>> 8).byteValue();
41          result[2] = Integer.valueOf(n).byteValue();
42          return result;
43       }
44
45       else if ((n >= 16777216)) {
46          result = new byte[4];
47          result[0] = Integer.valueOf(n >>> 24).byteValue();
48          result[1] = Integer.valueOf(n >>> 16).byteValue();
49          result[2] = Integer.valueOf(n >>> 8).byteValue();
50          result[3] = Integer.valueOf(n).byteValue();
51          return result;
52       }
53
54       else {
55          //code to be executed if all the conditions are false
56          return null;
57       }
58
59    }
```

2.4.1.11.2 Converting an integer to two bytes

In Section 2.4.1.11, we learned that the length of each input parameter Pi is represented by a two-byte value called Li.

The function "intToLength2ByteArray" in Listing 2.43 converts an integer input to a byte array with a length of two bytes.

Listing 2.43: implementation and verification of Converting an integer to two bytes

```
1  //@ public normal_behavior
2     //@ ensures (\result.length ==2) ;
3     //@ ensures (\result[0] == Integer.valueOf(n>>>8).byteValue());
4     //@ ensures (\result[1] == Integer.valueOf(n).byteValue());
5     //@ pure
6     public static byte[] intToLength2ByteArray(int n) {
7        byte[] result = new byte[2];
8        result[0] = Integer.valueOf(n >>> 8).byteValue();
9        result[1] = Integer.valueOf(n).byteValue();
10       return result;
```

```
11  }
```

2.4.1.12 (X)RES*

2.4.1.12.1 Specification The inputs and output of the KDF function that computes (X)RES* are listed below [20, p. 192] (see Section 2.4.1.11).

- inputs:
  - SNname
  - RAND
  - (X)RES
  - CK
  - IK
- output: (X)RES* = 128 least significant bits of HMAC-SHA-256 (KEY , S)
  - KEY = CK || IK
  - S = FC || P0 || L0 || P1 || L1 || P2 || L2
    - FC = 0x6B
    - P0 = SNname
    - L0 = length of SNname
    - P1 = RAND
    - L1 = length of RAND: $(00000000\ 00000010)_2$
    - P2 = (X)RES
    - L2 = lenth of (X)RES

2.4.1.12.2 Verification and refactoring The function "computeXresStar" in Listing 2.44 demonstrates the implementation and verification of (X)RES*. Line 11 in Listing 2.32 informs the prover that HMACSHA256 is a mathematical (deterministic) function, meaning that computing HMACSHA256 with the same inputs always results in the same output. This property is used by the prover to prove the post condition on line 32 of Listing 2.44.

Listing 2.44: implementation and verification of (X)RES*

```
1  //@ public normal_behavior
2    //@ requires (SNname!=null)&& (CK!=null)&& (IK!=null) ;
3    //@ requires
         (IntToByteArrayConverter.intToVariableLengthByteArray(0x6B).length +
         SNname.length+
         IntToByteArrayConverter.intToLength2ByteArray(SNname.length).length+
         Rand.length+
         IntToByteArrayConverter.intToLength2ByteArray(Rand.length).length +
         XRES.length+
         IntToByteArrayConverter.intToLength2ByteArray(XRES.length).length) <=
         Integer.MAX_VALUE ;
4    //@ requires (CK.length + IK.length)<= Integer.MAX_VALUE ;
5    //@ ensures Mac.isHmacSHA256( \result
         ,Concatination.concat2byteArray(IntToByteArrayConverter.
         intToVariableLengthByteArray(0x6B)
         ,Concatination.concat2byteArray(Concatination.concat2byteArray
         (SNname,IntToByteArrayConverter.intToLength2ByteArray(SNname.length)) ,
         Concatination.concat2byteArray(Concatination.concat2byteArray(Rand,
         IntToByteArrayConverter.intToLength2ByteArray(Rand.length)),
         Concatination.concat2byteArray(XRES,
         IntToByteArrayConverter.intToLength2ByteArray(XRES.length))))) ,
         Concatination.concat2byteArray(CK , IK) );
6    //@ pure
7    public static byte[] computeXresStar256(byte[] SNname, byte[] Rand, byte[]
         XRES, byte[] CK, byte[] IK) {
8
9       byte[] FC = IntToByteArrayConverter.intToVariableLengthByteArray(0x6B);
10      byte[] P0 = SNname;
11      byte[] L0 = IntToByteArrayConverter.intToLength2ByteArray(SNname.length);
12      byte[] P1 = Rand;
13      byte[] L1 = IntToByteArrayConverter.intToLength2ByteArray(Rand.length);
14      byte[] P2 = XRES;
15      byte[] L2 = IntToByteArrayConverter.intToLength2ByteArray(XRES.length);
16
17      byte[] s = Concatination.concat2byteArray(FC,
18           Concatination.concat2byteArray(Concatination.concat2byteArray(P0,
                L0), Concatination.concat2byteArray(
19                Concatination.concat2byteArray(P1, L1),
                      Concatination.concat2byteArray(P2, L2))));
```

```
20        byte[] key = Concatination.concat2byteArray(CK, IK);
21
22        byte[] result = Cryptography.macFn(s, key);
23        return result;
24    }
25
26
27  //@ public normal_behavior
28    //@ requires (SNname!=null)&& (CK!=null)&& (IK!=null) ;
29    //@ requires
          (IntToByteArrayConverter.intToVariableLengthByteArray(0x6B).length +
          SNname.length+
          IntToByteArrayConverter.intToLength2ByteArray(SNname.length).length+
          Rand.length+
          IntToByteArrayConverter.intToLength2ByteArray(Rand.length).length +
          XRES.length+
          IntToByteArrayConverter.intToLength2ByteArray(XRES.length).length) <=
          Integer.MAX_VALUE ;
30    //@ requires (CK.length + IK.length)<= Integer.MAX_VALUE ;
31    //@ ensures \result.length == 16;
32    //@ ensures Arrays.equals(\result, return16RightMostByteOf32(
          computeXresStar256(SNname , Rand , XRES, CK , IK)));
33    //@ pure
34    public static byte[] computeXresStar(byte[] SNname, byte[] Rand, byte[]
          XRES, byte[] CK, byte[] IK) {
35
36        byte[] XRES_STAR_256 = computeXresStar256(SNname, Rand, XRES, CK, IK);
37
38        byte[] XRES_STAR_128 = Authn.return16RightMostByteOf32(XRES_STAR_256);
39        //@ assert Arrays.equals(XRES_STAR_128, Arrays.copyOfRange(XRES_STAR_256,
              16, 32));
40        return XRES_STAR_128;
41
42    }
```

2.4.1.13 K_{AUSF}

2.4.1.13.1 Specification The inputs and output of the KDF function that computes K_{AUSF} are listed below [20, p. 191] (see Section 2.4.1.11).

- inputs:
  - SNname
  - SQN
  - AK
  - CK
  - IK
- output: K_{AUSF} = HMAC-SHA-256 (KEY , S)
  - KEY = CK || IK
  - S = FC || P0 || L0 || P1 || L1
    - FC = 0x6A
    - P0 = SNname
    - L0 = length of SNname
    - P1 = SQN $\oplus$ AK
    - L1 = length of (SQN $\oplus$ AK): $(00000000\ 00000110)_2$

2.4.1.13.2 Verification and refactoring The implementation of the xor function used in [69] is presented in Listing 2.45, which contains a bug at line 10 where "binStr1" was erroneously assigned instead of "binStr2". Such errors can occur when developers copy and paste code from online resources without proper understanding or modification. However, if a function is verified, others can rely on it to perform its intended task. This illustrates how verification instills trust in the implementation of cryptographic algorithms.

Listing 2.45: implementation of xor function in [69]

```
1  static String xorFunction(String binStr1, String binStr2)
2      {
3
4          int binStr1_len = binStr1.length();
5          int binStr2_len = binStr2.length();
6
7          // length balancing by adding zeroes
8          if (binStr1_len > binStr2_len)
9          {
10             binStr1 = lengthBalancerAddingZeroes(binStr2, binStr1_len -
                   binStr2_len);
```

```
        }
        else if (binStr2_len > binStr1_len)
        {
            binStr1 = lengthBalancerAddingZeroes(binStr1, binStr2_len -
                binStr1_len);
        }

        // new balanced length
        int balanced_bin_str_length = Math.max(binStr1_len, binStr2_len);


        String xor_returnVal = "";

        for (int i = 0; i < balanced_bin_str_length; i++)
        {
            if (binStr1.charAt(i) == binStr2.charAt(i))
                xor_returnVal += "0";
            else
                xor_returnVal += "1";
        }
        return xor_returnVal;
    }
```

Listing 2.46 presents the implementation and verification of the function "xorFunction-SameLength", which is used to xor two byte arrays with the same length. Since both SQN and AK have a length of six bytes, only cases where the inputs are of the same length are considered in this function.

Listing 2.46: implementation and verification of xor two byte arrays with same length

```
  //@ public normal_behavior
  //@ requires byteArray1!=null & byteArray2!=null;
  //@ requires (byteArray1.length == byteArray2.length);
  //@ ensures (\result.length ==byteArray1.length);
  //@ ensures (\forall int k; 0 <= k < byteArray1.length; \result[k] ==
      (\old(byteArray1[k]^ byteArray2[k])));
  //To help the prover in the xorFunction post conditions
  //@ ensures (\forall byte[] byteArray1Copy; (byteArray1Copy.length ==
      byteArray1.length)&&(\forall int index; 0<=index< byteArray1Copy.length;
      byteArray1Copy[index]==\old(byteArray1)[index]);
```

```
        Arrays.equals(xorFunctionSameLength(byteArray1, byteArray2),
        xorFunctionSameLength(byteArray1Copy, byteArray2)) );
    //@ ensures (\forall byte[] byteArray2Copy; (byteArray2Copy.length ==
        byteArray2.length)&&(\forall int index; 0<=index< byteArray2Copy.length;
        byteArray2Copy[index]==\old(byteArray2)[index]);
        Arrays.equals(xorFunctionSameLength(byteArray1, byteArray2),
        xorFunctionSameLength(byteArray1, byteArray2Copy)) );

    //@ pure
    public static byte[] xorFunctionSameLength(byte[] byteArray1, byte[]
        byteArray2) {

       int lengthBalance = byteArray1.length;
       byte[] res = new byte[lengthBalance];

       //@ loop_invariant 0 <= i <= lengthBalance;
       //@ loop_invariant \forall int k; 0 <= k < i; res[k] ==
           (byte)(byteArray1[k]^ byteArray2[k]);
       //@ loop_decreases lengthBalance-i;
       for (int i = 0; i < lengthBalance; i++) {
          res[i] = (byte) (byteArray1[i] ^ byteArray2[i]);
       }
       return res;
    }
```

The function "computeKausf" in Listing 2.47 demonstrates the implementation and verification of K_{AUSF}.

Listing 2.47: implementation and verification of K_{AUSF}

```
//@ public normal_behavior
   //@ requires (SNname!=null)&&(SQN!=null)&&(AK!=null)&& (CK!=null)&&
       (IK!=null) ;
   //the SN name is of maximum length of 1020 octets [3GPP TS 24.501 V16.5.1
       p650 section 9.12.1 ]
   //@ requires (1<=SNname.length <= 1020);
   //@ requires (SQN.length == 6);
   //@ requires (AK.length == 6);
   //@ requires (CK.length == 16);
   //@ requires (IK.length == 16);
   //@ requires
```

```
    (IntToByteArrayConverter.intToVariableLengthByteArray(0x6A).length +
    SNname.length+
    IntToByteArrayConverter.intToLength2ByteArray(SNname.length).length+
    xorFunctionSameLength(SQN, AK).length+
    IntToByteArrayConverter.intToLength2ByteArray(xorFunctionSameLength(SQN,
    AK).length).length ) <= Integer.MAX_VALUE ;
//@ ensures javax.crypto.Mac.isHmacSHA256( \result
    ,Concatination.concat2byteArray(IntToByteArrayConverter.
    intToVariableLengthByteArray(0x6A)
    ,Concatination.concat2byteArray(Concatination.concat2byteArray
    (SNname,IntToByteArrayConverter.intToLength2ByteArray(SNname.length)) ,
    Concatination.concat2byteArray(xorFunctionSameLength(SQN, AK),
    IntToByteArrayConverter.intToLength2ByteArray(xorFunctionSameLength(SQN,
    AK).length)))) , Concatination.concat2byteArray(CK , IK) );
//@ ensures \result != null;
//@ pure
public static byte[] computeKausf(byte[] SNname, byte[] SQN, byte[] AK,
    byte[] CK, byte[] IK) {

   byte[] FC = IntToByteArrayConverter.intToVariableLengthByteArray(0x6A);
   byte[] P0 = SNname;
   byte[] L0 = IntToByteArrayConverter.intToLength2ByteArray(SNname.length);
   byte[] P1 = xorFunctionSameLength(SQN, AK);
   byte[] L1 =
       IntToByteArrayConverter.intToLength2ByteArray(xorFunctionSameLength(SQN,
       AK).length);

   byte[] s = Concatination.concat2byteArray(FC, Concatination
         .concat2byteArray(Concatination.concat2byteArray(P0, L0),
            Concatination.concat2byteArray(P1, L1)));
   byte[] key = Concatination.concat2byteArray(CK, IK);

   byte[] result = Cryptography.macFn(s, key);

   return result;
}
```

2.4.1.14 AUTN

2.4.1.14.1 Specification AUTN can be computed using the values of SQN, AK,AMF and MAC as shown on [19, p. 22].

$$\text{AUTN} = (\text{SQN} \oplus \text{AK}) \,||\, \text{AMF} \,||\, \text{MAC}$$

2.4.1.14.2 Verification and refactoring The function "computeAutn" in Listing 2.48 demonstrates the implementation and verification of AUTN.

Listing 2.48: implementation and verification of AUTN

```
1   //@ public normal_behavior
2   //@ requires ( SQN != null) && (AK != null) && (AMF != null) && (MAC !=
        null);
3   //@ requires (SQN.length == 6) && (AK.length == 6) && (AMF.length== 2)
        &&(MAC.length == 8);
4   //@ ensures Arrays.equals(\result,
        Concatination.concat3byteArray(Authn.xorFunctionSameLength(SQN, AK),AMF,
        MAC));
5   //@ ensures \result.length ==16;
6   //@ pure
7   public static byte[] computeAutn(byte[] SQN, byte[] AK, byte[] AMF, byte[]
        MAC) {
8
9      byte[] SQN_xor_AK = Authn.xorFunctionSameLength(SQN, AK);
10
11     byte[] AUTN = Concatination.concat3byteArray(SQN_xor_AK, AMF, MAC);
12
13     return AUTN;
14
15  }
```

2.4.1.15 H(X)RES*

2.4.1.15.1 Specification The inputs and output of the KDF function that computes (X)RES* are listed below [20, p. 192].

- inputs:
  - RAND
  - (X)RES*
- output: H(X)RES* = 128 least significant bits of SHA-256 (S)
  - S = P0 || P1
    - P0 = RAND
    - P1 = (X)RES*

2.4.1.15.2 Verification and Refactoring The function "computeHxresStar" in Listing 2.49 demonstrates the implementation and verification of (X)RES*. Similar to Section 2.4.1.12, here Line 6 in Listing 2.19 informs the prover that SHA256 is a mathematical (deterministic) function. This property is used by the prover to prove the post condition on line 17 of Listing 2.49.

Listing 2.49: implementation and verification of H(X)RES*

```
1   //@ public normal_behavior
2   //@ requires (RAND != null) && (XRES_star != null);
3   //@ requires (RAND.length + XRES_star.length) <= Integer.MAX_VALUE ;
4   //@ ensures \result.length == 32;
5   //@ ensures java.security.MessageDigest.isHash256(
        \old(Concatination.concat2byteArray(RAND , XRES_star)),\result);
6   //@ pure
7   public static byte[] computeHxresStar256(byte[] RAND, byte[] XRES_star) {
8
9      return Cryptography.computeSha256(Concatination.concat2byteArray(RAND,
           XRES_star));
10
11  }
12
13      //@ public normal_behavior
14  //@ requires (RAND != null) && (XRES_star != null);
15  //@ requires (RAND.length + XRES_star.length) <= Integer.MAX_VALUE ;
16  //@ ensures \result.length == 16;
17  //@ ensures Arrays.equals(\result, return16RightMostByteOf32(
        computeHxresStar256(RAND , XRES_star)));
18  //@ pure
```

```
19    public static byte[] computeHxresStar(byte[] RAND, byte[] XRES_star) {
20
21       byte[] HXRES_STAR_256 =
             Cryptography.computeSha256(Concatination.concat2byteArray(RAND,
             XRES_star));
22
23       //return HXRES_STAR_256;
24       byte[] HXRES_STAR_128 = Authn.return16RightMostByteOf32(HXRES_STAR_256);
25
26       return HXRES_STAR_128;
27
28    }
```

2.4.1.16 K_{SEAF}

2.4.1.16.1 Specification The inputs and output of the KDF function that computes K_{SEAF} are listed below [20, p. 191] (see Section 2.4.1.11).

- inputs:
  - SNname
  - SQN
  - AK
  - CK
  - IK
- output: K_{SEAF} = HMAC-SHA-256 (KEY , S)
  - KEY = K_{AUSF}
  - S = FC || P0 || L0
    - FC = 0x6C
    - P0 = SNname
    - L0 = length of SNname

2.4.1.16.2 Verification and refactoring The function "computeKseaf" in Listing 2.50 demonstrates the implementation and verification of K_{SEAF}.

Listing 2.50: implementation and verification of K_{SEAF}

```
1  //@ public normal_behavior
2    //@ requires (SNname!=null) && (KAUSF!=null) ;
3    //@ requires
         (IntToByteArrayConverter.intToVariableLengthByteArray(0x6C).length +
         SNname.length+
         IntToByteArrayConverter.intToLength2ByteArray(SNname.length).length)<=
         Integer.MAX_VALUE ;
4    //@ ensures Mac.isHmacSHA256( \result ,
         Concatination.concat2byteArray(IntToByteArrayConverter.
         intToVariableLengthByteArray(0x6C),Concatination.concat2byteArray
         (SNname,IntToByteArrayConverter.intToLength2ByteArray(SNname.length))),
         KAUSF );
5    //@ pure
6    public static byte[] computeKseaf(byte[] SNname, byte[] KAUSF) {
7
8       byte[] FC = IntToByteArrayConverter.intToVariableLengthByteArray(0x6C);
9       byte[] P0 = SNname;
10      byte[] L0 = IntToByteArrayConverter.intToLength2ByteArray(SNname.length);
11
12      byte[] s = Concatination.concat2byteArray(FC,
            Concatination.concat2byteArray(P0, L0));
13
14      byte[] result = Cryptography.macFn(s, KAUSF);
15      return result;
16   }
```

2.4.1.17 5G SE AV

2.4.1.17.1 Specification To recap the beginning of Section 2.4.1, the 5G SE AV formula is 5G SE AV would be (RAND|| HXRES*|| AUTN) as stated in [20, pp. 43–44].

2.4.1.17.2 Verification and Refactoring The implementation and verification of 5G SE AV can be seen in Listing 2.51.

Listing 2.51: implementation and verification of 5G SE AV

```
1   //@ public normal_behavior
2   //@ requires ( RAND!= null)&&(HXRES_STAR != null)&&(AUTN != null);
3   //@ requires (RAND.length== 16)&&( HXRES_STAR.length== 16)&&(AUTN.length==
        16);
4   //@ ensures \result.length ==(16*3);
5   //@ ensures Arrays.equals(\result, Concatination.concat2byteArray(RAND,
        Concatination.concat2byteArray(HXRES_STAR, AUTN)));
6   //@ pure
7   public static byte[] computeSeAv(byte[] RAND, byte[] HXRES_STAR, byte[]
        AUTN) {
8
9      byte[] AV = Concatination.concat2byteArray(RAND,
           Concatination.concat2byteArray(HXRES_STAR, AUTN));
10
11     return AV;
12
13  }
```

2.4.2 Step 5 of 5G-AKA authentication protocol

During step 5 of the 5G-AKA authentication process, the AUSF sub-entity located in the HN sends the 5G SE AV to the SEAF sub-entity in the SN [20, pp. 44] (see Figure 2.24 and Figure 2.25).

2.4.3 Step 6 of 5G-AKA authentication protocol

At step 6 of the 5G-AKA authentication process, the SN's SEAF sub-entity sends the RAND, AUTN, ABBA and ngKSI to UE [20, pp. 44] (refer to Figure 2.24 and Figure 2.25). The RAND and AUTN sent to UE are part of 5G SE AV, which was previously sent to the SN by UE during step 5 (AUTN was discussed in Section 2.4.1.17).

In the case of a successful authentication at step 12, UE sends K_{SEAF} to the SEAF sub-entity in the SN [20, pp. 44]. Afterward, the SEAF sub-entity computes K_{AMF} from K_{SEAF}, ABBA, and SUPI, following Annex A.7 of [20]. Then, the SEAF sub-entity sends the ngKSI and KAMF to the Access and Mobility Management Function Access and

Mobility Management Function (AMF). The computation of K_{AMF} is not included in the implementation at [69, pp. 39], as it occurs after step 12.

2.4.4 Step 7 of 5G-AKA authentication protocol

2.4.4.1 Check AV freshness

Step Seven involves checking the freshness of AV by assessing AUTN [20, pp. 44]. Figure 2.29 illustrates the inputs and outputs of this check, while the data flow diagram can be found in Figure 2.30 [20, pp. 44] [19, pp. 24] [27, pp. 5].

As depicted in Figure 2.30, K and RAND are passed through f5 to produce AK_{UE}. AK_{UE} is then XORed with the (SQN $\oplus$ AK) component of AUTN to obtain SQN_{HN}. Finally, f1 is applied to $XSQN_{HN}$, RAND,K and AMF (part of AUTN) to calculate XMAC.

For checking AV freshness, Two conditions are checked. First is check whether MAC part of AUTN is equal to XMAC. The second check is whether $SQN_{UE} < XSQN_{HN}$.

To check the freshness of AV, two conditions are evaluated. The first check verifies that the MAC component of AUTN is identical to XMAC. The second check ensures that $SQN_{UE} < XSQN_{HN}$ [27, pp. 4–5]. If both checks pass, the protocol proceeds as described in Section 2.4.4.2. If the first check passes but the second fails, the protocol proceeds as described in Section Section 2.4.4.3. If the first check fails, the protocol proceeds as described in Section 2.4.4.4.

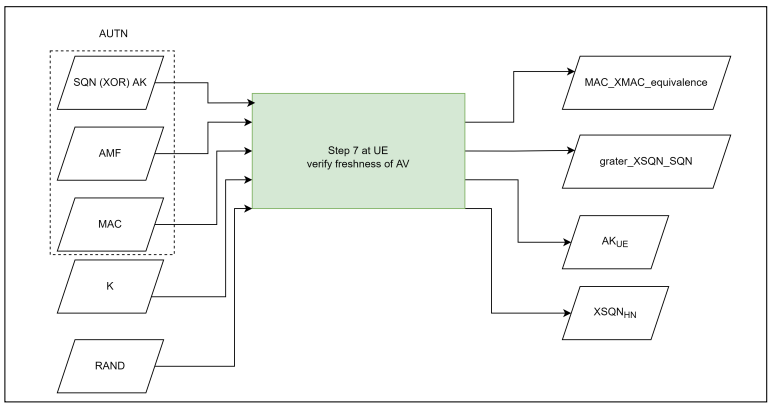


Figure 2.29: Steps Seven Check AV freshness in 5G-AKA Authentication protocol _ Inputs and Outputs

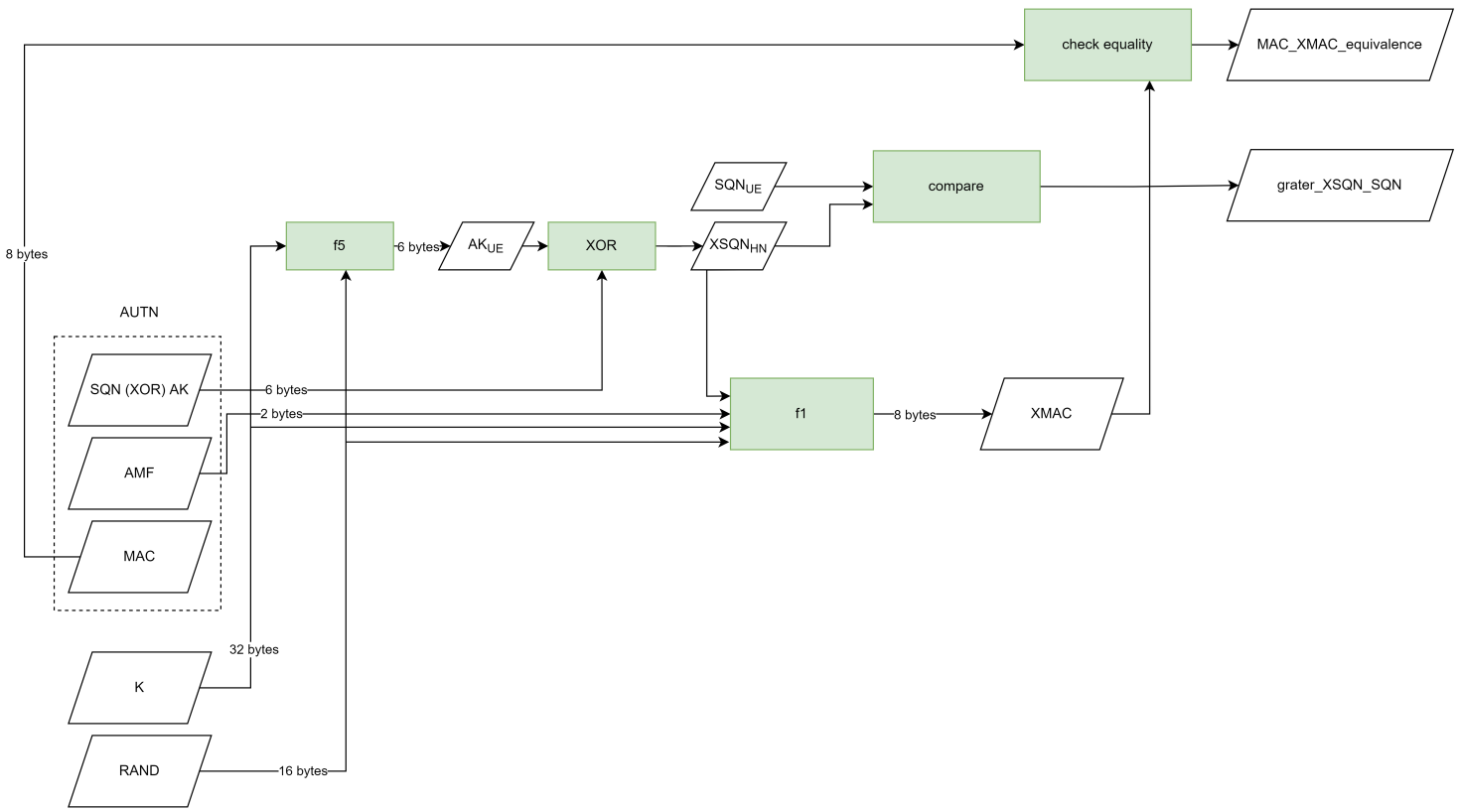


Figure 2.30: Steps Seven Check AV freshness in 5G-AKA Authentication protocol _ Data Flow diagram

2.4.4.2 AV Freshness Passed

Figure 2.31 illustrates the continuation of step seven in the 5G-AKA authentication protocol when AV freshness validation is successful [20, pp. 44] [27, pp. 5].

To begin with, the values of RES, CK and IK are computed, as previously discussed in Section 2.4.1.8 and Section 2.4.1.9. Next, RES* is obtained from RES, as explained in Section 2.4.1.12, followed by the calculation of K_{AUSF} and K_{SEAF}. Finally, the separation bit (MSB) in the AMF field of the AUTN value is checked. If set to 1, RES* is sent to SN [20, pp. 44].

The implementation and verification of checking the MSB bit of given bytes is provided in Listing 2.52.

Listing 2.52: implementation and verification of checking the MSB bit of given byte

```
1  //@ requires x.length>=1;
2  //@ ensures \result ==(((x[0] >> (7)) & 1) == 1)? true:false;
3  public static boolean checkMsbByteArray(byte[] x) {
4     return (((x[0] >> (7)) & 1) == 1) ? true : false;
5  }
```

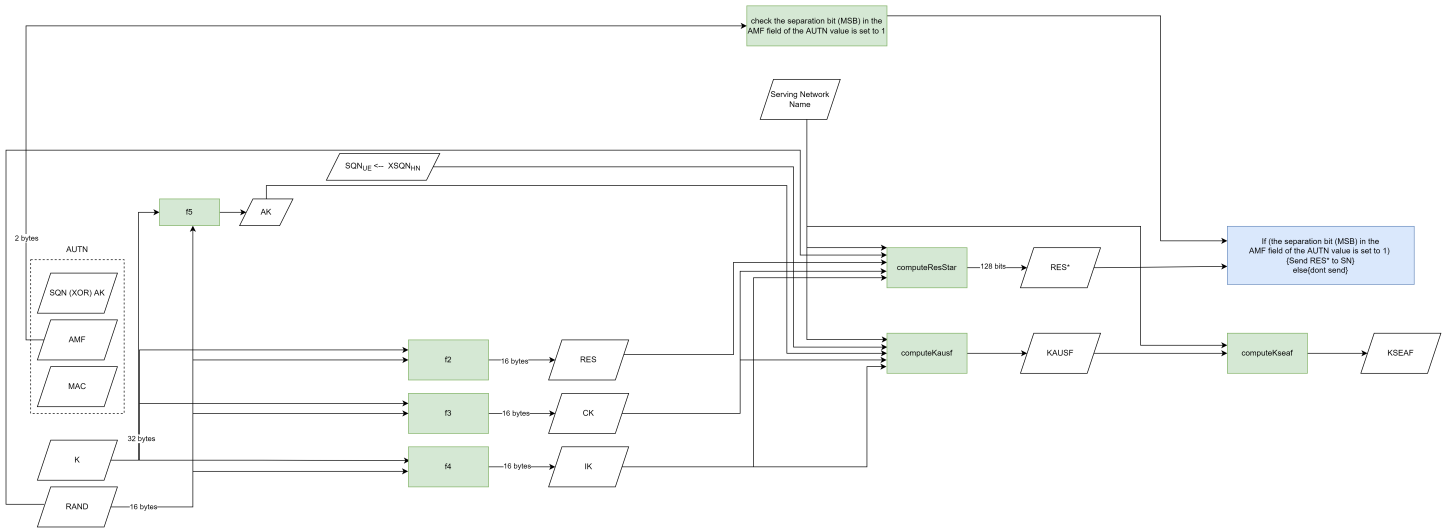


Figure 2.31: Continuation of step seven in the 5G-AKA authentication protocol when AV freshness validation is successful. _ Data Flow diagram

2.4.4.3 Synchronization Failure

Figure 2.33 illustrates the continuation of step seven in the 5G-AKA authentication protocol when AV freshness validation is successful [20, pp. 44] [27, pp. 5] [19, pp. 24–25]. The inputs and output of this part can be found in Figure 2.32.

The first step is to compute MAC_S from RAND, K, and AMF using $f1^*$ (like f1 for $f1^*$ HMAC-SHA-256 was used in [69]). Next, AK^* is computed using $f5^*$, as explained in Section 2.4.1.10. Then, SQN_{UE} and AK^* are XORed to form CONCS. Finally, AUTS is obtained by concatenating CONCS and MAC_S. The resulting AUTS is then sent to SN.

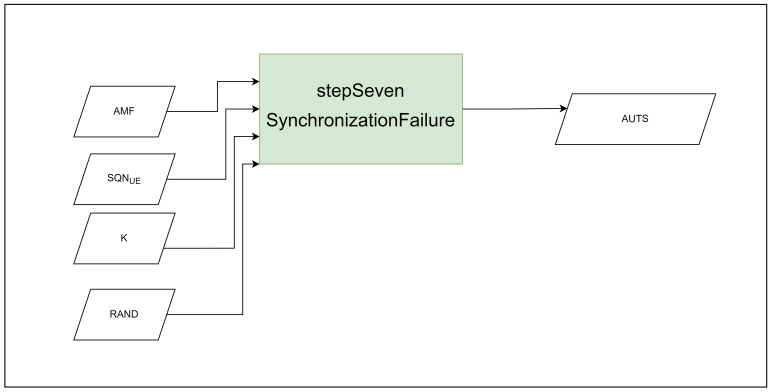


Figure 2.32: Continuation of step seven in the 5G-AKA authentication protocol when AV Synchronization failure occurs _ Inputs and Output

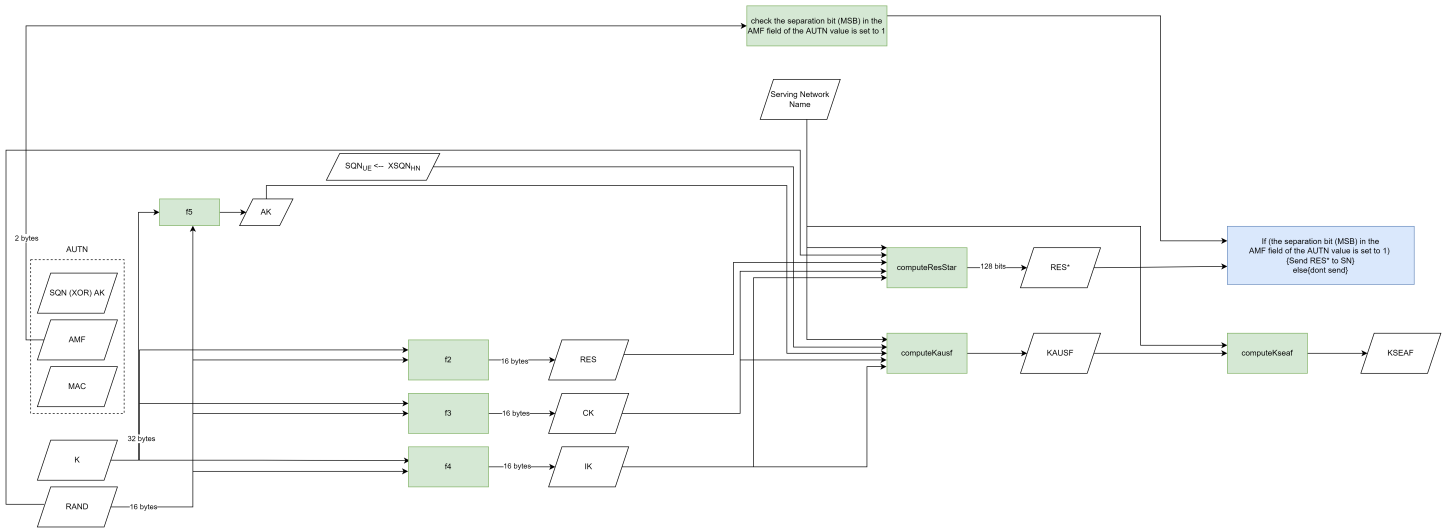


Figure 2.33: Continuation of step seven in the 5G-AKA authentication protocol when AV Synchronization failure occurs _ Data Flow diagram

2.4.4.4 MAC Failure

Figure 2.26 illustrates the case of a MAC failure during the 5G-AKA authentication procedure, the "MAC failure" message is sent to SN [19, pp. 14].

2.4.5 Step 8 of 5G-AKA authentication protocol

During step 8, the subscriber sends RES* that was computed in step 7 to the SEAF subentity of SN [20, pp. 44].

2.4.6 Step 9 of 5G-AKA authentication protocol

Figure 2.34 illustrates the data flow at step 9 of 5G-AKA authentication protocol. First, HRES* is computed from RES* and RAND as discussed in Section 2.4.1.15. Next, HRES* is compared to HXRES* part of 5G SE AV that was received earlier at step 5. If they match, the authentication from the SN's perspective is successful, as stated in [20, pp. 44]. If they do not match, the authentication is deemed unsuccessful [20, pp. 45].

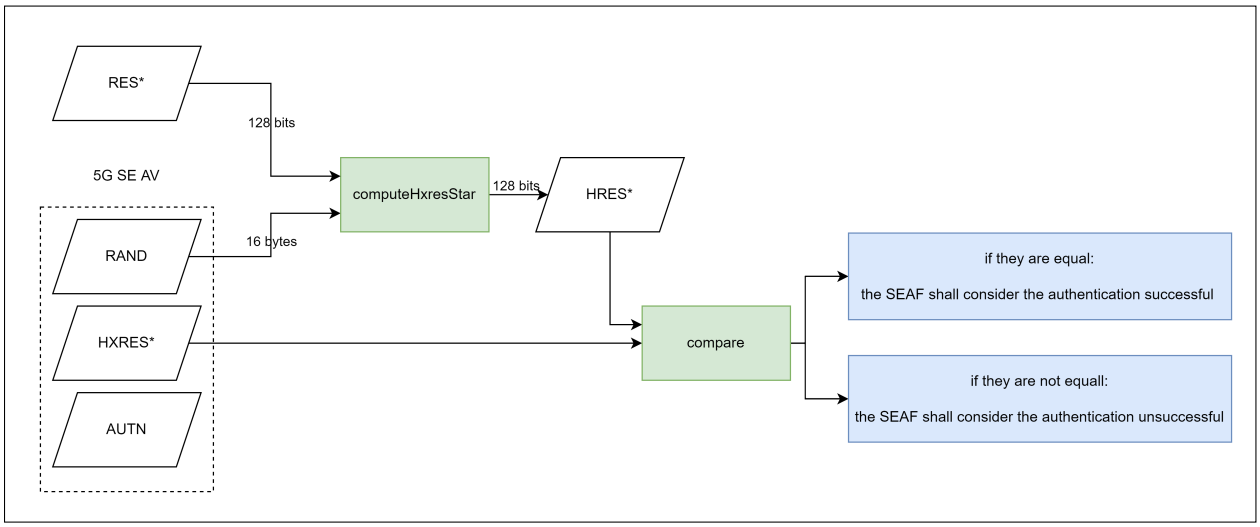


Figure 2.34: Step nine in 5G-AKA Authentication protocol _ Data Flow diagram

2.4.7 Step 10 of 5G-AKA authentication protocol

In step 10, the SEAF sub-entity of SN forwards the RES* it received in step 8, to HN [20, pp. 44].

2.4.8 Step 11 of 5G-AKA authentication protocol

During step 11, RES* (received in step 10) is compared to HRES* (computed in step 1). If they match, the authentication from the HNs perspective is successful, as stated in [20, pp. 44]. If they do not match, the authentication is deemed unsuccessful [20, pp. 44].

2.4.9 Step 12 of 5G-AKA authentication protocol

In step 12 of 5G-AKA authentication protocol, HN tells SN the result of step 11. In case the result is success, K_{SEAF} will be also sent (if authentication is successful, once SN will receive K_{SEAF}, will use it as anchor key in key hierarchy specified in 6.2 of [18]). If the result is success and HN has received SUCI in 5G-AKA initialization protocol, SUCI will be also sent [20, pp. 44].

In the 5G-AKA authentication protocol, step 12 involves HN informing SNN of the result of step 11. If the authentication is successful, K_{SEAF} will also be sent to SN. This key will serve as the anchor key in the key hierarchy specified in section 6.2 of [20]. Additionally, if HN has received SUCI in the 5G-AKA initialization protocol and the result of step 11 is successful, SUCI will also be sent to SN [20, pp. 44].

2.4.10 Step 13

In Section 2.4.1.7, we discussed the first solution proposed in [27, pp. 12] for eliminating the need for Key Confirmation, which was not included in 5G-AKA. In this section, we present the second solution proposed in [27, pp. 12], which involves adding an additional step to the end of the protocol.

At this step, SN sends "(any) message that is MACed with a key derived from K_{SEAF}" to the subscriber [27, pp. 12]. The implementation choice made by [69, pp. 48] for step 13 is HMAC-SHA-256.

Chapter 3

Software Improvements

3.1 Naming

We use the naming that Google-style suggests for Java programming [3].

3.1.1 Camel case

Camel case is a standard naming convention used in programming. There may be more than one correct way to convert an English phrase to a camel case. Google Style recommends the following scheme to convert an English phrase to camel case [3].

Step 1: Write the phrase in plain ASCII and omit any apostrophes

Step 2: Now we have a list of words separated by space. Put space instead of hyphen(s).

Step 3: Write everything in lowercase. Next, for the "UpperCamelCase", change the first character of each word to capital. For the "LowerCamelCase", change the first character of each word to capital with the exception of the first word.

Step 4: Put all words together.

3.1.2 Class names

Nouns or noun phrases are commonly used as class names [3]. "UpperCamelCase" is used for class names [3].

Table 3.1 shows class names in [69] and in "UpperCamelCase" (procedure for "UpperCamelCase", discussed in Section 3.1.1). For instance, the phrase "user equipment init. (abbreviation of initialization)" will become "UserEquipmentInit" and the phrase "user equipment authn. (abbreviation of authentication)" will become "UserEquipmentAuthn".

Table 3.1: Class names

entity	class name	
	in [69]	new name in "UpperCamelCase"
UE	userEquipment	UserEquipment
	ue_5gaka_init_auth	UserEquipmentInit
	ue_5gaka_he_av	UserEquipmentAuthn
SN	servingNetwork	ServingNetwork
HN	homeNetwork	HomeNetwork
	hn_5gaka_init_auth	HomeNetworkInit
	hn_5gaka_he_av	HomeNetworkAuthn

The name of a test class ends with "Test" [3]. If a test class pertains to one class only, the name of the test class is (the name of the class || Test) [3].

3.1.3 Method names

Verbs or verb phrases are commonly used as method names [3]. "LowerCamelCase" is used for method names [3]. JUnit test method names may have underscores (each part should be in "LowerCamelCase") [3].

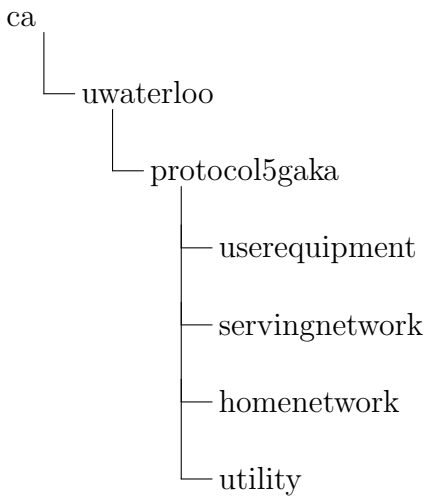
3.1.4 Project name

We use "LowerCamelCase" for the Java project name. converting "project 5G AKA" to "LowerCamelCase" will be "project5gAka".

3.1.5 Package names

Only lowercase letters and digits are used in package names [3] and they cannot start with digits [68]. Google style convention does not allow underscores in package names [3]. In a package name, words that follow each other are simply concatenated for example "deep space" will become "deepspace" [3].

Following the convention discussed above we will use the below hierarchy for package names.



3.2 Code Duplication

It is important to prevent Code Duplication. However they may happen specially if a good review of the code has not been done. Removing code duplication helps in several ways, including:

1. Improved maintainability: With fewer lines of code, the codebase is easier to understand and modify.
2. Reduced development time: It takes longer to make changes when code is duplicated because each instance of the duplicated code must be modified individually.
3. Increased code reusability: By eliminating duplications, it becomes easier to reuse code in different parts of the system, potentially improving overall efficiency.

4. Reduced errors: When code is duplicated, any bugs or errors that existed in the original code will also exist in the duplicate code.

5. Better code organization: As duplications are reduced, the codebase becomes more organized, making it easier to navigate and understand.

In [69] there was code duplication between three main entities as common functions copied and pasted in all three entities (for example there were methods "servingNetwork.sleepFn", "homeNetwork.sleepFn" and "userEquipment.sleepFn"). We remove these duplications by adding the package "utility" and putting common methods there.

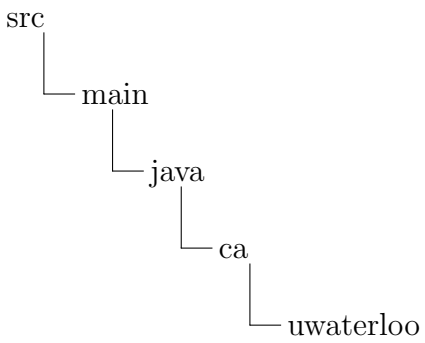
Besides copy-pasted functions across entities, there were code duplications at level of each entity (for instance both methods "servingNetwork.Send_Byte_Arr_to_UE" and "servingNetwork.Send_Byte_Arr_to_HN" had the same body).

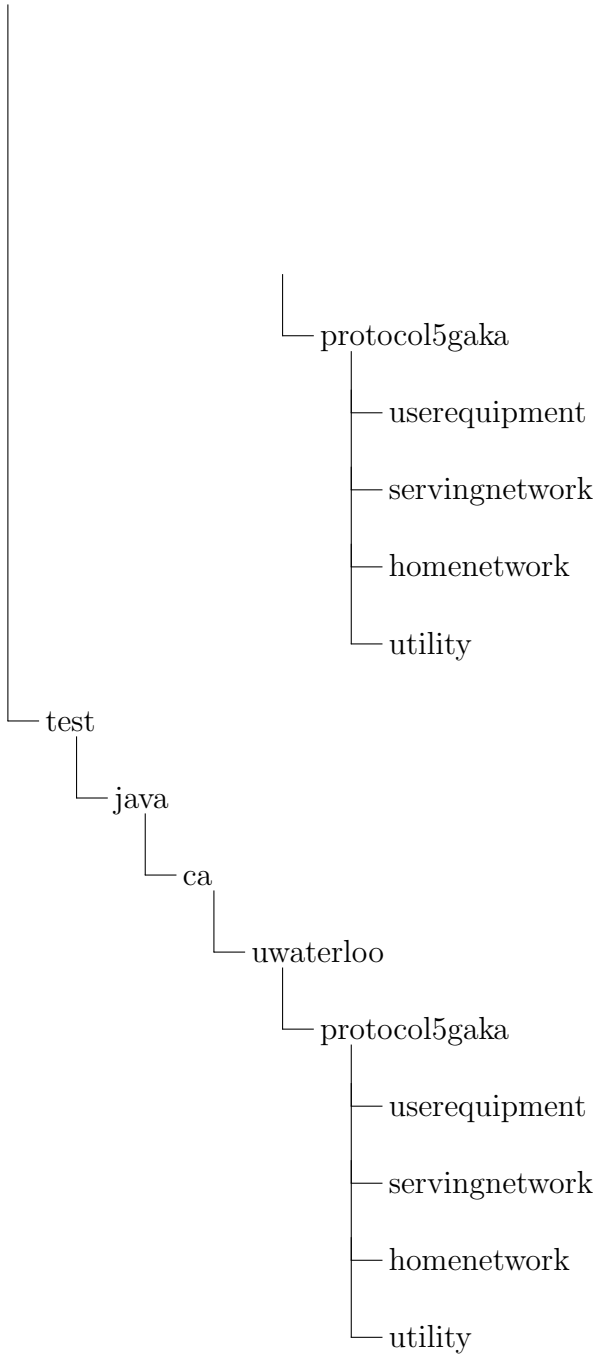
3.3 Structure tests

A single source folder with two subfolders, "src/main" and "src/test", to separate production and test code, is a common and acceptable method of organizing Java projects. This approach is recommended by the Maven project structure guidelines [47], Gradle [50], and Apache Ant [46].

It is a common practice to have the same package structure in both "src/main/java" and "src/test/java". This makes it easier to find the relevant test cases when working with the code.

The structure for main Java files and tests would be:





3.4 Using a build system

3.4.1 Some benefits of a build system like gradle

Using a build system like Gradle has several benefits for managing your project, especially when it comes to managing dependencies, running tests, and generating artifacts like JAR files.

In the case of Gradle, you can define all the dependencies your project needs in a single file called the build.gradle file. This makes it easier to keep track of all the dependencies and ensure that they are up-to-date. Gradle also provides a dependency management system that can automatically download and manage dependencies.

Another significant benefit of using a build system like Gradle is the ability to run tests easily. With Gradle, you can define and run tests as part of the build process, which ensures that your code is tested automatically whenever you build your project. Running all tests at once is very convenient as it saves time and makes it easy to keep track of your test results.

Finally, using a build system like Gradle makes it easy to generate different types of artifacts, like JAR files. By defining the appropriate tasks in your build.gradle file, Gradle can automatically generate the necessary artifacts for your project, including JAR files. This not only saves time but also ensures that the generated artifacts are consistent and follow the same standards across different builds.

In summary, using a build system like Gradle can greatly simplify the management of the project's dependencies, improve the efficiency of the testing process, and automate the generation of artifacts, saving time and effort in the development process.

3.4.2 Gradle Wrapper

The Gradle Wrapper is a script that allows running Gradle builds, regardless of whether or not Gradle is installed on your system. It automatically downloads and installs the correct version of Gradle for your project, ensuring that your build will always be run with the correct version of Gradle.

One of the main benefits of using the Gradle Wrapper is that it helps to ensure the reproducibility of builds. With the Gradle Wrapper, every member of the development team will be using the same version of Gradle, which eliminates the possibility of inconsistencies or errors that can occur when different team members use different versions of Gradle. This is especially important when it comes to building and testing your project in different environments, such as on different operating systems.

3.4.3 Gradle in current project

We defined tasks for producing three executable jar files, one for each of the three main entities (User Equipment, Serving Network, and Home Network). By running "./gradlew build" (Linux/Mac) or "gradlew.bat build" (Windows), the gradle wrapper downloads and installs the correct version of gradle, and three generated jar files become available in the folder "build/libs". The external jar file for Bouncy Castle was specified in the build.gradle file, is also copied in "build/libs".

Chapter 4

Conclusion

In this study, we have demonstrated that static verification is a powerful tool for ensuring the correctness of software implementations. Unlike other techniques such as auditing and testing, which are subject to human error and input space limitations, static verification provides a machine-checkable approach that can guarantee the correctness of an implementation.

Through our investigation of the 5G-AKA protocol, we have shown that static verification can be used to automatically verify the correctness of an implementation. By utilizing the JML specification language and the OpenJML verifier, we were able to identify several flaws and bugs in the initial implementation and make necessary changes to ensure that it met the specifications.

Our study serves as a proof by construction that an implementation of the 5G-AKA specification can be automatically verified using computer-aided cryptography tools. We believe that our findings highlight the importance of using these tools to ensure the correctness of cryptographic protocols, and serve as a case study for writing specifications in JML language from multiple documents written in English.

In conclusion, static verification offers a machine-checkable approach for ensuring the correctness of software implementations. While it may be more expensive than other techniques, it provides a guarantee of correctness that cannot be achieved through other methods. We hope that our study will encourage further research in this area and inspire others to use computer-aided cryptography tools to improve the security and reliability of cryptographic protocols.

Chapter 5

Future Work and Improvements

Several potential directions for future work are worth exploring. One possible path is to investigate the use of other static verification tools, such as the Key tool, to verify JML annotations. Comparing the effectiveness and complexity of different tools could help identify areas where these tools can be improved and optimized for handling JML annotations. This exploration could also provide valuable insights into the limitations and trade-offs of different static verification tools, ultimately contributing to the development of more accurate and reliable cryptographic protocols.

Another promising avenue is to explore the implementation of algorithm sets like MILENAGE and TUAK, which are used for 3GPP authentication and key generation functions. By developing an implementation of these algorithm sets and their corresponding specifications, we can verify their correctness using static verification tools like OpenJML.

In addition, we could also explore the use of pluggable type systems to enforce the properties identified in this study. To this end, we could explore the work presented in [54] and other relevant literature.

Lastly, this study has generated additional JML annotations that could be incorporated into the OpenJML project. Incorporating these annotations would contribute to the development of more comprehensive and accurate Java library specifications.

By pursuing these future directions, we can deepen our understanding of the effectiveness and limitations of static verification tools and further advance the development of secure cryptographic protocols.

References

[1] BouncyCastleProvider (Bouncy Castle Cryptography 1.68 API Specification). https://javadoc.io/static/org.bouncycastle/bcprov-jdk15on/1.68/org/bouncycastle/jce/provider/BouncyCastleProvider.html. [Online; accessed January 06, 2023].

[2] ECGenParameterSpec (Java Security Spec). https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/security/spec/ECGenParameterSpec.html#ECGenParameterSpec(java.lang.String). [Online; accessed January 04, 2023].

[3] Google Java Style Guide. https://google.github.io/styleguide/javaguide.html#s5-naming. [Online; accessed March 20, 2023].

[4] Java Security API: MessageDigest Class. https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/security/MessageDigest.html#getInstance(java.lang.String).

[5] JEP 324: Key Agreement with Curve25519 and Curve448. https://openjdk.org/jeps/324. [Online; accessed January 04, 2023].

[6] JML reference Manual. https://www.cs.ucf.edu/~leavens/JML/refman/jmlrefman.pdf.

[7] jmldoc - The JML documentation tool. https://www.cs.ucf.edu/~leavens/JML2/org/jmlspecs/jmldoc/package.html.

[8] Key (Java Security). https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/security/Key.html. [Online; accessed January 05, 2023].

[9] KeyAgreement (Java Cryptography). https://docs.oracle.com/en/java/javase/14/docs/api/java.base/javax/crypto/KeyAgreement.html#getInstance(java.lang.String). [Online; accessed January 05, 2023].

[10] KeyAgreement (Java Cryptography). https://docs.oracle.com/en/java/javase/14/docs/api/java.base/javax/crypto/KeyAgreement.html#init(java.security.Key). [Online; accessed January 06, 2023].

[11] KeyAgreement (Java Cryptography). https://docs.oracle.com/en/java/javase/14/docs/api/java.base/javax/crypto/KeyAgreement.html#doPhase(java.security.Key,boolean). [Online; accessed January 06, 2023].

[12] NamedParameterSpec (Java Security Spec). https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/security/spec/NamedParameterSpec.html#NamedParameterSpec(java.lang.String). [Online; accessed January 04, 2023].

[13] OpenJML. https://www.openjml.org/. [Online; accessed March 16, 2023].

[14] SECP256R1 Support - Hyperledger Besu. https://wiki.hyperledger.org/display/BESU/SECP256R1+Support#:~:text=For%20the%20secp256r1%20curve%2C%20the,bit%20integer%20(~%2033%20bytes). [Online; accessed January 04, 2023].

[15] Standard Names Spec (Java Security). https://docs.oracle.com/en/java/javase/14/docs/specs/security/standard-names.html#keypairgenerator-algorithms. [Online; accessed January 04, 2023].

[16] Standard Names Spec (Java Security). https://docs.oracle.com/en/java/javase/14/docs/specs/security/standard-names.html#keyagreement-algorithms. [Online; accessed January 05, 2023].

[17] 3GPP. *Specification of the MILENAGE algorithm set: An example algorithm set for the 3GPP authentication and key generation functions f1, f1*, f2, f3, f4, f5 and f5*; Document 2: Algorithm specification (TS 135 206)*, 08 2014. V14.0.0.

[18] 3GPP. *5G; Non-Access-Stratum (NAS) protocol for 5G System (5GS); Stage 3 (TS 24 501)*, 08 2020. V16.5.1.

[19] 3GPP. *Digital cellular telecommunications system (Phase 2+) (GSM); Universal Mobile Telecommunications System (UMTS); 3G security; Security architecture (TS 133 102)*, 08 2020. V16.0.0.

[20] 3GPP. *Security architecture and procedures for 5G System (TS 133 501)*, 08 2020. V16.3.0.

[21] 3GPP. *Specification of the MILENAGE algorithm set: an example algorithm set for the 3GPP authentication and key generation functions f1, f1*, f2, f3, f4, f5 and f5*; Document 5: Summary and results of design and evaluation (TS 135 909)*, 08 2020. V16.0.0.

[22] 3GPP. *Specification of the TUAK algorithm set: A second example algorithm set for the 3GPP authentication and key generation functions f1, f1*, f2, f3, f4, f5 and f5*; Document 1: Algorithm specification (TS 135 231)*, 08 2020. V16.0.0.

[23] 3GPP. *Technical Specification Group Core Network and Terminals; Numbering, addressing and identification (TS 23.003)*, 03 2020. V16.2.0.

[24] 3GPP. *Digital cellular telecommunications system (Phase 2+) (GSM); Universal Mobile Telecommunications System (UMTS); LTE; 5G; Generic Authentication Architecture (GAA); Generic Bootstrapping Architecture (GBA) (TS 133 220)*, 01 2021. V16.3.0.

[25] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H Schmitt, and Mattias Ulbrich. Deductive software verification–the key book. *Lecture notes in computer science*, 10001, 2016.

[26] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. SoK: Computer-Aided Cryptography. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 777–795. IEEE, 2021.

[27] David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirovic, Ralf Sasse, and Vincent Stettler. A formal analysis of 5G authentication. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 1383–1396, 2018.

[28] Benny. ECDSA signature generation using secp256r1. https://stackoverflow.com/q/25261823, November 2014. [Online; accessed January 04, 2023].

[29] Daniel J Bernstein. Curve25519: new Diffie-Hellman speed records. In *International Workshop on Public Key Cryptography*, pages 207–228. Springer, 2006.

[30] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and

applications. *International journal on software tools for technology transfer*, 7(3):212–232, 2005.

[31] Yoonsik Cheon and Gary Leavens. A runtime assertion checker for the Java Modeling Language (JML). 2002.

[32] Oracle Corporation. Java Cipher Algorithms. https://docs.oracle.com/en/java/javase/14/docs/specs/security/standard-names.html#cipher-algorithm-names, 2023. [Online; accessed March 20, 2023].

[33] Oracle Corporation. Java Cipher Init. https://docs.oracle.com/en/java/javase/14/docs/api/java.base/javax/crypto/Cipher.html#init(int,java.security.Key,java.security.spec.AlgorithmParameterSpec), 2023. [Online; accessed March 20, 2023].

[34] Oracle Corporation. Java Cipher Instance. https://docs.oracle.com/en/java/javase/14/docs/api/java.base/javax/crypto/Cipher.html#getInstance(java.lang.String), 2023. [Online; accessed March 20, 2023].

[35] Oracle Corporation. Java IVParameterSpec. https://docs.oracle.com/en/java/javase/14/docs/api/java.base/javax/crypto/spec/IvParameterSpec.html, 2023. [Online; accessed March 20, 2023].

[36] Oracle Corporation. Java Mac Algorithms. https://docs.oracle.com/en/java/javase/14/docs/specs/security/standard-names.html#mac-algorithms, 2023. [Online; accessed March 20, 2023].

[37] Oracle Corporation. Java Mac DoFinal. https://docs.oracle.com/en/java/javase/14/docs/api/java.base/javax/crypto/Mac.html#doFinal(byte[]), 2023. [Online; accessed March 20, 2023].

[38] Oracle Corporation. Java Mac GetInstance. https://docs.oracle.com/en/java/javase/14/docs/api/java.base/javax/crypto/Mac.html#getInstance(java.lang.String), 2023. [Online; accessed March 20, 2023].

[39] Oracle Corporation. Java Mac Init. https://docs.oracle.com/en/java/javase/14/docs/api/java.base/javax/crypto/Mac.html#init(java.security.Key), 2023. [Online; accessed March 20, 2023].

[40] Oracle Corporation. Java Random NextBytes. https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/Random.html#nextBytes(byte[]), 2023. [Online; accessed March 20, 2023].

[41] Oracle Corporation. Java Security API: MessageDigest Algorithms. https://docs.oracle.com/en/java/javase/14/docs/specs/security/standard-names.html#messagedigest-algorithms, 2023. [Online; accessed March 20, 2023].

[42] Oracle Corporation. Java Security Provider. https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/security/Provider.html, 2023. [Online; accessed March 20, 2023].

[43] Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 470–485. IEEE, 2016.

[44] Stijn de Gouw, Frank S de Boer, Richard Bubel, Reiner Hähnle, Jurriaan Rot, and Dominic Steinhöfel. Verifying OpenJDKs sort method for generic collections. *Journal of Automated Reasoning*, 62:93–126, 2019.

[45] David L Detlefs, K Rustan M Leino, Greg Nelson, and James B Saxe. *Extended static checking*. Digital Equipment Corporation Systems Research Center [SRC], 1998.

[46] Apache Software Foundation. Directory Structure for Tests in Apache Ant. https://ant.apache.org/easyant/history/trunk/ref/Directorystructure.html. [Online; accessed January 30, 2023].

[47] Apache Software Foundation. Maven – Introduction to the Standard Directory Layout. https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html, 2023. [Online; accessed January 30, 2023].

[48] GeeksforGeeks. XOR Two Binary Strings of Unequal Lengths. https://www.geeksforgeeks.org/xor-two-binary-strings-of-unequal-lengths/, December 2019. [Online; accessed March 20, 2023].

[49] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1994.

[50] Gradle. Organizing Gradle Projects. https://docs.gradle.org/current/userguide/organizing_gradle_projects.html, 2023. [Online; accessed January 30, 2023].

[51] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA formal methods symposium*, pages 41–55. Springer, 2011.

[52] Donald Knuth. *The TEXbook*. Addison-Wesley, Reading, Massachusetts, 1986.

[53] Leslie Lamport. *LATEX — A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, second edition, 1994.

[54] F. Lanzinger, A. Weigl, M. Ulbrich, and W. Dietl. Scalability and Precision by Combining Expressive TypeSystems and Deductive Verification. *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*.

[55] Gary T. Leavens. Larch/C++, an interface specification language for C++. Technical report, Technical report, Iowa State University, Ames, Iowa 50011 USA, 1997.

[56] Gary T. Leavens, Elodie Albert, Alex Baker, John Boyland, Yoonsik Cheon, K. Rustan M. Leino, and Erik Poll. Java Modeling Language. https://www.cs.ucf.edu/~leavens/JML/index.shtml, 2023. [Online; accessed March 16, 2023].

[57] Gary T. Leavens, Adam Baker, and Cyril Ruby. JML Reference Manual. https://www.cs.ucf.edu/~leavens/JML-release/docs/JMLReferenceManual.pdf, 2021. [Online; accessed March 20, 2023].

[58] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: a Java modeling language. In *Formal Underpinnings of Java Workshop (at OOPSLA98)*, pages 404–420. Citeseer, 1998.

[59] Gary T. Leavens and Yoonsik Cheon. Design by Contract with JML, 2006.

[60] Gary T. Leavens, David R Cok, and Amirfarhad Nilizadeh. Further lessons from the JML project. In *The Logic of Software. A Tasting Menu of Formal Methods*, pages 313–349. Springer, 2022.

[61] K. Rustan M. Leino. Dafny Ref Manual. https://dafny.org/dafny/DafnyRef/DafnyRef.html, 2021. [Online; accessed March 20, 2023].

[62] Tiziana Margaria and Bernhard Steffen. *Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 2008.

[63] Fonenantsoa Maurica, David R Cok, and Julien Signoles. Runtime assertion checking and static verification: collaborative partners. In *International Symposium on Leveraging Applications of Formal Methods*, pages 75–91. Springer, 2018.

[64] Microsoft Corporation. HMACSHA256 Class (System.Security.Cryptography Namespace). https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.hmacsha256?view=net-7.0. [Online; accessed January 09, 2023].

[65] OpenJML. OpenJML GitHub page. https://github.com/OpenJML. [Online; accessed March 18, 2023].

[66] OpenJML. OpenJML User Guide. https://www.openjml.org/documentation/OpenJMLUserGuide.pdf, 2022. [Online; accessed March 16, 2023].

[67] Oracle. Java Security API: Key Interface. https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/security/Key.html#getEncoded(), 2020. [Online; accessed March 20, 2023].

[68] Oracle. Naming a Package. https://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html, 2023. [Online; accessed March 20, 2023].

[69] Rahul Punchhi. An Implementation of 5G-AKA and a Usability Analysis of OpenLDAP Access Control Lists (ACLs). Master's thesis, University of Waterloo, 2021.

[70] Sean Turner, D Brown, Kelvin Yiu, Russell Housley, and Tim Polk. Elliptic Curve Cryptography Subject Public Key Information. Technical report, 2009.

[71] Yuelei Xiao and Shan Gao. Formal Verification and Analysis of 5G AKA Protocol Using Mixed Strand Space Model. *Electronics*, 11(9):1333, 2022.

[72] Daniel M Zimmerman and Rinkesh Nagmoti. JMLUnit: The next generation. In *Formal Verification of Object-Oriented Software: International Conference, FoVeOOS 2010, Paris, France, June 28-30, 2010, Revised Selected Papers*, pages 183–197. Springer, 2011.