UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Applying security features to GA4GH Phenopackets

Master's Thesis

Submitted to the Faculty of the

Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona

Universitat Politècnica de Catalunya

by

Judit Caballero Moro

In partial fulfilment

of the requirements for the degree of

Master in Cybersecurity

Advisor: Silvia Llorente

Barcelona, July 2022

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

telecos
BCN

Final work sheet

Title of the thesis:	Applying security features to GA4GH Phenopackets
Author	Judit Caballero Moro
Advisor	Silvia Llorente Viejo
Delivery date	01/09/2022
Master's degree	Cybersecurity
Language	English
Key words	data sharing, Phenopackets, phenotype, privacy, security mechanisms

Abstract

Global Alliance for Genomic and Health has developed a standard file format called Phenopacket to improve the exchange of phenotypic information over the network. However, this standard does not implement any security mechanism, which allows an attacker to obtain sensitive information if he gets hold of it.

This project aims to provide security features within the Phenopacket schema to ensure a secure exchange. To achieve this objective, it is necessary to understand the structure of the schema in order to classify which fields need to be protected.

Once the schema has been designed, an investigation is conducted into which technologies are currently the most secure, leading to the implementation of three security mechanisms: digital signature, encryption, and hashing.

To conclude, several verification tests are performed to ensure that both the creation of Phenopacket and the security measures applied have been correctly implemented, confirming that data exchange is possible without revealing any sensitive data.

Acknowledgements

This Master's thesis would not have been possible without the support of several people who have been by my side during this long period and whom I would like to express my deepest gratitude.

On the one hand, I would like to thank my mentor Silvia Llorente not only for her dedication and patience throughout the semester in the face of all the setbacks encountered, but also for helping me through times of doubt and misunderstanding. Thank you for allowing me to freely develop the project and to give me ideas whenever I had none. And lastly, thank you for your understanding when I decided to extend the presentation date, even if this meant spending hours during the summer.

On the other hand, I would like to thank my family who have always shown their confidence in me with all the decisions I have made. To my parents and my brother, thank you for letting me be me because it brought me to this point.

I would also like to thank my friends Toni and Irene for being there during this period full of ups and downs, for motivating me and showing me the light at the end of the tunnel, especially in the last days. Thanks also to my Master’s classmate Enric, together we have shared the experience of doing the Master’s thesis this summer and I want to thank him for those dinners tasting burgers that were so good for us.

And finally, to all the people who have shared with me a part of this journey, thank you for always being there and allowing me to grow both professionally and personally.

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

telecos BCN

Table of Contents

1. Introduction 1
   1.1. The importance of data sharing in healthcare sector 1
      1.1.1. Why it is necessary to apply security 1
   1.2. Objectives 2
   1.3. Methods and procedures 3
   1.4. Work Planning 3
   1.5. Time planning 4
      1.5.1. Deviations from the initial plan 4
   1.6. Summary of products obtained 4
2. State of the art 5
   2.1. Standardizing and Exchanging Patient Phenotypic Data 5
   2.2. Genomics informatics — Phenopackets: A format for phenotypic data exchange 5
   2.3. A computable representation of clinical data for precision medicine 6
3. Phenopackets 7
   3.1. Phenopackets schema 7
   3.2. Data model 8
4. Data classification 13
   4.1. Requirements 13
   4.2. Type of data 14
      4.2.1. PII 14
      4.2.2. PHI 14
   4.3. Levels of classification 15
   4.4. Classification of Phenopacket elements 15
5. Security Methods 17
   5.1. Encryption algorithms 17
      5.1.1. Advanced Encryption Standard (AES) 18
      5.1.2. Elliptic Curve Cryptography (ECC) 19
   5.2. Digital Signatures 19
      5.2.1. Elliptic Curve Digital Signature Algorithm (ECDSA) 20
6. Applying security to GA4GH Phenopackets 21
   6.1. Definition 22
      6.1.1. Protocol Buffers 22

6.2. Security mechanisms 24
6.2.1. Hybrid encryption 25
6.2.2. Digital signature 26
6.2.3. Hash 27
6.3. Development 28
6.3.1. Software requirements 28
6.3.2. Phenopacket schema classes 30
6.3.3. Security Mechanisms classes 36
6.3.4. Additional class and files 42
7. Evaluation 45
7.1. Verification tests 45
7.1.1. Phenopacket creation test 45
7.1.2. Hybrid encryption tests 46
7.1.3. Digital signature tests 51
7.1.4. Hashing tests 52
7.2. Phenotools-validator 53
8. Conclusions and future development 55
Bibliography 57
Appendices 61

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

telecos BCN

List of Figures

FIGURE 1 GANTT CHART OF THE PROJECT 4
FIGURE 2 SOURCE: MEDRXIV, "THE GA4GH PHENOPACKET SCHEMA: A COMPUTABLE REPRESENTATION OF CLINICAL DATA FOR PRECISION MEDICINE," 2021. [ONLINE]. AVAILABLE: HTTPS://DOI.ORG/10.1101/2021.11.27.21266944 8
FIGURE 3 HYBRID ENCRYPTION SCHEME 18
FIGURE 4 DIGITAL SIGNATURE SCHEME 19
FIGURE 5 SCRUM WORKFLOW 21
FIGURE 6 SOURCE CODE OF PHENOPACKETS.PROTO 23
FIGURE 7 PROTOBUF WORKFLOW 24
FIGURE 8 HYBRID ENCRYPTION FLOWCHART 26
FIGURE 9 DIGITAL SIGNATURE FLOWCHART 27
FIGURE 10 PROTOBUF DEPENDENCY IN POM.XML FILE 29
FIGURE 11 PHENOPACKET SCHEMA JAR FILE DEPENDENCY IN POM.XML FILE 29
FIGURE 12 SOURCE CODE OF CREATEONTOLOGYCLASS METHOD 31
FIGURE 13 SOURCE CODE OF CREATETIMEELEMENTAGE METHOD 31
FIGURE 14 SOURCE CODE OF GETAGE METHOD 32
FIGURE 15 SOURCE CODE OF GENERATEINDIVIDUALID METHOD 33
FIGURE 16 SOURCE CODE OF CREATEMEDICALTREATMENT 33
FIGURE 17 SOURCE CODE OF GETMETADATA METHOD 34
FIGURE 18 SOURCE CODE OF GENERATEPHENOPACKETID METHOD 34
FIGURE 19 SOURCE CODE OF VERIFYPHENOPACKET METHOD 35
FIGURE 20 SOURCE CODE OF EXPORTPHENOPACKET METHOD 35
FIGURE 21 PRIVATE METHOD TO ENCRYPT AN ELEMENT 37
FIGURE 22 SOURCE CODE OF HYBRIDENCRYPTION METHOD 38
FIGURE 23 PRIVATE METHOD TO SIGN AN ELEMENT 39
FIGURE 24 SOURCE CODE OF PROTECTWITHDS METHOD 40
FIGURE 25 SOURCE CODE OF COMPUTEHASH METHOD 41
FIGURE 26 SOURCE CODE OF COMPUTEDISEASEHASH METHOD 41
FIGURE 27 SOURCE CODE OF CHECKHASH() METHOD 42
FIGURE 28 FUNCTION TO GET A JSON OBJECT FROM FILE 43
FIGURE 29 EXPECTED ERROR IN PHENOPACKETCREATION() TEST 46
FIGURE 30 TESTS CREATED FOR HYBRID ENCRYPTION 46
FIGURE 31 TESTS CREATED FOR HASHING 52
FIGURE 32 PHENOPACKET HAS BEEN SUCCESSFULLY VALIDATED 54
FIGURE 33 PHENOPACKET SCHEMA DIAGRAM 61
FIGURE 34 PROJECT FOLDER STRUCTURE 62

List of Tables

Table 1 Elements and fields included in the Phenopackets schema 11
Table 2 Medical actions and fields included in the Phenopackets schema 12
Table 3 Cryptography-based mechanisms 18
Table 4 Security technologies to be applied in the Phenopacket schema 25
Table 5 Test to verify Age encryption 47
Table 6 Test to verify Age decryption 47
Table 7 Test to verify CreatedBy field encryption and decryption 48
Table 8 Test to verify MetaData encryption and decryption 49
Table 9 Test to verify storage and decryption of Phenopacket elements 50
Table 10 Test to verify digital signature feature 51
Table 11 Test to verify the hashing process 52
Table 12 Test to validate a hash 53

1. Introduction

1.1. The importance of data sharing in healthcare sector

On March 11, 2020, the World Health Organization (WHO) declared the coronavirus as a global pandemic [1] being one of the most relevant events in history, especially in the healthcare sector. Since the beginning of the pandemic, the lack of knowledge about COVID-19 has been a problem when it came to treating those affected, leading to the deaths of millions of people around the world, more than two million in Europe at the time of writing this project [2]. Despite the fact that WHO published some articles to help countries face the disease [3], as well as some tools such as WHO academy [4] to provide knowledge in the process of treating patients, such support was not sufficient to confront the situation, demonstrating that the health sector was unprepared.

One of the greatest challenges faced by researchers has been to find an efficient way to share data and results [5]. Healthcare provides a range of data that can help in the investigation and treatment of illnesses. On the one hand, genomic data is a source of information in the discovery of new disease and treatments. On the other hand, data generated in level-patient, known as biomedical data, allows us to analyze research results, such as how a new treatment affects the patient, and thus improve the response quality against future diseases. Sharing these two data may be essential for limiting the spread of the disease and improving healing.

However, data sharing in health services is still having limited resources for their implementation [6] not only because the cost of maintaining such technologies could be high because of the different formats and databases they integrate, but also because of the privacy laws that must be complied with for its use. Nevertheless, the number of advantages it brings, such as enhancing the development and improvement of research and treatments, as well as improving patient care, will undoubtedly mean that a growing number of organizations will invest in this area.

1.1.1. Why it is necessary to apply security

While the benefits of collecting, using, and transmitting medical data are well known, medical records contain sensitive information that jeopardizes the privacy of individuals. This implies that, in case it is obtained by third parties, it would violate the rights of the individual.

When the pandemic was declared, cyber-attacks also increased, taking advantage of the situation to attack different sectors, including the medical sector [7]. For healthcare, cyber-attacks have significant consequences not only in terms of privacy breaches, but also as an obstacle to ongoing investigations. The technologies used in this sector are systems connected to the network via Wi-Fi, which perform functions such as monitoring a patient's condition, making these systems more vulnerable to cyber-attacks.

By the end of 2020, according to CheckPoint research [8], there was a 45% increase in the number of cyber-attacks against healthcare organizations, becoming one of the most targeted sectors compared to all other industries during the same period.

In the 2021 annual data breach cost report by IBM [9] identified healthcare organizations as the sector with the highest average data breach, increasing by almost 30% over the previous year, being the sixth-most targeted industry in 2021 [10]. Specifically, these

attacks are mainly carried out via ransomware, where the average ransom payment was around $322,000 [11]. But a recent report by Verizon [12] shows that attacks through Web applications and servers are on the rise, as their use is becoming more and more common within the medical environment.

As for this year, attacks targeting the U.S and Europe have intensified. In the case of the United States, 43 health data breaches were reported to the U.S. Department of Health and Human Services in March alone, leading to the exposure or theft of more than 3 million health records, most of them by hacking activities. However, the total amount of data exposed appears to be slightly lower than last year as reported in a HIPAA report [13]. Meanwhile, in Europe, the number of attacks on the medical sector remains stable, for instance in Spain thanks to information reported by AEPD[1] [14] in 2022 there have already been about 100 security breaches involving health data.

It could be concluded that data breaches are one of the most common incidents in this sector, as Personal Health Information (PHI) is more valuable on the black market than any other type of data, which makes medical information an interesting target for threat actors. The 2018 Trustwave Global Security Report [15] showed the different prices on the dark web according to the stolen data being PHI averages $250 at the time of sale.

Since finding a way to share patient data in a secure manner must be a priority, this project will use a standardized format for the exchange of health information called Phenopacket, in which security mechanisms will be applied to ensure a more secure data sharing.

1.2. Objectives

To improve the quality of phenotypic data sharing, this project will aim to study and implement security mechanisms within the file format called Phenopackets.

General objectives:

- Develop a self-designed Phenopacket schema
- Implement security features in the Phenopacket created
- Check the correct use of these Phenopackets

Specific objectives:

- Study of Phenopackets
- Create a customized data model
- Classify the data to apply security
- Research into technologies which offer greater security
- Implement security mechanism
- Perform verification tests

[1] Agencia Española de Protección de Datos

1.3. Methods and procedures

The methods and procedures followed in this project will be divided into two parts: one for research and the other for practical development using an Agile scrum methodology in order to fulfill the objectives listed in section 1.2.

First of all, the study will be focused on what Phenopackets are, how they are structured and their purpose, in order to design a customized data model. On the other hand, it is important to know what data will be treated to apply security measures, therefore, an analysis and classification of the data will be carried out, differentiating them by security levels. To conclude the research part, an investigation will be made of the different available security mechanisms that can then be integrated into the data model.

In parallel, as knowledge of the standard is acquired, the Phenopacket data model to be used will be designed and implemented. Once the most suitable technologies for this project have been analyzed and selected, the next step will be the application of such mechanisms.

Therefore, this second part will be divided into different stages that will include the definition and design of the Phenopackets together with the selection, development and evaluation of their security features using Java as the programming language.

Finally, once a secure Phenopacket schema has been created, it will be checked that it is fully operational through verification tests.

1.4. Work Planning

The methodology described above will be accomplished through a series of tasks that correspond to each of the steps to be followed for the proper development of the work.

Each of the tasks that are part of the project are detailed below:

a. Study of Phenopackets

The main objective of the study will be to learn about this new standard, from its release to the different versions provided as well as its workflow. In addition, the data schema will be analyzed along with the technology used for its application.

b. Design Phenopackets data model

Once the original schema is studied, it will proceed to the implementation of a specific data model for this project.

c. Classification of data in different security levels

Due to the large number and variety of fields offered by the schema, it is necessary to conduct a study to classify which data are considered sensitive to apply security.

d. Research of security mechanisms

Since there are different methods for applying data privacy, the best valued at the time of project development will be analyzed and chosen.

e. Development of Phenopackets schema

The first step will be to define and design the chosen data model, and then apply the security features only to those fields or elements that have been classified as sensitive information.

f. Create and verify secure Phenopackets

To verify if the Phenopacket schema is correctly created, different tests will be developed to check both the Phenopacket creation, and the correct use of the security features applied.

g. Write the final project report

Once all the previous tasks have been completed, the project report will be written, including procedures and conclusions obtained throughout the process.

h. Prepare the project defense

Finally, a project defense will be prepared to demonstrate the knowledge acquired and the work applied during the last months.

1.5. Time planning

The Gantt chart shown in Figure 1 is an indicative illustration of how the project is intended to be developed from March to September.

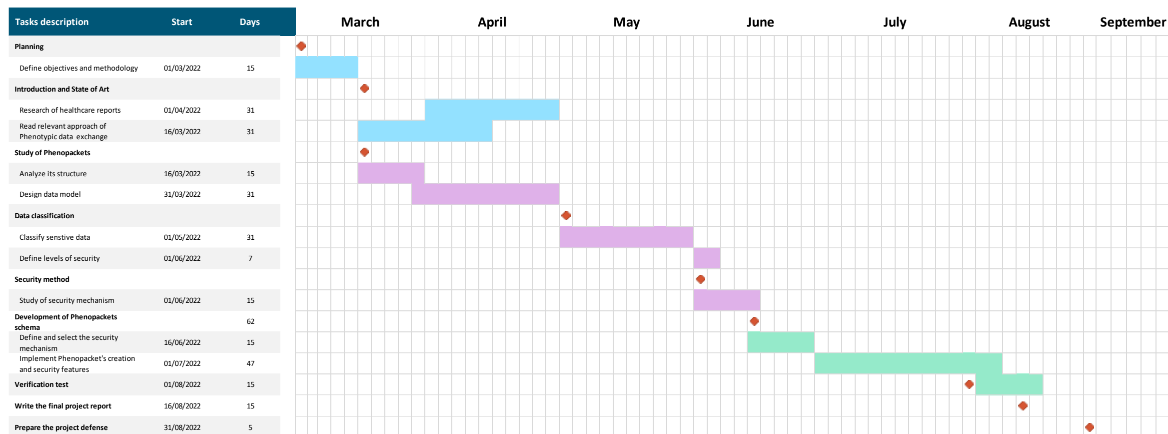


Figure 1 Gantt chart of the project

1.5.1. Deviations from the initial plan

Initially the project was intended to be delivered in July, but due to the scope of the project as well as the lack of knowledge of the topic, it was decided to present it in September.

This lack of knowledge also led to changes in the selection of mechanisms to be implemented, since not all the possible options proposed at the beginning could be finally implemented. The same case applies to the design of the Phenopackets, since it uses a technology unknown to me, which led to several errors at the time of understanding its implementation.

1.6. Summary of products obtained

During the course of this project, a series of deliverables are obtained as the tasks described in the previous section are completed. These products are the following:

- The report, being the present document where all the work done is written in PDF format
- The source code developed for the creation of Phenopackets

2. State of the art

2.1. Standardizing and Exchanging Patient Phenotypic Data

Healthcare professionals and scientists seek to understand diseases by analyzing both a patient's genome and their phenotype. However, there is currently a great difference when it comes to the technology and standards created to deal with genomic data and those created for phenotypic information. While the former treats the genetic information of an organism with various standardized formats and their own databases that define their use and exchange. Phenotypic information, understood as information that provides knowledge about the features, symptoms, and responses of a disease in an individual, suffers from a lack of resources whose sources are widely dispersed and make it difficult to deal with.

Therefore, an international non-profit alliance called the Global Alliance for Genomics and Health (GA4GH), known for creating frameworks and the standards formats that provide a secure way to share genomic data, has developed a new standard they named Phenopackets [16]. A Phenopacket is a standard file format for the exchange of phenotypic information across clinical and research environments. The first version was launched in 2019 focused on describing phenotypic characteristics observed in a subject with an unusual disease, but now has a second version that offers a broader scope where all types of diseases can be addressed.

The objective of Phenopackets is to provide more homogeneity among researchers, medical personnel and clinical analysts who want to operate with phenotypic data using common resources. This new standard provides a better understanding of diseases and treatments, as its interoperability makes it possible to share and analyze data in a faster way. Although the project is still under development, there are some tools that can be used with Phenopackets such as Phenopackets-tools for upgrading from version one to version two or for validation. However, GA4GH is working towards future implementation into Electronic Health Records (EHR) using FHIR[2] framework to encourage its adoption among the healthcare system.

2.2. Genomics informatics — Phenopackets: A format for phenotypic data exchange

The International Organization for Standardization (ISO) has been developed together with GA4GH, the international standard for Phenopackets. In order to collect large samples of phenotypic data from all over the world, standardization is necessary, hence the ISO/DIS 4454 corresponds to the description of Phenopacket as a worldwide standard that provides a computable, machine-readable phenotype data for research and clinical use [17].

On July 7, the document was published [18] in which it defines the entire Phenopackets schema to facilitate the exchange of data between medical communities around the world. The documentation provides what this new format consists of, as well as its components and functionalities, in order to help integrate it within medical systems and thereby, be able to access information from different cases worldwide to improve patient care.

[2] Fast Healthcare Interoperability Resources

2.3. A computable representation of clinical data for precision medicine

At present, medical systems and clinical analysis tools do not contain a lot of phenotypic information. Several approaches have been created over the years to exchange formats for clinical data such as Variant Call Format (VCF), a standard to store genotype information which may then be used by analytical technologies [19]. The first standard dedicated to phenotypic information used rule-based methods in conjunction with clinical data to determine patient cohorts in a wide range of diseases. But there was no approach looking for patient-level exchange until the creation of Phenopacket by GA4GH.

From research environments to medical response, the integration of this standard into medical applications will allow a better response to disease research and provide better patient care, as it contains the set of phenotypic attributes specific to each individual. Subsequently, this information can be used and shared in different scenarios such as clinical management, treatment selection and cohort identification.

In this approach [20], researchers developed the so-called Phenopacket schema to provide that other developers or researchers could design their own Phenopackets encoded in protobuf or JSON formats. As it was mentioned, the schema has two versions, the first one supported two data exchange formats: JSON and YAML, in order to transfer data from providers such as a medical record to a receiver, including web applications. In addition, both versions include the Pedigree standard, which represents pedigree information through PED format to describe a patient's family relationships. Finally, the second version incorporates the Variation Representation Specification (VRS) and VRSATILE framework as well as contains a reference to a VCF file, all together resulting in an improved differential diagnostic process.

There are currently a number of tools that facilitate the management and export of Phenopackets for analytical and data sharing purposes, for example PhenoTips [21]. This software incorporates a user interface that allows the generation of Phenopackets from patients and relatives records, and also includes additional information such as de-identified demographic data or pedigree data.

Meanwhile, in this case, researchers implemented a Java library with command-line applications that validate Phenopackets using a JSON schema. The Phenopacket-validator tool checks that the required elements are present in the schema as well as validates the ontological terms used to define the medical data in the schema.

Despite the different approaches created to date are designed under FAIR[3] principles, none of them apply to secure phenotypic data sharing. For that reason, this project has the main objective to provide security features that allow the Phenopackets schema to be transferred securely across medical environment systems.

[3] Findable, Accessible, Interoperable, and Reusable

3. Phenopackets

The widespread adoption of health records, specifically in this case phenotype information, requires the integration of this data into a secure digital health infrastructure to support patient care. For this purpose, before implementing the various security features to Phenopackets, it is necessary to learn how this new standard works. Although an introduction to this standard has already been given in section 2, it is important to know in depth how it is composed as well as its functionality within the data exchange.

A Phenopacket [22] is a standard providing a phenotypic description of a subject containing a set of components for defining and sharing this information that describes atypical as well as common disease, including cancer. Although the first version was approved at the end of 2019, a new version was created in 2021 which included a better representation of fields related to disease and treatment.

The following sections will explain the latest release, Phenopacket v2.0, in which the structure, functionality and required technology will be described, as well as the data model selected to implement security.

3.1. Phenopackets schema

As previously mentioned, Phenopackets are an open standard for sharing phenotypic information in order to improve the diagnostic process either by investigating new diseases or by providing new treatments for current diseases, resulting in a more efficient and quicker response. For this purpose, it is created to be interoperable and computable to validate and exchange medical information across the different scenarios discussed above.

A Phenopacket [23] links the phenotype specifications of the patient along with information about the patient's disease, allowing sector workers such as clinicians or disease researchers to build more complete disease models. Although it is still evolving, thanks to the personnel in charge of medical data and the different repositories that exist today, the standard can be developed and adopted in the health and research sectors.

To represent the different types of information mentioned above, the schema is formed by several elements and subfields, listed in the Data model, which constitute the Phenopacket structure by means of different levels and multiplicities. The version used has three different levels: required, recommended and optional.

When it comes to multiplicity, it helps to better understand the level of the field. In case the field is mandatory, it will be represented with a multiplicity of 1...1. Then, if the element is recommended or optional it will be represented with 0…1. In other cases, the elements can be presented as a set of them, then the symbol * indicates this property.

In terms of the technology used to enable its application, the phenotype information is exchanged with a format known as PXF[4] files. This format makes it possible to represent the information in both human and machine-interpretable format by encoding it in JSON or YAML to facilitate the transmission of data and its documentation.

For a better understanding of the computational logic of the file format, PXF and in particular Phenopackets, use ontologies to describe most of their elements and thus ensure interoperability between different sources. In this scenario, an ontology is a systematic

[4] Phenotype Exchange Format

formal representation of concepts and categories within the medical area that shows how they relate to each other.

There are different ontologies to represent biomedical information, an interesting ontology is the Human Phenotype Ontology (HPO) [24] that defines patients' phenotypic characteristics, and it is specifically intended for systems biology applications in the context of rare diseases. For cancer knowledge representation, there is the National Cancer institute's Thesaurus (NCIt) while for unit terms found in medical records there is the Units of measurement ontology (UO).

In addition, Phenopackets are defined using the protocol buffer schema that allows them to automatically generate in other languages, including Java, Python and C++. This schema will be discussed later in section 6.1.1.

Finally, once it has been generated and is ready for use, it can be exchanged within the medical framework. The Phenopackets workflow starts with the providers, it can be both medical personnel and researchers as well as laboratories or genomic analysts. The schema works as a common template for capturing data from multiple sources, from the most basic such as clinical notes to the most advanced technologies including mHealth.

In the future, it is intended to use Phenopackets along with some other methods such as FHIR and EHR text-mining, however the second version has already integrated the variation representation specification (VRS). It thus improves its interpretations and provides genotype and phenotype information to the multiple's receivers, including clinical laboratories, diagnostic services, or Electronic Health Records.

The following figure shows the complete Phenopackets workflow.

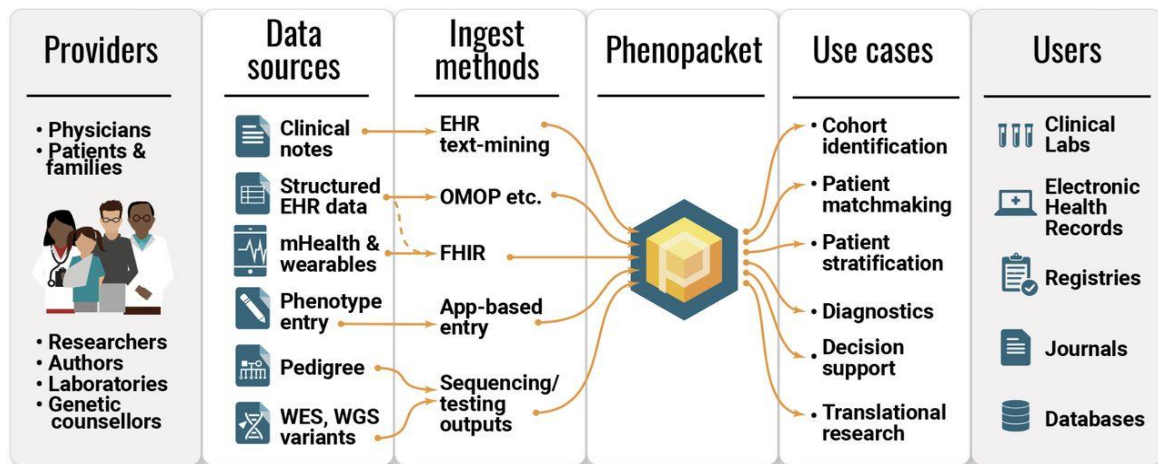


Figure 2 Source: medRxiv, "The GA4GH Phenopacket schema: A computable representation of clinical data for precision medicine," 2021. [Online]. Available: https://doi.org/10.1101/2021.11.27.21266944

3.2. Data model

In section 3.1, it has been defined how the schema created by GA4GH is composed. In this case, the version to be implemented is the second version released with a few differences from the original schema. The Phenopacket data model Appendix shows the set of selected fields that make up the schema and to which security features will be added.

On the one hand, the second version has been chosen due to the addition of elements such as measurements or medical actions as well as new fields to express time or age.

These improvements allow to better narrow down the patient information for all types of diseases, being more accurate than the first version.

On the other hand, the original schema contains many elements and fields that address both to track disease progression and the rapid response (treatment) to these diseases. For this reason, this project has considered to deal only with those fields that make possible the second approach, i.e., only the elements corresponding to phenotype information will be included, along with the diagnosis and appropriate medical actions for the subject, the rest of the elements have been discarded.

Table 1 describes both the primary elements of the Phenopacket schema and the individual blocks which are subfields that compose the previous ones to provide more information. It also specifies the type for each field, defining the element with **[E]**, a time element via **[TE]** and ontology class using **[OC]**.

Field	Type	Description
Id	str	Random ID required to uniquely identify the Phenopacket
Subject	E	Recommended element to represent the patient or proband of the study
• *id*	str	Required identifier being a unique random value for an individual
• *time at last encounter*	TE	Recommended field to represent the age of an individual at last encounter using ISO8601
• *vital status*	Enum	Optional field that includes the status of an individual commonly used in cancer. This field can only have one of the following values: *unknown, alive, and deceased*
• *karyotypic sex*	Enum	Optional field for the chromosomal sex of an individual. The schema includes 10 chromosome types, but only one can be assigned per patient.
Phenotypic features	E	Recommended element listing the phenotypic findings observed in the study
• *type*	OC	Required term describing the phenotypic feature
• *severity*	OC	Optional field to describe the severity of the condition
• *onset*	TE	Optional field representing the time at which a phenotypic feature was noticed or diagnosed.
• *resolution*	TE	Optional field representing the time when the feature resolved or disappeared

• *evidence*	OC	Recommended element to specify how the phenotype was determined. It includes a *code* field to represent the evidence from publication.
Diseases	E	Optional element listing diagnosed or suspected conditions
• *term*	OC	Required term that denotes the disease
• *excluded*	Bool	Boolean value indicating if disease was observed. It is important to check for correct interpretation
• *onset*	TE	Optional field describing the age at which a disease was noticed or diagnosed
• *disease stage*	OC	Optional field describing the development of the disease
• *clinical tnm finding*	OC	Optional field to describe cancer progress
• *primary site*	OC	Optional field to describe the location where the disease was observed
Medical actions	E	Optional element listing different actions performed on the subject, such as procedures or treatments
• *action*	E	Required element being one of the four actions included in the schema: *procedure, treatment, radiation therapy* and *therapeutic regime*
• *treatment target*	OC	Optional field representing the condition or disease that the treatment is for
• *treatment intent*	OC	Optional field to specify the intention of the treatment
• *response to treatment*	OC	Optional field describing how the patient is responding to the treatment
• *adverse events*	OC	Optional field listing adverse effects experienced because of the treatment
• *treatment termination reason*	OC	Optional field indicating the reason why the actions were stopped

Metadata	E	Required element to define the resources as well as ontologies used within the Phenopacket
• *created*	TE	Timestamp element representing the time when Phenopacket was created using ISO8601 UTC timestamp
• *created by*	str	Provides the name of the contributor or program
• *submitted by*	str	Represents the individual who submitted the Phenopacket
• *Phenopacket schema version*	str	The version used for its creation
• *resources*	E	Required element to list external resources referenced in Phenopacket. Includes the following required string subfields: – *id*: identification of the resource – *name*: name of the ontology used – *namespace prefix*: CURIE prefix – *url*: link to the ontology URL – *version*: resource to make the annotation – *iri*: Internalized Resource Identifier
• *updates*	TE	Required timestamp element to represent when the Phenopacket was updated

Table 1 Elements and fields included in the Phenopackets schema

As it was mentioned, `MedicalAction` element requires a field to represent a specific type of medical action. The following table defines each of these actions.

Field	Type	Description
Procedure	E	Element to describe the clinical procedure
• *code*	OC	Required field indicating the clinical procedure
• *body site*	OC	Optional field to specify the location of the body where the procedure was performed
• *performed*	TE	Optional field indicating the time of the procedure
Treatment	E	Agent element being drugs or medicines
• *agent*	OC	Required field to indicate the specific agent

• *route of administration*	OC	Recommended field describing how the agent was administered
• *dose interval*	E	Recommended element listing the dosage of medication. This element includes the following required fields: – *quantity* via *unit* and *value* fields – *schedule frequency* – *interval* at which the dose was administered
• *cumulative dose*	E	Optional element representing the total quantity of the treatment
Radiation therapy	E	Element used for cancer cases
• *modality*	OC	Required field describing the radiation modality
• *body site*	OC	Required field indicating the site where the action was performed
• *dosage*	int	Required field representing the total dose given
• *fractions*	int	Required field corresponding to the radiation dosage divisions
Therapeutic regimen	E	Specify the set of treatments conducted
• *identifier*	OC	Required field to identify the regimen
• *start time*	TE	Recommended field to represent the time when the regimen started
• *end time*	TE	Recommended field to represent the time when the regimen ended
• *status*	str	Required field to specify the status. This field includes four values: *unknown*, *started*, *completed*, and *discontinued*

Table 2 Medical actions and fields included in the Phenopackets schema

Finally, a couple of things should be noted. First of all, not all the fields specified in the original schema for each of these elements are included, only the required, recommended and those that could be of interest for the proposed scenario have been considered. Secondly, to start with a more secure schema, it has been decided to use dates for those elements that allow to choose between age and time, in addition to not including sensitive data fields such as DOB or gender, which has been replaced by the *karyotypic sex* field.

4. Data classification

In the introduction section some statistics were exposed which showed how attractive PHI is for the attackers, since they can exchange records for money. This fact remarks on the importance of implementing advanced security solutions to the medical data, but first it needs to know what type of data it is going to deal with and classify data into categories according to the amount of sensitive information it provides.

This section describes the data classification process followed as well as the different levels of security applied to the data model mentioned in section 3.2 on the basis of which data would be the most sensitive in a data breach attempt.

4.1. Requirements

To achieve better classification, there are several frameworks and legal regulations which define some requirements to classify data depending on the type of information that an individual must recollect or transmit. Considering the context of the project, it is going to introduce two standards whose aims are to protect sensitive data, including health data. These two regulations are HIPAA and GDPR.

On one hand, there is the **Health Insurance Portability and Accountability** (HIPAA) is a US public law (104-191) [25] designed to provide privacy standards to protect electronic health care transactions. The law considers as high risk the compromise of data classified as PHI, so it designed a specific rule called HIPAA Security Rule that demands the implementation of preventive measures as well as classification procedures to ensure the integrity, availability, and privacy of this type of data. This rule also restricts the use and disclosure of all PHI collected, stored, and transmitted by an entity.

On the other hand, the **General Data Protection Regulation** (GDPR) [26] is a European level data protection regulation, this means that any entity that collects, transmits, and uses any information of European citizens must apply its data privacy legislation. Moreover, the GDPR categorizes as special data those depending on the race, ethnic, biometric data, and health data. It also requires classification by levels, as will be shown in section 4.3.

Throughout the project, it has gathered information from both standards, however, being a project carried out within the European Union, it is worth mentioning some of the regulations provided by the GDPR in order to provide a correct classification of the composed fields of the Phenopackets.

Article 4 defines personal data as any information that allows to identify directly or indirectly a natural person such as its name, identification number or information related to their physical, genetic, and mental conditions. Also, online information that may be used to create profiles of them, including IP addresses or cookie identifiers.

In terms of physical and mental health conditions of a subject, the Recital 35 specifies that any number which identifies a patient or information derived from testing or examination of the body, including genetic data and biological samples as well as disease, clinical treatment, and biomedical state, will be considered health data.

Finally, the article 9 details the processing of health data where it is described that the use of this data is allowed as long as it is for diagnostic purposes or in the public interest. Furthermore, the subject must have given his explicit consent for such purposes.

4.2. Type of data

In the previous section it has been differentiated between personal data and health data, this distinction is due to the fact that each of these data can be associated with a type of sensitive personal information. Specifically, these two types are: Personally identifiable Information (PII) and Protected Health Information (PHI).

It should be mentioned that HIPPA considers both types within its framework, while GDPR talks about personal data to address to PII, including health data, however, as data concerning health is treated as special data, it could be considered so-called PHI. For a better understanding, the information included in each type is defined below; Note that in this project, both types have been differentiated as HIPPA does.

4.2.1. PII

Personally Identifiable Information is those data that alone or in combination with other data could allow to identify an individual, in this case it is used outside the healthcare context.

The data that uniquely identifies a subject are names, surnames, residence, and identity number, being the most sensitive one. Then there are other data, commonly available information, that combined between them can reidentify a subject, such as age, gender, date of birth as well as country of birth.

4.2.2. PHI

Protected Health Information or electronically Protected Health Information (ePHI), according to HIPAA law, is any identifiable information used in medical environments as well as communication between healthcare personnel, including research. This implies that any information related to the patient's treatment, results or PII recorded in a medical record will be considered PHI.

The following identifiers are examples of PHI which can allow an individual to be identified:

- Name
- Biometric data
- Medical record numbers
- Medical device serial numbers
- Dates of visits, admission discharge and treatment
- Diagnostic codes
- Entities associates such hospitals, insurance companies, healthcare clearinghouses

In addition, any other sensitive information not included in the list below but is considered a unique identifying characteristic will be considered as Protected Health Information.

Although the combination of different fields within the healthcare scenario could also lead to the re-identification of an individual, it could not be considered as PHI the results of medical tests such as blood sugar readings or heart rate monitoring.

4.3. Levels of classification

To classify data, it is important to consider what is the Phenopacket collecting and what is its level of sensitivity. The schema involves both types of sensitive information explained regarding the patient and medical personnel.

In relation with the sensitivity of the data, different sensitivities lead to different levels of classification, which usually involves four levels: public, internal, confidential, and restricted data [27] [28]. The first two types of data are not considered within the scope, since either the schema does not provide such data, or they have not been included in the designed one. Nevertheless, it is important to know what they are in order to differentiate them.

It is known as public data, information that is available to any individual, being able to perform actions such as storing or distributing this information without any limitation, so the data have no security since they are public values. In this category some PII are included, for example addresses, dates of birth or gender, however, it is not taken into account since this information has been removed from the schema.

In case of internal data, also known as private, includes data associated with internal personnel of the company to whom access is given, such as phone numbers or email addresses. It is also in this category the business information that is required to protect its integrity, e.g., data stored in files. Although this data may not require a high level of security, it is advisable to apply some protection such as password protection for corporate web platforms.

The following two categories are those that include sensitive information about an individual that cannot be disclosed and, therefore, the categories considered in this project to classify the schema:

a. Confidential data: a user must be authorized to obtain the data included in this type. It has a high level of security applied to them since it involves aspects of identity and permissions. It will consider confidential data: social security numbers, cardholder, sensitive documents protected by laws like HIPAA or GDPR and medical and health records as well as biometric identifiers.

b. Restricted data: it is the most sensitive of the data classifications where only a few users could have access. The intrusion of a third-party is punishable by law, for this reason, it has strict security controls such as data encryption. Some examples of restricted data are business proprietary information, protected health information (PHI) and data protected by the state.

4.4. Classification of Phenopacket elements

The previous sections have been useful to differentiate the level of sensitivity of each type of data. The next step is to choose which fields of the schema will be protected as well as the security level to apply. As a result, after analyzing the designed schema, the following decisions are taken.

On the one hand, since the format is created to provide the minimum of identifiable information within their schema, it has only been identified the name of individuals involved in the Phenopacket as well as the age and the karyotypic sex of the subject as PII. On the other hand, Phenopackets identifiers, individual identifiers, treatments, and the different dates found, for example Phenopacket creation, will be identified as PHI. In addition, the

PhenotypicFeature element will be also considered PHI, since it defines different aspects of the medical history of the subject.

Nonetheless, the aforementioned elements will be treated in two categories: once for restricted data being the fields with the highest level of security, and confidential data in which the elements themselves are not a concern but contain fields that may contain more sensitive medical information. For this reason, it is recommended to apply a low level of security to safeguard their integrity. Therefore, the proposed elements for each category are listed below.

Considering **confidential data** as the different information found in a medical record, including sample of disease and medical actions such as treatments, it will be included:

- The PhenotypicFeatures element. Although the element has been categorized as PHI, not all fields may pose a threat, but rather a factor that may allow re-identification. Hence, it is considered to be classified within this category
- The Disease element. The information found in this element is commonly stored in a medical record
- The MedicalAction element. The different actions allowed do not provide enough information on the treatment to re-identify the patient, therefore it is not considered a high risk

As for the **restricted data**, the following fields and elements have been considered as it is data classified as PHI as well as PII:

- All TimeElement blocks that indicate the age of the subject, such as TimeAtLeastEncounter from an individual and onset from diseases
- The field named KaryotypicSex from a subject as it is related to the sex of the patient
- The MetaData element or specifically, the field related to the creator of the Phenopacket. This element has been proposed because it is the only element required in the Phenopacket. It includes information about the date of creation and the names of the creators as well as the resources used to describe the phenotypic information of the patient

Furthermore, both confidential and restricted data have also been proposed because of the thought that the union of any of them could lead to a possible re-identification of the patient.

5. Security Methods

Once the data is properly classified, the next step is to apply security to them. In the scenario of data sharing, and most specifically in Phenopackets schema, the best security features that could be applied to data are technical safeguards. According to the security Rule defined by HIPAA [29], it talks about technical safeguards as all the mechanisms, procedures and policies for its use that protects and manages PHI. Some examples of this mechanism are access control, entity authentication or transmission security. In this case, what it wants to achieve is the secure transmission of the Phenopacket over the network, therefore, in accordance with Article 32 of the GDPR, security will be ensured by end-to-end encryption.

In this section, the current methods for applying security will be explained. The variety of mechanisms to implement on the data is quite wide, however, the mechanisms that will be considered are those that provide confidentiality, integrity, and authentication to the data. Therefore, encryption algorithms will be used to guarantee data confidentiality, while digital signatures will be used to provide authentication.

5.1. Encryption algorithms

This section talks about encryption mechanisms that protect health information with the aim to convert the original data into encoded or unreadable text that is eventually decrypted into plain text.

There are several types of available encryption mechanisms to implement, in this case it will be explained those relying on cryptographic techniques. Therefore, hashing is also included in this section along with symmetric cryptography and asymmetric cryptography, even though a hash is not really an encryption algorithm, it is still useful as a verification method.

The following table explains each of them, as well as their advantages and disadvantages.

Method	Description	Advantages	Disadvantages
Hash	Fixed-length value, numbers, and letters, based on a mathematical function obtained from the contents of a file. The same input will also provide the same hash output.	Tamper resistant. The same hash value cannot be found with other content.	Less flexible than other methods. Can be vulnerable to dictionary attacks.
Symmetric Encryption	The same key is used to both encrypt and decrypt data. For that reason, it is shared among all parties involved in the transmission.	Enables fast data sharing. Requires less resources than asymmetric encryption.	It is considered less secure than asymmetric encryption.

Asymmetric Encryption	Requires two different keys, a public key where anyone can know its value, and a private one that only knows the owner. Then, the public key is used to encrypt the data and the secret key will be used to decrypt it.	More secure than other methods	Reduce the speed of the networks and technologies as well as the transactions involved in communication.

Table 3 Cryptography-based mechanisms

In relation to hash algorithms, several of them have been considered weak because they have been broken by brute force attacks, such as MD5. However, there are other functions that provide secure hash encoding, for example Keccak256, known as a SHA3 hashing algorithm that provides 32-byte hashes and it is used in several blockchain projects, such as Monero [30].

As for encryption algorithms, there are many options to consider depending on the number of keys or the size of the blocks, the most reliable methods to date are explained in the following sections. However, a better approach is using both encryptions known as hybrid encryption in which one-time symmetric key is generated to encrypt the message. After that, the receiver's public key is used to encrypt the previous key and the message.

For a better understanding, the following image shows the hybrid encryption process.

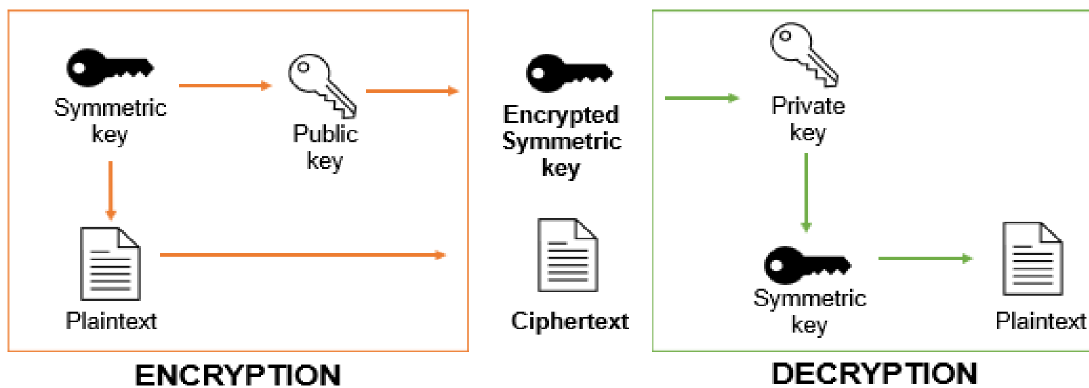


Figure 3 Hybrid encryption scheme

5.1.1. Advanced Encryption Standard (AES)

Advanced Encryption Standard (AES) is one of the most used and trusted encryption algorithms to protect the data. It relies on symmetric key encryption, which encrypts data in a single block with different blocks of 128 bit-sizes and different key sizes that will determine the name of each AES type, for example, AES256 has a key size of 256, but other available AES algorithm are AES128 and AES512.

Another characteristic is that it incorporates different numbers of rounds, which are the processes of changing a plaintext into encrypted data. In the case of AES128, the number of rounds is 10, whereas AES256 has four more, giving a total of 14 rounds.

It is currently considered secure against almost every known attack, making it the most widely deployed algorithm in applications such as data tools market or wireless security.

5.1.2. Elliptic Curve Cryptography (ECC)

Elliptic Curve is a more complex algorithm used in asymmetric encryption since it relies on the representation of a set of points that satisfy a mathematical equation ($y^2 = x^3 + ax + b$). The algorithm includes the elliptic curve parameters and a reference point *P* as a public parameter, then both sender and receiver choose a random value to compute the agreed point resulting in the private parameter *K*.

It has the advantage of making it easy to process the encryption since it uses a shorter key length, but also the mathematical formula that implements makes it harder to undo it. As a comparison, its highest setting with a 512-bit key is comparable to a 15.360-bit RSA key [31].

5.2. Digital Signatures

To provide authentication it will use a signature scheme, specifically the data to protect even the one encrypted, will not be able or decrypted until the signature is validated, providing more security to the schema. According to CISA[5] [32] a digital signature is a mathematical algorithm to sign and verify messages in order to validate the authenticity and integrity of a message.

The signature is linked to a specific individual or entity, so it also provides non-repudiation which means that once the signature is created, it is confirmed that it was made by the owner. Some typical scenarios where this can be applied are payment transactions or online contracts. However, it can also be seen in other scenarios to provide transparency and trust over the network. Likewise, if it wants to identify and validate the person who creates the signature, the signature could be supported by a digital certificate.

This mechanism is used in asymmetric encryption scenarios, so it is necessary a public and private key. What differentiates digital signature from the asymmetric encryption explained above is that, in this case, the private key will be used to sign the data. First, the process requires the calculation of the hash, then it will be encrypted with the private key and thus, preserve the message's integrity. Finally, the receiver will use the public key to verify the signature.

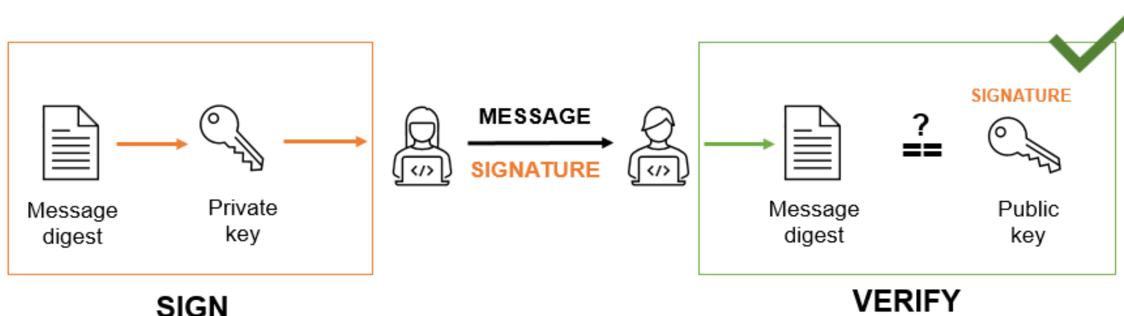


Figure 4 Digital signature scheme

[5] Cybersecurity & Infrastructure Security Agency

5.2.1. Elliptic Curve Digital Signature Algorithm (ECDSA)

There are several digital signatures schemes to apply, most of them based on RSA and ECC such as ECDSA, being the most used schemes. As this project will consider those mechanisms focused on the elliptic curve algorithm, in this section it will detail the functionality of the ECDSA scheme, in order to have a previous knowledge of what is going to be implemented later on.

Based on the ECC cryptographic algorithm, ECDSA provides digital signatures from the mathematics of elliptic curve cyclic groups over finite fields. This method is known as a variant of the ElGamal signature, in which is formed by an elliptic curve and the private key and public key, in which the latter is generated from the private key multiplied by the generator point.

The process to create the signature involves the hashing process, the private key and a random point computed by a random number. Then, to verify it only requires the hash with the public key and the signature. However, to make the algorithm more secure, there is a variant where the random value is changed by the HMAC of the private key.

The highlight of this scheme is that it provides a shorter signature with a higher security and better performance compared to other commonly used signatures like RSA or DSA. Hence, it has become one of the most widely used signature algorithms.

6. Applying security to GA4GH Phenopackets

This section details the practical part of the project in which the final goal is to implement a more secure Phenopackets schema through security mechanisms in order to provide phenotypic data exchange over the network.

As it was mentioned, an Agile scrum methodology was chosen to gradually develop the project. This decision is due to the fact that it is a completely new technology, so it is necessary to have a framework that allows to define the final Phenopacket as more information about the format is known. For this reason, the objectives concerning the use of the format and the security measures to be implemented have been redefined on the basis of the experience acquired.

Regarding the process, an iterative approach is followed, consisting of several stages that allow the development of the software while testing the implemented functions. This also includes reviewing the work done and deciding on how it can be improved as the project progresses. Below, the process followed for the development of the software is presented with three of the steps explained in this section: definition, design of the security mechanisms and development.

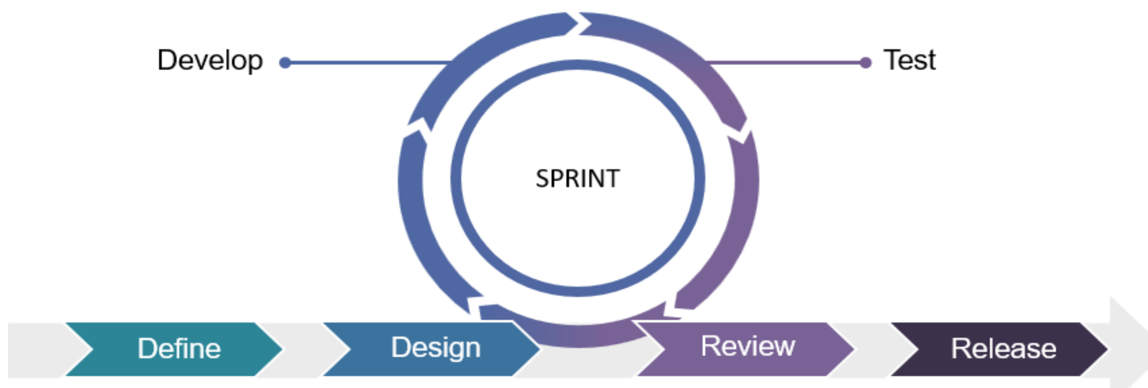


Figure 5 Scrum workflow

As Figure 5 shows, the first step is to define the Phenopacket schema. It is important to keep in mind that the original schema consisted of more fields than the schema described in section 3.2, however, it is the latter that will be used in the development process. As discussed in the next section, the schema is defined using the Protobuf exchange format, where it will explain how it works and how it was implemented.

Once the schema is generated, the next step is to select the available technology for Java environments that provides security within the code. The purpose of this step is to design the security features of the Phenopacket, being a total of three described in section 6.2.

To conclude, section 6.3 will explain the whole development process of this project. This step is performed along with the testing phase described in section 7; it corresponds to the sprint shown in the workflow. Each sprint develops a feature of the schema, in this case three different ones will be distinguished. First, it will describe which classes have been designed to create and make use of the schema. Next, all those classes created to provide the required security measures within Phenopacket will be explained and finally, the additional classes as well as the resulting files generated by the program will be shown. Once the functionalities are achieved, they are reviewed until it is finally decided that they are done.

6.1. Definition

The objective of this project is to add some security features to the Phenopackets schema created by GA4GH. To do so, it is necessary to first create the schema, as it has been defined with fewer fields (see section 3.2), it is decided to modify the original schema in order to create the designed one, then the technology called Protobuf makes these changes possible.

6.1.1. Protocol Buffers

Protocol Buffers [33], also known as Protobuf, is a language-neutral exchange format developed by Google being faster and simpler than other schemas such as XML or JSON for using automatic generation and validation of data objects.

Its purpose is to provide a serialization format for typed structured data packages that are up to a few megabytes in size mainly used in inter-server communications as well as for storing data archives on disk.

The schema is defined by *messages* which represent the data structure of the objects to be created. These messages are formed by the different fields that constitute the object using a unique number to identify those fields in the message binary format.

Related fields can be implemented as several data types, such as integers, strings, or Booleans. It also includes enumeration and *oneof* types, in which the latter is applied when a message has multiple optional fields and no more than one field is defined at the same time. However, personal message types can also be set, only has to specify a field type, and import the required definitions.

The version implemented for Phenopackets is proto3 which defines the fields as optional when it is not necessary to be set, repeated to specify fields that may be present a number of times, including zero, or singular, when a message can have 0 or 1 of this field but not more.

An example of a message is shown in Figure 6, in which the element Phenopacket is described in a file with the extension `.proto`. As it has the advantage of being readable in all programming languages, once the developer has selected his/her language, the proto file will generate a code to later encode or decode the data.

```
phenopackets.proto ×
syntax = "proto3";

package org.phenopackets.secure.schema;

import "phenopackets/secure/schema/core/base.proto";
import "phenopackets/secure/schema/core/disease.proto";
import "phenopackets/secure/schema/core/individual.proto";
import "phenopackets/secure/schema/core/phenotypic_feature.proto"
import "phenopackets/secure/schema/core/meta_data.proto";          ;
import "phenopackets/secure/schema/core/medical_action.proto";

option java_multiple_files = true;
option java_package = "org.phenopackets.secure.schema";

message Phenopacket {

    string id = 1;

    core.Individual subject = 2;

    repeated core.PhenotypicFeature phenotypic_features = 3;

    repeated core.Disease diseases = 4;

    repeated core.MedicalAction medical_actions = 5;

    core.MetaData meta_data = 6;
}
```

Figure 6 Source code of phenopackets.proto

There are two variables defined: java_multiple_files and java_package, since the programming language selected to develop the project is Java. The first one `option java_package = "org.phenopackets.secure.schema";` is the package declaration where the generated code will be placed. While `option java_multiple_files = true;` is to specify that separate files have to be created according to the message, enumeration and service identified in the file.

Then to create, import and export protobuf data to Java code, it is necessary to use Protoc, the protocol buffer compiler created by Google. It can be found on GitHub [34] as well as the official page of Google developers [33]. So, to generate the Java files associated to each protobuf the following command must be executed:

```
$ protoc --proto_path= phenopackets/secure/schema --java_out=. phenopackets/secure/schema/core/base.proto
```

- The parameter `-proto_path` is the source directory with the proto file
- The `-java_out` option is the destination directory where the java code will be written

Finally, the code generated by Protobuf provides a variety of methods that allow retrieving data from files or serializing data to a file as well as getting specific values from fields or checking if it exists. The following figure shows the Protobuf workflow provided by Google but adapted to the characteristics of this project.

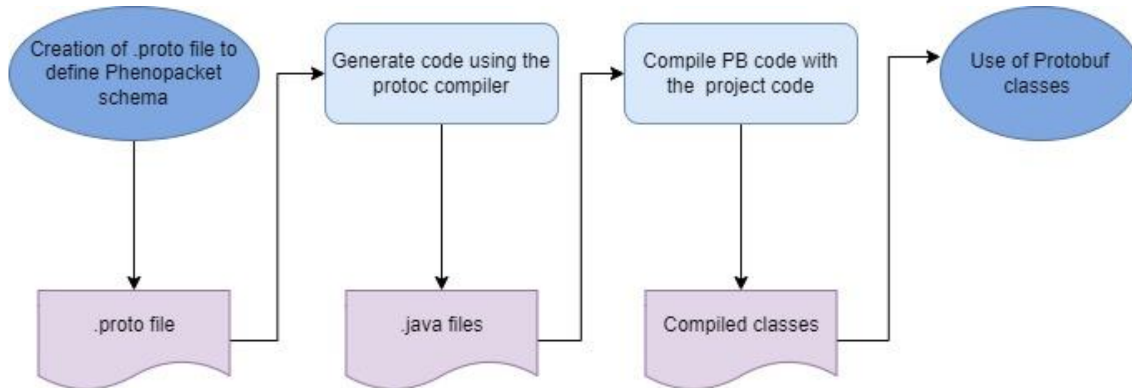


Figure 7 Protobuf workflow

6.2. Security mechanisms

In this section, it will be explained the three chosen mechanisms that will provide such security features. As defined in section 4.4, the data is separated into two categories: restricted and confidential. This classification is done to provide different levels of security depending on the data required, therefore, two different mechanisms have been chosen. In the case of restricted data, a Hybrid encryption will be used, while for confidential data a Hash function will be applied. In addition, the entire Phenopacket schema will be digitally signed by its creator to verify that the schema sent is the same from one side of the transmission to the other.

Based on this, an intensive search was carried out on which Java libraries were currently available that provide these features, finding Google cryptographic API known as Tink [35] and Bouncy Castle API [36].

Tink is an open-source library that allows you to apply different algorithms within your code via cryptographic building blocks that are named as primitives. Each of them offers a specific API according to the encryption method chosen, hence a specific key type. It has the advantage of connecting to an external Key Management System (KMS) to store the generated keys. However, since this project aims to show how the different mechanisms have been implemented as well as their use, a simpler implementation has been adopted where the keys will be stored locally in JSON format.

On the other hand, in order to compute the hash of the element, another package was required, namely Bouncy Castle. This is a lightweight cryptographic API developed in Java that provides several cryptographic algorithms including hashing. The API was selected because it allows to implement different strong hashing algorithms without requiring long lines of code. As in the previous case, the calculated hashes will be stored together in a file to facilitate the search and verification of each one of them.

It should be noted that the most correct and secure scenario would be through the use of KMS as well as some other tools or database that allows more secure storage of computed hashes.

Table 4 represents a summary of the methods applied as well as the technologies and elements to be protected, which will be explained in more detail in the following subsections.

Type of data	Security method	Technology
Restricted data	Hybrid encryption	Tink Library
Confidential data	Keccak Hash	Bouncy Castle API
Phenopacket	Digital Signature	Tink Library

Table 4 Security technologies to be applied in the Phenopacket schema

6.2.1. Hybrid encryption

The hybrid encryption has the objective of protecting data that wants to be exchanged. In Encryption algorithms it could be seen that there are two types of encryptions: symmetric and asymmetric, in which it was also pointed out how a better approach will be the combination of both algorithms.

In order to work in a hybrid encryption scenario, Tink incorporates two primitives called `HybridEncrypt` and `HybridDecrypt`, that allows sharing encrypted data using the public key to encrypt and the private key to decrypt. This feature is useful, since it is not necessary to keep any kind of secret, since only the public key of the receiver is used for encryption.

Regarding the key type selected, Tink incorporates different types to generate the keyset depending on their security and project requirements. In this case, the following key template has been selected:

```
ECIES_P256_HKDF_HMAC_SHA256_AES128_GCM
```

The template incorporates three different algorithms:

- First, it uses **ECDH over NIST P256** as a Key Encapsulation Mechanism (KEM) which relays in asymmetric encryption for symmetric key distribution
- Then, the **AES128-GCM** algorithm is selected as Data Encapsulation Mechanism (DEM) in charge to encrypt the message via symmetric encryption
- Finally, a **HKDF-HMAC-SHA256** as the Key Derivation Function (KDF) that derives secret encryption material from a shared key using a hash function as a pseudorandom function

These primitives also included the use of a context being an extra parameter that is linked to the data to be encrypted and whose information is publicly available. Its application allows to add integrity to the data since the context will be necessary for the correct decryption.

Finally, the following figure represents the implementation flowchart, which will be explained in more detail in section 6.3.3.

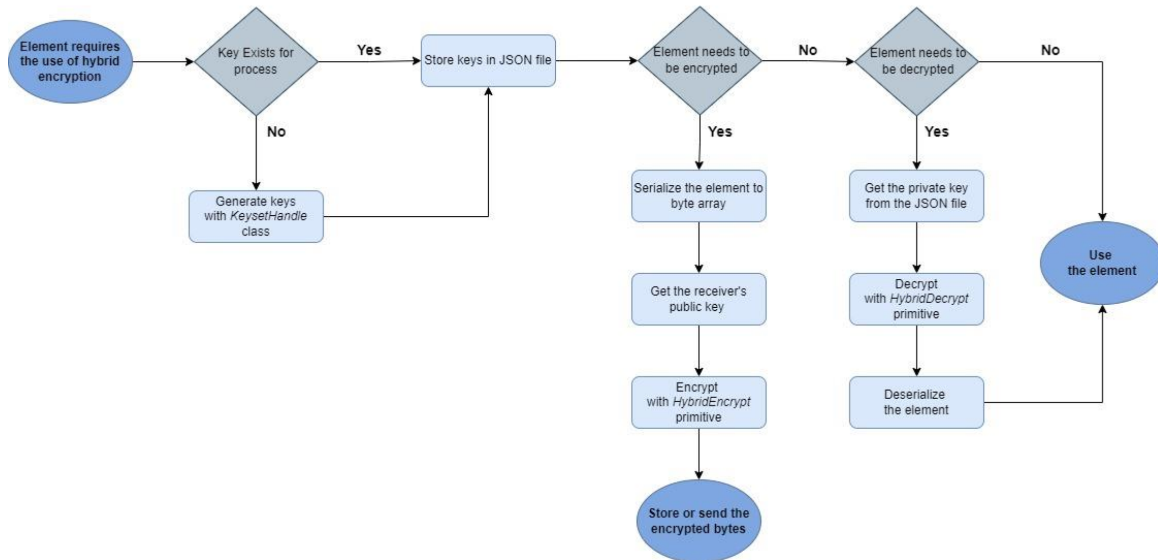


Figure 8 Hybrid encryption flowchart

6.2.2. Digital signature

The above mechanism provided secrecy to the Phenopackets elements; however, it is also important to bring authenticity with integrity to the schema in order to verify that the data has not been corrupted during the exchange. This authenticity also allows us to confirm that the Phenopacket sender is who he claims to be.

As this feature will be applied for the whole Phenopacket structure, Tink includes a primitive to implement the digital signature mechanism. Specifically, it will use two primitives: `PublicKeySign` and `PublicKeyVerify` which require another set of keys to carry out its process. In section 5.2.1, it was mentioned that a good digital signature algorithm is ECDSA, and since Tink provides it, the keys are generated using the **ECDSA_P256** key template.

In this case, the keys have been generated using the Google tool called *tinkey* that automatically creates them using a specific algorithm. This tool is used because it was considered that these keys represent a security threat if they are created from the code, due to the characteristics of the mechanism itself.

The following commands have to be executed in order to generate the keyset.

For the private key:

```
$ tinkey create-keyset --key-template ECDSA_P256 --out-format JSON --
out sk_sign.json
```

And for the public:

```
$ tinkey create-public-keyset –in sk_sign.json --in-format JSON --out-
format JSON --out pk_verify.json
```

As it can be seen, both keys are generated in JSON format with the `--out/in-format` parameter. Then, the private key named `sk_sign.json` is generated using the Elliptic Curve Digital Signature Algorithm (ECDSA) indicated with the `--key-template parameter`. Finally, the public key named `pk_verify.json` is created from the private key via `create-public-keyset` parameter.

Once the keys have been obtained, the Phenopacket can be signed. As shown in Figure 9, the process would start with the creator of the Phenopacket who wants to send the schema to another person. The steps to follow are similar to the previous mechanism, in this case the creator's private key is used to sign and the signature is stored together with the serialized Phenopacket in a JSON file. Once this is done, the creator can send the file that has just been created through a secure channel.

The receiver then obtains the file and uses the sender's public key to verify the signature. If the verification is correct, then it can confirm that the sender of the Phenopacket is really its creator and can make use of it, otherwise the format has been corrupted and should be discarded.

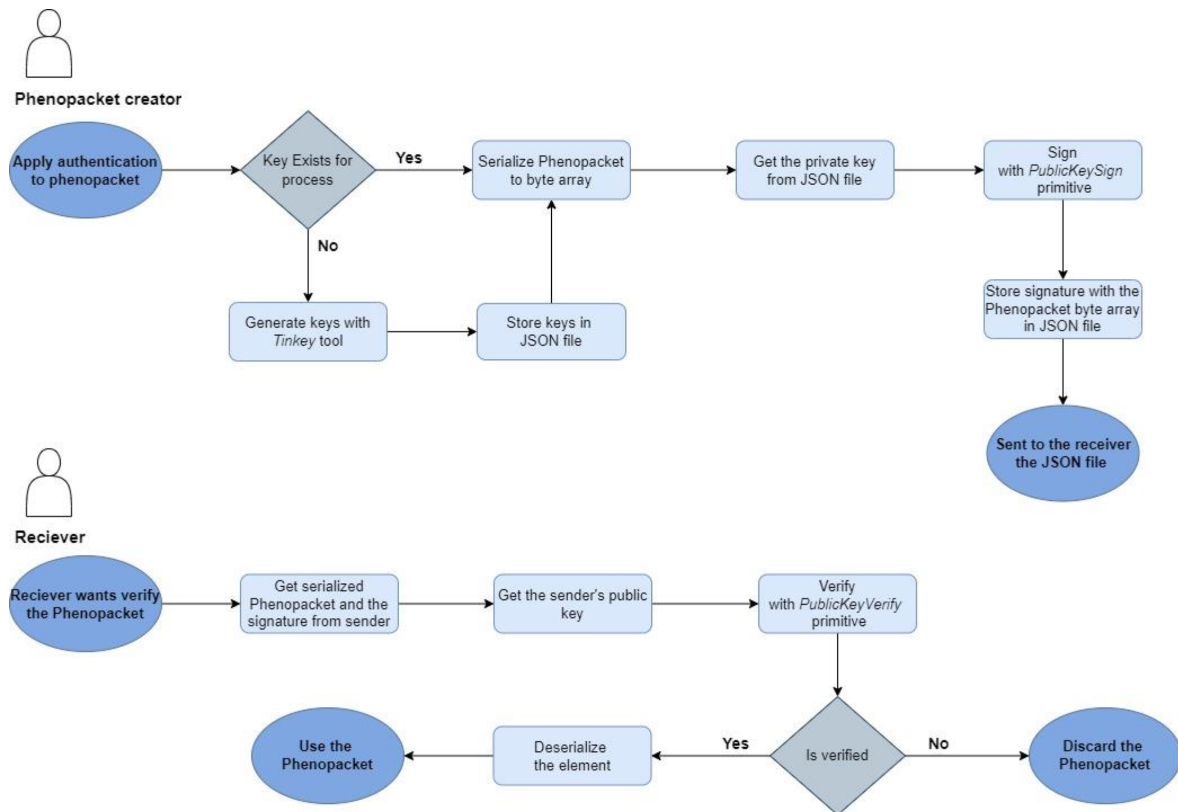


Figure 9 Digital signature flowchart

6.2.3. Hash

It was explained that a cryptographic hash function produces a unique output with the same size for the same input. Therefore, this mechanism was selected to ensure integrity in those circumstances where the data sent include confidential information, but still not represent a threat to the individual.

As previously mentioned, the Bouncy Castle API was used to implement this mechanism. Among the algorithms included in the API, it was decided to choose **Keccak256**, since it is one of the most secure algorithms currently considered. This mechanism is the easiest to implement and use, it only requires importing the library and calling the methods created by the class and it will automatically do the hashing.

6.3. Development

The development or implementation phase is the most important stage, as its purpose is to add the aforementioned mechanisms within the structure.

When working with an Agile methodology, changes are made progressively as some of the main ideas are discovered or changed. Therefore, in this section it will find annotated what changes have been made as well as the final implementation.

On the other hand, working with Protobuf implies using the classes and methods created by it with any possible modification and consequently, all new code must be implemented in different files. Therefore, to have a better schematized project, it was decided to export a JAR file of the Phenopackets schema project obtained from Protobuf, to later import it into the final project where the security features will be implemented. This decision has the advantage of being able to use in the future the original schema created by GA4GH, since it will only require adding the dependency to the project and modify the package names.

The project can be found on GitHub, but its structure can also be seen in the Project folder structure. However, in the following pages it will highlight where to find each of the classes created. All the technologies and libraries used to carry out the whole development process are mentioned in the following section. Subsequently, the different classes created to implement each of the elements and blocks that make up the schema will be described, as well as the three selected mechanisms, specifying the classes and functions designed for each one of them. Finally, additional functions and files have been created in order to improve the process of development.

6.3.1. Software requirements

All the software used has been chosen considering the programming language of the project, in this case Java. In addition, software for project management Apache Maven has been used, since in the original version it was the selected one to work with the format.

The different technologies and libraries used are listed below with the specific version added to the project:

- Visual Studio Code
- Java – v11
- Apache Maven
- Protoc compiler
- Java Security package
- Protocol Buffers library – v3.21.5
- Phenopacket secure schema JAR file – v1.0.0
- Tink Cryptography library – v1.6.1
- Bouncy Castle Crypto APIs – v1.7
- Nimbus JOSE + JWT – v9.23
- Junit Jupiter API – v5.7.1

To make use of these libraries, they must be added as dependencies in the file named `pom.xml`. As an example, the dependency corresponding to the Buffer Protocol is shown in Figure 10.

```
<dependencies>
        <dependency>
             <!-- Protobuf -->
            <groupId>com.google.protobuf</groupId>
            <artifactId>protobuf-java</artifactId>
            <version>3.21.5</version>
        </dependency>
</dependencies>
```

Figure 10 Protobuf dependency in pom.xml file

JAR file

Before starting to detail each one of the classes created, it is necessary to understand what the JAR file contains. As mentioned before, this file is composed of the different Java's classes generated by the `.proto` files in the Definition stage that were associated with each block and element of the proposed schema.

Once those files are generated, each element of the schema has two different Java classes named as Builder and the main class which cannot be modified. The Builder class is used to create the instance of the class, so it only has setter methods. Then, the main class of the element contains accessor methods for each field that composes the element, this means that getters and setters are defined to set a proper value in the field and finally construct the element making use of the `build()` method.

Furthermore, each main class has methods for writing and reading messages of the type chosen using the protocol buffer binary format such as serializing the message to return a byte array containing its raw bytes or parses a message from the given byte array.

As it can be seen, the project contains a large number of files, which could make it difficult to read the new code created. Therefore, it was decided to export the project to a JAR file and include it in the final project as a dependency, as Figure 11 shows.

Note that the JAR file has to be built in the repository before adding it as a dependency through the following command:

```
$ mvn install:install-file -Dfile=[jar_file_path]
```

```
<dependencies>
        <dependency>
             <!-- JAR file -->
            <groupId>org.phenopackets.secure.schema</groupId>
            <artifactId>Phenopackets-secure-schema</artifactId>
            <version>1.0.0</version>
        </dependency>
</dependencies>
```

Figure 11 Phenopacket schema JAR file dependency in pom.xml file

This solution provided two important advantages. Firstly, it allows to have a better structure within the project and secondly, by being treated as a dependency, it allows the possibility that in the future the original schema can be added where it will only be required to update the imports within the classes.

6.3.2. Phenopacket schema classes

In the JAR file it has been explained that the automatically generated classes cannot be modified, this implies that in order to work with the different elements of the Phenopacket, new Java's classes are required.

In this case, three classes have been designed to create and use the format, which can be found at: **`\src\main\java\phenopackets\schema.`** Each of these classes are explained below being one for blocks elements, another for the main elements that compose the Phenopacket and finally, a class to create a secure Phenopacket schema.

A. BlockBuilder

The *BlockBuilder* class corresponds to the implementation of all the building blocks that are defined within the main elements. Specifically, these blocks correspond to:

- Ontology Class
- Resource
- Evidence
- Quantity
- Dose Interval
- Time elements: Age, Timestamp, TimeInterval
- Medical actions: Procedure, Treatment ,RadiationTherapy, TherapeuticRegimen

Before starting to explain the different functions created, it has to be noted that the `KaryotypicSex` field could not be secured as proposed in section 4.4, since the implementation of the field does not allow making any kind of change on it. However, it was finally decided that the field will no longer be categorized as restricted or confidential data, since not provides the sex of the individual in a currently manner, and thus it provides some security.

All blocks have been created using the same procedure. First, as input parameters the fields that compose that element are required. Then, to create a new instance, the methods used are those implemented by default in the aforementioned Java classes. Specifically, the following three methods are called:

1. The newBuilder method is the function in charge of creating the new instance of the object, so it will always be called, whether it is a block or a main element
2. The set**[name_field]** method is the function that adds the value specified by the input parameters to each field. Each field has its own set method, and this method also performs the corresponding input validations to use only the specified types. This means that if a field corresponds to a String, it can only contain String values
3. The build() method is used to build the object which will be returned as an output parameter

Figure 12 shows the method to create an `OntologyClass` object, as it can be observed, the method has two input values: the ontology identifier and the label.

```
public static OntologyClass createOntologyClass (String id, String label){

     return OntologyClass.newBuilder()
             .setId(id)
             .setLabel(label)
             .build();
  }
```

Figure 12 Source code of createOntologyClass method

In this class there are two methods to highlight related to `Age` element. First of all, it is the block that assigns the value of `timeAtLeastEncounter` field, which was categorized as restricted data, and therefore to be protected. Thus, the first designed function uses a hybrid encryption method in its creation, as shown in Figure 13, since it is the one chosen to ensure the security of this type of data.

```
public static TimeElement createTimeElementAge(byte[] isoDuration, byte[]
context) throws IOException, GeneralSecurityException, URISyntaxException{

     // Encrypt the age and store it in Base64
     byte[] cipherBytes = HybridEncryption.hybridEncryption(MODE_ENC,
     isoDuration, context);
     String cipherAge = Base64.getEncoder().encodeToString(cipherBytes);

     // Create age element
     Age age = Age.newBuilder().setIso8601Duration(cipherAge).build();

     // Create and return TimeElement
     return TimeElement.newBuilder().setAge(age).build();
 }
```

Figure 13 Source code of createTimeElementAge method

The second method is a function to recover the original value by means of the decryption function. As Figure 14 shows, the method gets the `Age` instance from the `TimeElement` object, and then it uses the decryption function to recover the plaintext. The hybrid encryption will be explained in more detail in section 6.3.3.

```
public static String getAge(TimeElement timeElement, byte[] context)
throws IOException, GeneralSecurityException, URISyntaxException{
      // Get age from TimeElement and the corresponding value
      Age ageElement = timeElement.getAge();
      String isoDuration = ageElement.getIso8601Duration();

      // Decrypt age
      byte[] ageBytes = Base64.getDecoder().decode(isoDuration);
      byte[] age = HybridEncryption.hybridEncryption(MODE_DEC, ageBytes,
      context);

      // Return the Age value as String
      return new String(age);
  }
```

Figure 14 Source code of getAge method

B. MainElements

The *MainElements* class represents the creation of the five elements that constitute the Phenopacket being: Individual, PhenotypicFeature, MetaData, Disease and MedicalAction.

The methodology followed to build each element is the same as blocks, but in this case some elements require an input validation for its own creation. Specifically, those elements including a time element field either age or a time interval.

Starting with `Individual` element, only one method has been designed to build the element. In this element, it is necessary to validate the input parameter named `timeAtLastEncounter` since it corresponds to the age of the subject and also, it is one of the encrypted fields. It is also included the subject identifier , and then another function was implemented to create them as arbitrary identifiers (see Figure 15). This function called `generateIndividualId()` returns an identifier formed by a letter together with six digits, which are chosen randomly thanks to the cryptographically Java class *SecureRandom* of the Security Java package.

The same input validation is required for the `Disease` element according to its `onset` field. It should also be mentioned that the creation of a disease is divided into two methods: one for rare or common disease and another for cancer, since it may include or not all the fields depending on the subject's disease.

The last required input validation is for the `PhenotypicFeature` element, in which the fields regarding the time at was first observed as well as was resolved, it is set via a timestamp.

```
private static String generateIndividualId() throws NoSuchAlgorithmException, NoSuchProviderException{

	SecureRandom secureRandom;

	String randomNumber = new String();

	// Get a true random number generator
	try {
		secureRandom = SecureRandom.getInstanceStrong();
	} catch (NoSuchAlgorithmException ex) {
		secureRandom = new SecureRandom();
	}

	// Get a random number of 6 digits
	for (int i=0; i<6; i++) {
		randomNumber += String.valueOf(secureRandom.nextInt(9));
	}

	// Individual identifier begins with P
	String id = "P"+randomNumber;

	return id;
}
```

Figure 15 Source code of generateIndividualId method

As for the `MedicalAction` element, it has been divided into several methods, each one implementing an allowed action using the same procedure previously described. There are 8 methods available, two for each action, where the first four create the element with all its fields whereas the last four only build the required action.

Figure 16 represents the function to create a medical action related to treatment.

```
public static MedicalAction createMedicalTreatment(Treatment treatment, OntologyClass treatmentTarget, OntologyClass treatmentIntent, OntologyClass treatmentResponse, List<OntologyClass> adverseEvents, OntologyClass treatmentTermination){

	return MedicalAction.newBuilder()
		.setTreatment(treatment)
		.setTreatmentTarget(treatmentTarget)
		.setTreatmentIntent(treatmentIntent)
		.setResponseToTreatment(treatmentResponse)
		.addAllAdverseEvents(adverseEvents)
		.setTreatmentTerminationReason(treatmentTermination)
		.build();
}
```

Figure 16 Source code of createMedicalTreatment

Finally, the MetaData element involves the development of five functions in order to implement the security features chosen in the second phase of the project. An explanation of each of these methods is given below.

The first method called createMetaData() is the same function created for the building of the rest of elements, where any security mechanism has been implemented. Then, a second function with the same objective named protectedMetadataCreator() is used to encrypt the creator of the Phenopacket and construct the element using a hybrid encryption. The output cipher is used to set the value of the createdBy field.

For those situations where it may be necessary to protect the entire element, two functions have been developed to allow encryption and decryption. The first method called protectedMetaData() has as an input the particular element, which is serialized to a byte array and then encrypted. While the second method getMetaData() uses the decrypt function to recover the original bytes in order to reconstruct the element using a parse function provided by the original Java class.

```
public static MetaData getMetaData(byte[] metaDataBytes, byte[] context)
throws IOException, GeneralSecurityException, URISyntaxException{

    // Decrypt MetaData element
    byte[] plainMetaData = HybridEncryption.hybridEncryption(MODE_DEC,
    metaDataBytes, context);

    // Create a new MetaData element using parseFrom function
    return MetaData.parseFrom(plainMetaData);
}
```

Figure 17 Source code of getMetaData method

Additionally, an extra function has been created to decrypt and recover the creator called getMetaDataCreator() which follows the same procedure that the last one mentions, but in this case gets the corresponding bytes of the createBy field and use it as an input parameter in the decryption function, to finally return the original value.

C. SecurePhenopacket

Once we have all the elements and blocks defined, the construction of the Phenopacket can be done. For this purpose, a class called *SecurePhenopacket* was defined to create a Phenopacket with all the main elements as well as to implement the security features regarding it.

As in the previous cases, the first designed method was the one that allows the creation of a new Phenopacket, which has been named as createPhenopacket(), using the same procedure as the other elements described above. In addition, a Phenopacket also needs an identifier, so the function generatePhenopacketId() was implemented to generate a universal unique identifier (UUID) class as follows:

```
public static String generatePhenopacketId() {
    return String.valueOf(UUID.randomUUID());
}
```

Figure 18 Source code of generatePhenopacketId method

In relation to the security features added to the Phenopacket element, it was decided to implement the digital signature to provide authentication. Therefore, within this class there are two functions: `signPhenopacket()` and `verifyPhenopacket()`, that allow signing and verifying Phenopackets both functions, only requires the element as input parameter.

As an example, Figure 19 shows the function that allows verifying a Phenopacket, the *DigitalSignature* class is explained in section 6.3.3.

```
public static void verifyPhenopacket(Phenopacket phenopacket) throws
IOException, URISyntaxException, ParseException{

	// Serialize to byte array
	byte[] phenopacketBytes = phenopacket.toByteArray();

	// Get the ID as identifier
	String id = phenopacket.getId();

	// Verify the element
	DigitalSignature.protectWithDS("verify", phenopacketBytes, id);
 }
```

Figure 19 Source code of verifyPhenopacket method

In the case of wanting to protect the `MetaData` element, a function has been created to retrieve the element from the Phenopacket and then encrypt it by means of the function explained in the previous section. Simultaneously, the element is removed from the Phenopacket so that no plain data is kept, however it is required to store both elements together in order to be able to proceed later to its decryption. This function can be found under the name `protectMetaData()`.

Finally, several additional functions have been designed to retrieve the stored bytes from files for its future re-building, as well as to import and export a Phenopacket in JSON format.

```
public static void exportPhenopacket(Phenopacket phenopacket) throws
URISyntaxException {

	try{
		String jsonString =
		JsonFormat.printer().includingDefaultValueFields().print(phenop
		acket);
		File phenopacketJson = externalResource.createNewFile("P-
		"+phenopacket.getId()+".json" );

		BufferedWriter fileWriter = new BufferedWriter(new
		FileWriter(phenopacketJson));
		fileWriter.write(jsonString);
		fileWriter.close();
	}catch(IOException ex){
		ex.printStackTrace();
	}
}
```

Figure 20 Source code of exportPhenopacket method

6.3.3. Security Mechanisms classes

This section explains the different classes designed for the chosen security mechanism, which can be found at: `\src\main\java\phenopackets\securityMechanisms`.

First, the *HybidEncryption* class that allows encrypting the data considered as restricted is implemented, both the encrypt and decrypt functions are explained. Next, it describes the *DigitalSignature* class including the process of signing and verifying an element. Finally, for those data that require a minimum of security, previously considered as confidential data, a class named *Hashing* has been developed and thus, provide integrity to the added value.

A. HybridEncription

The *HybridEncription* class is formed by multiple methods that constitute the entire process of encryption. Specifically, these methods are:

- `createKeySet()` – it creates the key set
- `private hybridEncryption()` – it performs the encryption process
- `private hybridDecryption()` – it performs the decryption process
- `public hybridEncryption()` – main function that runs the specified encryption mode
- `saveInFile()` – it stores the corresponding encrypted bytes in a JSON file
- `getCipherBytes()` – it retrieves the stored cypher bytes

To proceed with the hybrid encryption, it is necessary to have a key set, for that reason the function `createKeySet()` was created. This method generates a pair of keys, private and public, considering the chosen algorithm and storing each key in a JSON file for ease of use.

Then, the `hybridEncryption()` method implements the encryption process using the *HybridEncrypt* class of Tink. Likewise, the private method to decrypt called `hybridDecryption()` uses the functions defined in *HybridDecrypt* class.

Both actions follow the same methodology, first the keyset needs to be read and stored into the *KeysetHandle* class. Depending on the action to perform, a primitive will be created getting the object encryptor or decryptor and subsequently, it will call to the encrypt or decrypt function implemented by Tink. Finally, in order to simplify handling and sending, both the ciphertext and the plaintext will be serialized to a byte array.

Figure 21 shows the private method related to the encryption step.

```
private static byte[] hybridEncryption(byte[] element, byte[] contextInfo)
throws GeneralSecurityException, URISyntaxException{

      // Input validation
      if (element == null || element.length == 0){
            throw new NullPointerException();
      }
      if (contextInfo == null || contextInfo.length == 0){
            throw new NullPointerException();
      }

      // Read the keyset into a KeysetHandle
      KeysetHandle handle = null;
      try {
            handle =
            CleartextKeysetHandle.read(JsonKeysetReader.withFile(externalRe
            source.getFileFromResource(PK_FILE)));
      } catch (GeneralSecurityException | IOException ex) {
            System.err.println("Process error: " + ex);
      }

      // Get primitive related to the encryption
      HybridEncrypt encryptor = null;
      try {
            encryptor = handle.getPrimitive(HybridEncrypt.class);
      } catch (GeneralSecurityException ex) {
            System.err.println("Process error: " + ex);
      }

      // Encrypt and return the ciphertext
      byte[] ciphertext = encryptor.encrypt(element, contextInfo);
      return ciphertext;
 }
```

Figure 21 Private method to encrypt an element

Since both previous methods were designed as private, a main function was created to perform any of the two functions by passing the required action as a parameter. This method, also named as `hybridEncryption()` checks if the corresponding keys exist as well as the indicated mode is correct. Once the validations were done, the method called the previous functions depending on whether it wanted to encrypt or decrypt the data.

Figure 22 shows the complete function where a mode, an element and a context are required as an input parameter. The context parameter was explained in the Hybrid encryption section and represents the Phenopacket ID associated with the data.

```
public static byte[] hybridEncryption(String mode, byte[] element, byte[]
context) throws IOException, GeneralSecurityException, URISyntaxException{

      byte[] res;

      // Initialize the hybrid configuration
      HybridConfig.register();

      // Check if exist the keyset
      File hybridFile = externalResource.getFileFromResource(SK_FILE);

      List<String> lines = Files.readAllLines(hybridFile.toPath());

      // If not, create the keyset for the process
      if (lines.size()==0) {
            createKeySet();
      }

      // Check the mode is correct
      if (!mode.equals("encrypt") && !mode.equals("decrypt")) {
            System.err.println("Incorrect mode.");
      }

      // If the mode is "encrypt", then call function hybridEncryption,
      otherwise call hybridDecryption
      if (mode.equals("encrypt")) {
            res = hybridEncryption(element,context);
            return res;
      }else{
            res = hybridDecryption(element, context);
            return res;
      }
  }
```

Figure 22 Source code of hybridEncryption method

B. DigitalSignature

The *DigitalSignature* class is composed of three methods that carry out the process of signing and verifying a Phenopacket. This class is also using the Tink cryptographic library by Google. An additional function was also designed to search for a signature along with the associated Phenopacket in a JSON file.

The `signElement()` method has the functionality of signing a Phenopacket using the *PublicKeySign* class provided by Tink as well as the private key. It has been designed as a private method and the only required input parameter is the element bytes, therefore the returned output is the corresponding signature bytes.

```
private static byte[] signElement(byte[] element) throws
GeneralSecurityException, IOException, URISyntaxException{

	// Input validation
	if (element == null || element.length == 0){
		throw new NullPointerException();
	}

	// Read and store the private key to sign
	KeysetHandle handle =
	CleartextKeysetHandle.read(JsonKeysetReader.withFile(resourceFile.get
	FileFromResource(SK_FILE)));

	// Create the signer instance and get the associated primitive
	PublicKeySign signer = null;

	try{
		signer = handle.getPrimitive(PublicKeySign.class);
	}catch(GeneralSecurityException ex){
		System.err.println("Process error: " + ex);
	}

	// Sign and return the signature bytes
	byte[] signature = signer.sign(element);

	return signature;
}
```

Figure 23 Private method to sign an element

To verify the signature, it is required another function that uses the returned bytes together with the element bytes and proceeds with the verification. The `verifyElement()` is a private method that uses the *PublicKeyVerify* class from Tink to perform this action, although the steps followed are the same shown in Figure 23, in this case a Boolean value is returned to validate the process.

As the hybrid encryption class, a main function named `protectWithDS()` has been implemented allowing another class to apply this mechanism. This method requires the element name, its bytes, and the mode to apply as an input parameter.

Regarding the modes, there are two possibilities: sign or verify, any other input will be rejected, and the procedure will fail. If the mode is "sign", then the private function is called, and the returned bytes are stored in a file together with the Phenopacket byte array. Then, this file will be used to search for the corresponding signature and check via the last function if the signature is valid.

```
public static void protectWithDS(String mode, byte[] elementBytes, String
elementID) throws IOException, URISyntaxException, ParseException{
      //Set variable
      Boolean isVerified = false;

      try {
      // Set the Digital Signature configuration
      SignatureConfig.register();

      // Check the mode is correct
      if (!mode.equals("sign") && !mode.equals("verify")) {
            System.err.println("Incorrect mode.");
      }

      if(mode.equals("sign")){
            byte[] signatureBytes = signElement(elementBytes);

            // Store the signature in a jsonObj with the signature
            String ptSignature = new
            String(Base64.getEncoder().encode(signatureBytes),
            StandardCharsets.UTF_8);
            String ptPhenopacket = new
            String(Base64.getEncoder().encode(elementBytes),
            StandardCharsets.UTF_8);
            externalResource.createJSONFile(SIGNATURES_FILE, ptPhenopacket,
            elementID);
            externalResource.createJSONFile(SIGNATURES_FILE, ptSignature,
            elementID+"-Signature");

      } else if(mode.equals("verify")){
            isVerified = searchSignatureAndVerify(elementBytes, elementID);
            System.out.println("Verified:" + isVerified);
      }
  } catch (java.security.GeneralSecurityException e){
      System.out.println("Error protecting with DS");
  }
  }
```

Figure 24 Source code of protectWithDS method

As Figure 24 shows, the verification process uses a private method named `searchSignatureAndVerify()` which corresponds to the search of a signature stored in a file whose only requirement is to specify the Phenopacket ID used as the filename.

C. Hashing

The *Hashing* class performs hash computation functions for different elements within the schema. Specifically, it includes the main function being responsible for hashing and five more functions that allow to compute, get, and check the hash of the elements classified as confidential data (see section 4.4).

Starting with the function in charge of hash calculation, `computeHash()` is a private method that uses the `Keccak.Digest256()` instance offered by Bouncy Castle and returns the corresponding hash bytes of the element passed as input parameter.

```
private static byte[] computeHash(byte[] element) {

     // Input validation
     if (element == null || element.length == 0){
            throw new NullPointerException();
      }

     // Generate a new Keccak instance
     Keccak.Digest256 digest256 = new Keccak.Digest256();

     // Compute hash of an element
     byte[] hashBytes = digest256.digest(element);
     return hashBytes;
}
```

Figure 25 Source code of computeHash method

The rest of functions use this method at the moment to calculate it, so a function was created for each element with the same procedure. First, it is necessary as input parameter the element and the Phenopacket ID. Once the hash has been computed, the obtained bytes are stored in a file linked to the element's name which will allow for checking if the hash is still the same or has been manipulated. As a proof of concept, the following figure shows the function developed for the `Disease` element.

```
public static String computeDiseaseHash(Disease disease, String
phenopacketId ) throws IOException, URISyntaxException{
     // Serialize the Disease element to a byte array
     byte[] diseaseBytes = diseaseElement.toByteArray();

     // Compute the hash
     byte [] hash = computeHash(diseaseBytes);

     // Store the hash in a file linked with its name
     externalResource.addHashToFile(phenopacketId, hash, diseaseName);

     // Return the hash as String
     return new String(Hex.encode(hash));
 }
```

Figure 26 Source code of computeDiseaseHash method

Next, to retrieve the stored hash, the `getHash()` function has been created. In this case, the file and the name of the element is required as an input parameter. Then, the function searches in the file and returns the hash according to the input name.

Finally, to check if the hash remains the same, a function has been developed that calculates the hash of an element and then compares it with the stored one. In Figure 27 it is shown how a Boolean variable is returned to validate the hash, so if the hashes match, the function returns *true*, otherwise the element is corrupted, and the function returns *false*.

```
public static boolean checkHash(byte[] element, String storedHash){
	boolean result = false;

	// Input validation
	if (element!=null && !storedHash.isBlank()){
		byte [] hash = computeHash(element);
		String computedHash = new String(Hex.encode(hash));

		// Compare both values
		result = computedHash.equals(storedHash);
		System.out.println(result);
	}else{
		throw new NullPointerException();
	}
	return result:
}
```

Figure 27 Source code of CheckHash() method

6.3.4. Additional class and files

During the course of the development, certain circumstances have arisen that required external functions not related to the purpose of the project, but necessary for its implementation. In addition, for a better management of the project it was algo required to create different external files.

In this section it is going to explain the class that includes all these external functions named as *ExternalResources* and which can be found at: **\src\main\java\ phenopackets\securityMechanisms.** Next, the different external files with those created by the project will be listed. These files are located under the *resources* folder.

A. ExternalResource class

This class has been created to implement the different functions needed to develop the main security features of the project. Basically, the class is composed of several methods that allow it to work with external files, from the creation of a new file to looking for a specific one. These methods are:

- `getFileFromResource()` – it returns a file located in resources folder, otherwise creates a new one
- `createNewFile()` – it creates a new file
- `addHashToFile()` – it adds a new hash to file
- `createJSONFile()` – it creates a new JSON file and adds a JSON object. In case the file already exists, then add the new object to the file
- `getJSONFromFile()` – it returns a JSON object from the file

Among the different functions that can be found in the class, it is important to highlight three, since they are called in the encryption and signing process. The first one is named `addHashToFile()` and is used in the *Hashing* class to save the computed hash in a file. This function needs the filename to store the hash with its value as an input parameter. Then, the method gets the file and adds in a new line the hash encoded in hexadecimal.

The next method created was `createJSONFile()` and as its name suggests, it is designed to create JSON files. This function is used to store both the different signatures that can be created, as well as the encrypted elements, since both require to be linked to another factor, such as the name of the element or directly the Phenopacket serialized in byte array.

It has been contemplated that a file already contains a JSON object and therefore, only needs to add a new field within it. For example, if the sender wants to send more than one signed Phenopacket at a time, it will require to sign each of them and save the corresponding signature in one file. In that case, the previous function calls a third one that allows retrieval of the stored JSON object, and which will be saved in a variable to be manipulated later, adding this new field.

This new method is called `getJSONFromFile()` and Figure 28 shows how it was implemented.

```
public JSONObject getJSONFromFile(File jsonFile) throws ParseException,
IOException {
	JSONObject js = new JSONObject();

	try(FileReader reader = new FileReader(jsonFile)) {
		JsonReader jsReader =  new JsonReader(reader);
		jsReader.beginObject();

		// Save fields and keys
		while (jsReader.hasNext()) {
		    js.appendField(jsReader.nextName(), jsReader.nextString());
		}
		jsReader.endObject();
		jsReader.close();

	}catch (FileNotFoundException e) {
		e.printStackTrace();
	}
	return js;
}
```

Figure 28 Function to get a JSON object from file

B. Files

The previous section showed how additional functions were required to create files. The files created automatically during the execution of those functions are listed below with a brief explanation of what each of these files are.

Note that there are two more files within this project that are not explained in this section, since they were created using the tinkey tool (see Section 6.3.3).

The created files are:

- `signatures.json`. A JSON file where all the signatures created with the *`protectWithDS()`* method linked to the Phenopacket identifier are stored
- `sk_hybridEnc.json`. The private key generated with the *`createKeySet()`* method of the *HybridEncryption* class is stored in this file
- `pk_hybridEnc.json`. Like the previous file, this one has the public key generated from the private key
- **`[PhenopacketID]`**`.txt`. This file stores the different hashes that can be computed in the *Hashing* class where **[PhenopacketID]** is the Phenopacket identifier that includes the element
- **`P-[PhenopacketID]`**`.json`. Same as the above file, but in this case the different encryptions performed in *HybridEncryption* are stored along with the serialized Phenopacket in a byte array

7. Evaluation

Once all the security mechanisms have been developed, they must be tested in order to corroborate that the implementation has been correctly done. Considering the scope of the project, these verification tests must include both the correct creation of the different elements and Phenopacket, and also that the previously described functions fulfill each of their functionalities.

Therefore, this section will explain the several tests designed to evaluate whether the implementation has been carried out correctly or not. All verification tests created in the project can be found at: **`\src\test\java\tfm\phenopackets_schema`** and **`\src\test\java\tfm\securityMechanisms`**.

Furthermore, in order to perform the test, two java classes have been created, which can be found in the *examples* folder. These classes generate the different elements associated with a medical case, whose values have been obtained from [23]. Specifically, these two examples correspond to a covid19 case and a cancer case, one example with the values of these classes could be found at appendix Covid19 example.

7.1. Verification tests

The tests explained in this section have been developed using the JUnit framework that allows automated tests on Java via `@test` annotation and check if the obtained results are the expected ones.

7.1.1. Phenopacket creation test

Before starting to check if security mechanisms are correctly implemented, the creation of the Phenopacket elements as well as the Phenopacket schema must be tested to ensure that the following proves could be created without errors. As a result, eight test cases were developed in *SecurePhenopacketTest.java* :

- `checkIndividualCreation()` – it creates an `Individual` element and compares it with the original values
- `checkPhenotypicFeatureCreation()` – it creates a `PhenotypicFeatures` element and compares it with the original values
- `checkDiseaseCreation()` – it creates a `Disease` element and compares it with the original values
- `checkMedicalActionCreation()`– it creates a `MedicalAction` element and compares it with the original values
- `checkMetaDataCreation()` – it creates a `MetaData` element and compares it with the original values
- `checkPhenopacketCreation()` – it creates a `Phenopacket` element and compares it with the original values
- `checkJsonExportation()` – it returns a JSON file with the Phenopacket schema
- `checkImportPhenopacketFunction()` – it verifies that a Phenopacket can be retrieved from a JSON file

Then, the procedure followed in these tests was as follows:

1. Get the values from the covid19 example
2. Each element has its own creation functions in covid19 class, which was used to create the element to check
3. Finally, each field of the element is compared with the value defined in the class
4. If the values are equals, the test will be passed, otherwise an error is shown that indicates which values are not the same

All tests created were passed, however, to test the correct implementation of ID's it was changed to the assertion in `checkPhenopacketCreation()` test since its creation implies arbitrary identifiers for the subject of the Phenopacket, so the following error is shown.

```
Expected
1  id: "17a1a6ad-2ea1-40ee-9308-1401fa096c0c"
2  subject {
3-   id: "P715733"
4    time_at_last_encounter {
5      age {
6-       iso8601duration: "AW9xkcME+3IGfkOU/
-        x1JGRsjs0uw++fLn5VUm9bcPt9GyPre33N/
-        Y1QohflfkmwjhjgcKQ2snUfoOW2axmv6qvwuRuBgASXm
-        gW/q9kQ5Q7hf2gt1wc+lPD+dDt9Lf8EwVjdXdfra"
7      }
8    }
9    vital_status {
10     status: DECEASED
11   }
12   karyotypic_sex: XY
13 }
14 phenotypic_features {
```

```
Actual
1  id: "17a1a6ad-2ea1-40ee-9308-1401fa096c0c"
2  subject {
3+   id: "P038416"
4    time_at_last_encounter {
5      age {
6+       iso8601duration: "AW9xkcME2DEvq2ta95yy
+        +l4D4IcXGbUx32uL+d/AlzulR6sxCJlW5p/
+        qnkuEhpRg2wZgSZ2267eDaQtSPDufaPA3mbVhnEELZdG
+        2eq2asz5kdvwDuBFkmIsJo3zC66sMSuPBP6dk"
7      }
8    }
9    vital_status {
10     status: DECEASED
11   }
12   karyotypic_sex: XY
13 }
14 phenotypic_features {
```

Figure 29 Expected error in phenopacketCreation() test

As Figure 29 shows, another value differs from the expected Phenopacket, which corresponds to the age value. This fact occurs because of the security mechanism implemented in this field. It was explained that it uses hybrid encryption to encrypt the value and use some randomness to provide more security, so it is correct that both values are not equal.

7.1.2. Hybrid encryption tests

This class is implemented to test the methods developed in the *HybridEncryption* class using the Covid19 case. To avoid unexpected errors, the Phenopacket ID was created by the `generatePhenopacketId()` function and defined within the example case. Likewise, it has five test cases, a summary of them is provided in the following tables.

As a proof of concept, the following figure shows that all tests have been passed correctly:

```
HybridEncryptionTest 494ms
  encryptAge() 226ms
  checkAgeDecryption() 14ms
  createMetaDataProtectingCreator() 30ms
  checkMetaDataEncryption() 132ms
  getAndDecryptElementsFromFile() 92ms
```

Figure 30 Tests created for hybrid encryption

The tests related to the `Age` element are:

<table>
<tr><td>Test name</td><td>encryptAge()</td></tr>
<tr><td>Test Objectives</td><td>Test that will encrypt an iso8601 using the HybridEncription class and then insert it into the Age block. Specifically:
• Verify the correct implementation of <code>createAge()</code> function</td></tr>
<tr><td>Test expected result</td><td>Expected encrypted age and not the original value provided.</td></tr>
<tr><td>Test and result obtained</td><td>
</td></tr>
</table>

Table 5 Test to verify Age encryption

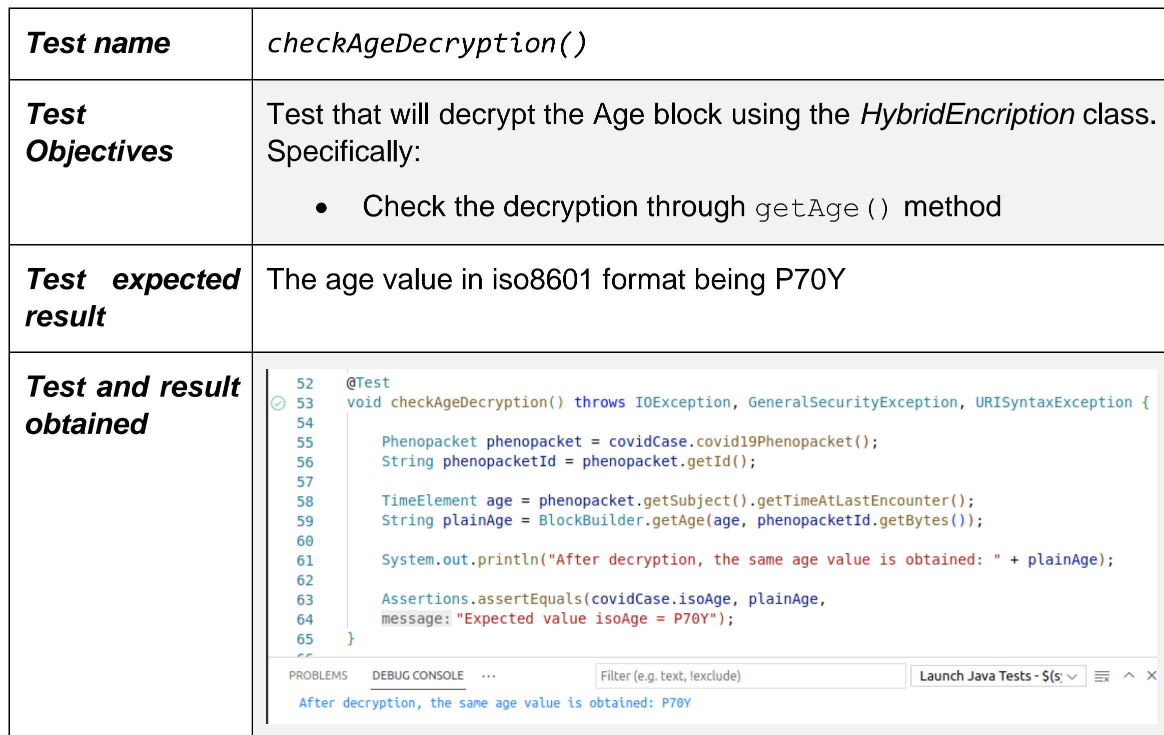
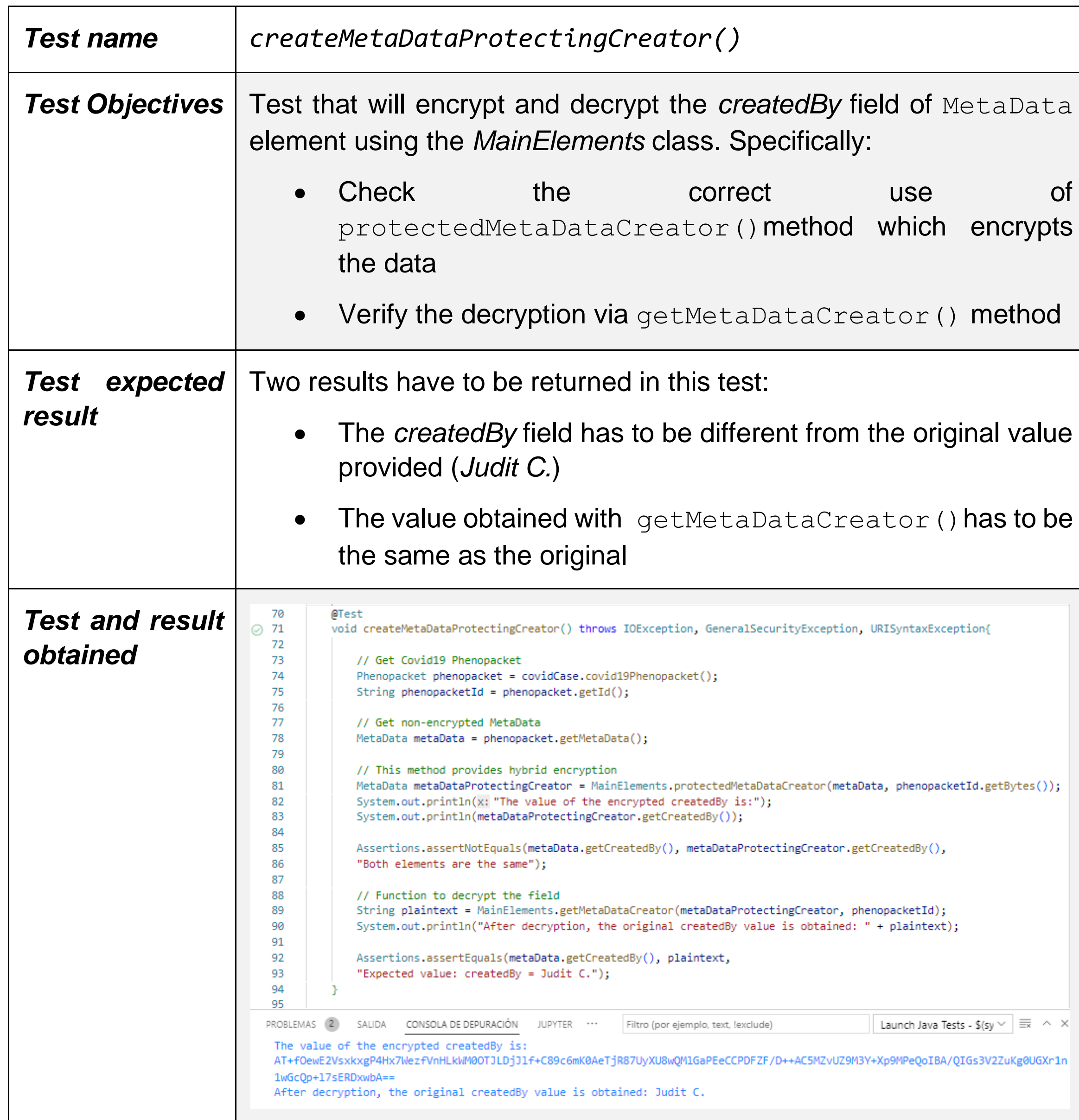
<table>
<tr><td>Test name</td><td>checkAgeDecryption()</td></tr>
<tr><td>Test Objectives</td><td>Test that will decrypt the Age block using the HybridEncription class. Specifically:
• Check the decryption through <code>getAge()</code> method</td></tr>
<tr><td>Test expected result</td><td>The age value in iso8601 format being P70Y</td></tr>
<tr><td>Test and result obtained</td><td>
</td></tr>
</table>

Table 6 Test to verify Age decryption

A test to check the encryption of Phenopacket creator:



Test name	*`createMetaDataProtectingCreator()`*
Test Objectives	Test that will encrypt and decrypt the *createdBy* field of `MetaData` element using the *MainElements* class. Specifically: • Check the correct use of `protectedMetaDataCreator()` method which encrypts the data • Verify the decryption via `getMetaDataCreator()` method
Test expected result	Two results have to be returned in this test: • The *createdBy* field has to be different from the original value provided (*Judit C.*) • The value obtained with `getMetaDataCreator()` has to be the same as the original
Test and result obtained	(code and console output below)

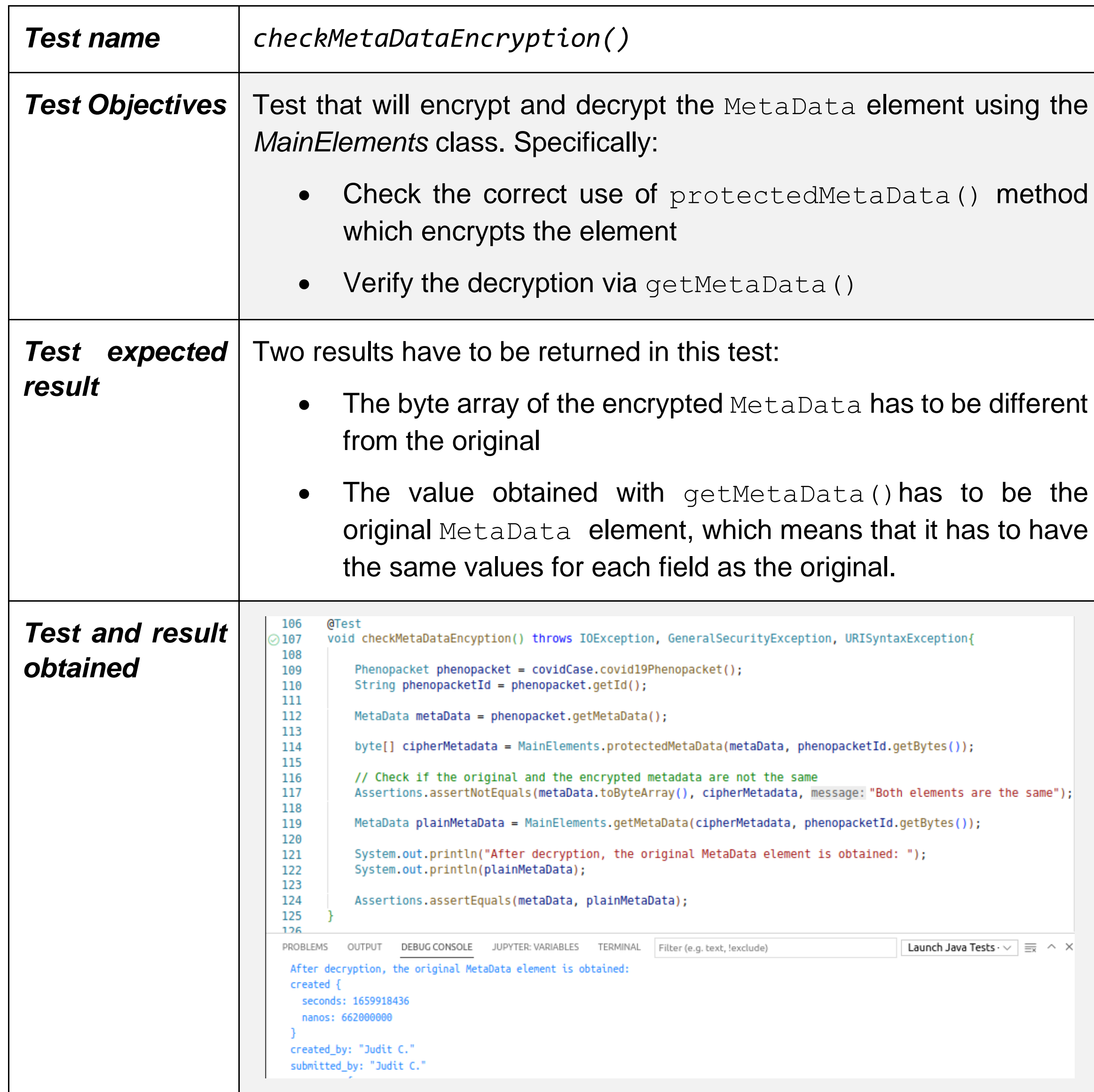
```
70      @Test
71      void createMetaDataProtectingCreator() throws IOException, GeneralSecurityException, URISyntaxException{
72
73          // Get Covid19 Phenopacket
74          Phenopacket phenopacket = covidCase.covid19Phenopacket();
75          String phenopacketId = phenopacket.getId();
76
77          // Get non-encrypted MetaData
78          MetaData metaData = phenopacket.getMetaData();
79
80          // This method provides hybrid encryption
81          MetaData metaDataProtectingCreator = MainElements.protectedMetaDataCreator(metaData, phenopacketId.getBytes());
82          System.out.println(x: "The value of the encrypted createdBy is:");
83          System.out.println(metaDataProtectingCreator.getCreatedBy());
84
85          Assertions.assertNotEquals(metaData.getCreatedBy(), metaDataProtectingCreator.getCreatedBy(),
86          "Both elements are the same");
87
88          // Function to decrypt the field
89          String plaintext = MainElements.getMetaDataCreator(metaDataProtectingCreator, phenopacketId);
90          System.out.println("After decryption, the original createdBy value is obtained: " + plaintext);
91
92          Assertions.assertEquals(metaData.getCreatedBy(), plaintext,
93          "Expected value: createdBy = Judit C.");
94      }
95
```

PROBLEMAS 2 SALIDA CONSOLA DE DEPURACIÓN JUPYTER ··· Filtro (por ejemplo, text, !exclude) Launch Java Tests - $(sy

```
The value of the encrypted createdBy is:
AT+fOewE2VsxkxgP4Hx7WezfVnHLkWM0OTJLDjJ1f+C89c6mK0AeTjR87UyXU8wQMlGaPEeCCPDFZF/D++AC5MZvUZ9M3Y+Xp9MPeQoIBA/QIGs3V2ZuKg0UGXr1n
1wGcQp+l7sERDxwbA==
After decryption, the original createdBy value is obtained: Judit C.
```

Table 7 Test to verify CreatedBy field encryption and decryption

A test to check the encryption of `MetaData` element:

Test name	*`checkMetaDataEncryption()`*
Test Objectives	Test that will encrypt and decrypt the `MetaData` element using the *MainElements* class. Specifically: • Check the correct use of `protectedMetaData()` method which encrypts the element • Verify the decryption via `getMetaData()`
Test expected result	Two results have to be returned in this test: • The byte array of the encrypted `MetaData` has to be different from the original • The value obtained with `getMetaData()` has to be the original `MetaData` element, which means that it has to have the same values for each field as the original.
Test and result obtained	

```java
106    @Test
107    void checkMetaDataEncyption() throws IOException, GeneralSecurityException, URISyntaxException{
108
109        Phenopacket phenopacket = covidCase.covid19Phenopacket();
110        String phenopacketId = phenopacket.getId();
111
112        MetaData metaData = phenopacket.getMetaData();
113
114        byte[] cipherMetadata = MainElements.protectedMetaData(metaData, phenopacketId.getBytes());
115
116        // Check if the original and the encrypted metadata are not the same
117        Assertions.assertNotEquals(metaData.toByteArray(), cipherMetadata, message: "Both elements are the same");
118
119        MetaData plainMetaData = MainElements.getMetaData(cipherMetadata, phenopacketId.getBytes());
120
121        System.out.println("After decryption, the original MetaData element is obtained: ");
122        System.out.println(plainMetaData);
123
124        Assertions.assertEquals(metaData, plainMetaData);
125    }
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   JUPYTER: VARIABLES   TERMINAL   Filter (e.g. text, !exclude)   Launch Java Tests
After decryption, the original MetaData element is obtained:
created {
  seconds: 1659918436
  nanos: 662000000
}
created_by: "Judit C."
submitted_by: "Judit C."
```

Table 8 Test to verify MetaData encryption and decryption

A test to represent the process of encryption and decryption with a file:

<table>
<tr><td>Test name</td><td>getAndDecryptElementsFromFile()</td></tr>
<tr><td>Test Objectives</td><td>This test is the most complex one, its objective is to reproduce the process that would be carried out in case of encrypting the whole MetaData element. The process is as follows:
1. First, the Phenopacket is serialized to byte array and the MetaData element is encrypted
2. Both results are stored in a JSON file
3. The receiver would receive the file and extract both byte arrays, first deserialize the Phenopacket and then, decrypt the MetaData element
4. Finally, the age is decrypted to check that the internal encryptions also work correctly once the Phenopacket is stored in a file.</td></tr>
<tr><td>Test expected result</td><td>Several results have to be returned in this test:
• The correct encryption of both Phenopacket and MetaData elements
• The JSON file with the byte arrays of both elements stored
• The correct decryption of the retrieved byte array, in this case it would check the decryption of MetaData element and age</td></tr>
<tr><td>Test and result obtained</td><td>135 @Test
136 void getAndDecryptElementsFromFile() throws URISyntaxException, IOException, GeneralSecurityException, ParseException{
137
138 // Create Covid19 Phenopacket
139 Phenopacket phenopacket = covidCase.covid19Phenopacket();
140 String phenopacketId = phenopacket.getId();
141 // Save the Phenopacket and the encrypted MetaData
142 SecurePhenopacket.protectMetaData(phenopacket);
143
144 // Get Phenopacket bytes from file
145 byte[] phenopacketBytes = HybridEncryption.getCipherBytes(elementName: "Phenopacket", phenopacketId);
146
147 // Get encrypted MetaData
148 byte[] cipherMetadata = HybridEncryption.getCipherBytes(elementName: "Metadata", phenopacketId);
149
150 // Decrypt MetaData
151 MetaData plainMetaData = MainElements.getMetaData(cipherMetadata, phenopacketId.getBytes());
152
153 Phenopacket phenopacketFromFile = Phenopacket.parseFrom(phenopacketBytes);
154 TimeElement age = phenopacketFromFile.getSubject().getTimeAtLastEncounter();
155 String plainAge = BlockBuilder.getAge(age, phenopacketId.getBytes());
156
157 // Checks
158 Assertions.assertNotEquals(phenopacket.getMetaData().toByteArray(), cipherMetadata,
159 message: "The two elements are the same");
160 Assertions.assertEquals(phenopacket.getMetaData(), plainMetaData,
161 message: "Expected same MetaData element");
162 Assertions.assertEquals(covidCase.isoAge, plainAge,message: "Expected value isoAge = P70Y");
163 }
164 }

Proof of concept of the JSON file:

{} 17a1a6ad-2ea1-40ee-9308-1401fa096c0c.json ●
target > classes > {} 17a1a6ad-2ea1-40ee-9308-1401fa096c0c.json > ...
1 {
2 "Phenopacket":"CiQxN2ExYTZhZC0yZWExLTQwZWUtOTMwOC0xNDAxZmEwOTZjMGMSoAEKB1A0Mz
3 "Metadata":"AT+fOewEVpN3YU8rv0TEscHeHnGX4HPYiqU34pRHO9K2IBTjBmsOP0TL9VPCzvmhL
4 }</td></tr>
</table>

Table 9 Test to verify storage and decryption of Phenopacket elements

7.1.3. Digital signature tests

This class is implemented to test the methods developed in the *DigitalSignature* class using the Covid19 case. Specifically, a function has been created for testing the digital signature and checking if the stored signature in the file allows to verify the Phenopacket correctly.

Test name	*`checkDigitalSignature()`*
Test Objectives	Test that will check the digital signature implementation. Specifically: • Sign a Phenopacket • Store the signature along with the Phenopacket serialized to byte array in a file • Retrieve the signature • Check the verification with the signature and the Phenopacket
Test expected result	Two results have to be returned in this test: 1. A new signature has to be added in the files named `signatures.json` 2. A true value has to be returned if the verification works
Test passed?	DigitalSignatureTest 450ms checkDigitalSignature() 450ms

Test and result obtained	`23 @Test`
	`24 void checkDigitalSignature() throws IOException, URISyntaxException, ParseException, GeneralSecurityException{`
	`25`
	`26     Phenopacket covidPhenopacket = covidCase.covid19Phenopacket();`
	`27     String phenopacketId = covidPhenopacket.getId();`
	`28`
	`29     System.out.println("The unique identifier is :" + phenopacketId);`
	`30     // Function that signs the Phenopacket and saves the signature in a JSON file`
	`31     SecurePhenopacket.signPhenopacket(covidPhenopacket);`
	`32     // Function that retrieves the signature from the file and verifies it`
	`33     SecurePhenopacket.verifyPhenopacket(covidPhenopacket);`
	`34`
	`35 }`
	`36`
	PROBLEMAS 2 SALIDA CONSOLA DE DEPURACIÓN JUPYTER TERMINAL Filtro (por ejemplo, text, !exclude) Launch Java
	`The unique identifier is :17a1a6ad-2ea1-40ee-9308-1401fa096c0c`
	`Verified:true`
	The new signature has successfully been included in the JSON file.
	signatures.json X
	target › classes › signatures.json › ...
	`1 {`
	`2     "17a1a6ad-2ea1-40ee-9308-1401fa096c0c-Signature":"ARMtWgAwRQIgT9Ih8ZP+QWkvi8ul+IX`
	`3     "0fdf0d92-c1d0-4c89-b8c6-c3233db58496-Signature":"ARMtWgAwRQIhAL9W1\/WJetxFhMy1xf`
	`4     "17a1a6ad-2ea1-40ee-9308-1401fa096c0c":"CiQxN2ExYTZhZC0yZWExLTQwZWUtOTMwOC0xNDAxZ`
	`5     "0fdf0d92-c1d0-4c89-b8c6-c3233db58496":"CiQwZmRmMGQ5MiljMWQwLTRjODktYjhjNiljMzIzM`
	`6 }`

Table 10 Test to verify digital signature feature

7.1.4. Hashing tests

This class is implemented to test the methods developed in the *Hashing* class using the oncology case. In this case, a test has been created for each selected element to compute the hash, however just one will be described since the procedure performed is the same in all tests. In addition, a test to check if the method *`checkHash()`* has been implemented correctly is also included in this class. To demonstrate that tests were successfully passed, Figure 31 shows the response obtained by the program.

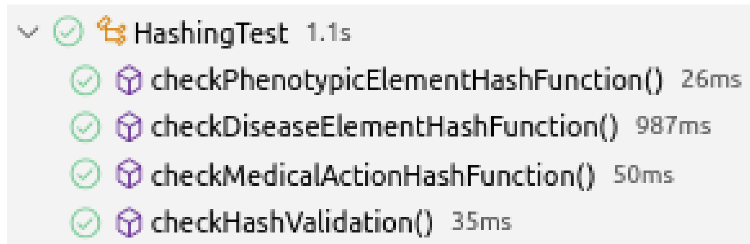


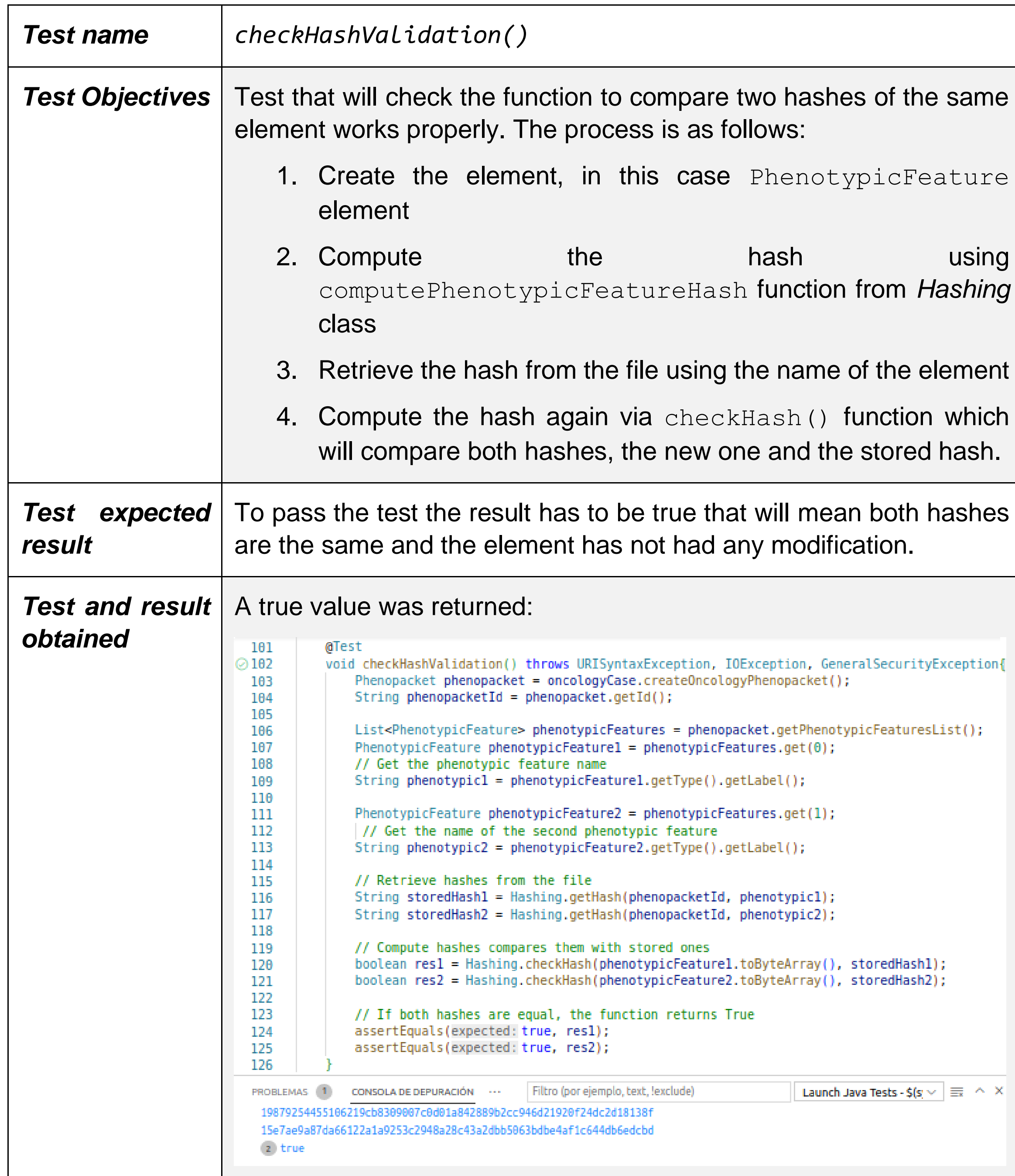
Figure 31 Tests created for hashing

The table below is an example of how it was checked the hashing of the `PhenotypicFeature` element. The same test process is created for the other two elements: `Disease` and `MedicalAction`.

Test name	*`checkPhenotypicELementHashFunction()`*
Test Objectives	Test that will check the hash function created using keccak256 without any error. The process is as follows: 1. Create the element, in this case `PhenotypicFeature` element 2. Compute the hash using `computePhenotypicFeatureHash()` function 3. The aforementioned function will save the hash in a file 4. Retrieve the hash from the file and compares with the last computed hash to check that the storage is done correctly
Test expected result	To pass the test a new file with the computed hashes has had to be created.
Test result obtained	The hashes values have successfully been included in the file together with the name of the element. f9f2d029-e1e3-42a4-bb79-ee39652c8c07.txt target › classes › f9f2d029-e1e3-42a4-bb79-ee39652c8c07.txt 1 Infiltrating Urothelial Carcinoma:f4664a2bc57b1866e003ab8b6368b9321788799172ad383f21d6d4f8fd242f8d 2 Hematuria:19879254455106219cb8309007c0d01a842889b2cc946d21920f24dc2d18138f 3 Dysuria:15e7ae9a87da66122a1a9253c2948a28c43a2dbb5063bdbe4af1c644db6edcbd 4

Table 11 Test to verify the hashing process

The last test created is to corroborate that the same element computes the same hash.



Test name	*checkHashValidation()*
Test Objectives	Test that will check the function to compare two hashes of the same element works properly. The process is as follows: 1. Create the element, in this case `PhenotypicFeature` element 2. Compute the hash using `computePhenotypicFeatureHash` function from *Hashing* class 3. Retrieve the hash from the file using the name of the element 4. Compute the hash again via `checkHash()` function which will compare both hashes, the new one and the stored hash.
Test expected result	To pass the test the result has to be true that will mean both hashes are the same and the element has not had any modification.
Test and result obtained	A true value was returned: (code below)

```
101    @Test
102    void checkHashValidation() throws URISyntaxException, IOException, GeneralSecurityException{
103        Phenopacket phenopacket = oncologyCase.createOncologyPhenopacket();
104        String phenopacketId = phenopacket.getId();
105
106        List<PhenotypicFeature> phenotypicFeatures = phenopacket.getPhenotypicFeaturesList();
107        PhenotypicFeature phenotypicFeature1 = phenotypicFeatures.get(0);
108        // Get the phenotypic feature name
109        String phenotypic1 = phenotypicFeature1.getType().getLabel();
110
111        PhenotypicFeature phenotypicFeature2 = phenotypicFeatures.get(1);
112         // Get the name of the second phenotypic feature
113        String phenotypic2 = phenotypicFeature2.getType().getLabel();
114
115        // Retrieve hashes from the file
116        String storedHash1 = Hashing.getHash(phenopacketId, phenotypic1);
117        String storedHash2 = Hashing.getHash(phenopacketId, phenotypic2);
118
119        // Compute hashes compares them with stored ones
120        boolean res1 = Hashing.checkHash(phenotypicFeature1.toByteArray(), storedHash1);
121        boolean res2 = Hashing.checkHash(phenotypicFeature2.toByteArray(), storedHash2);
122
123        // If both hashes are equal, the function returns True
124        assertEquals(expected: true, res1);
125        assertEquals(expected: true, res2);
126    }

PROBLEMAS 1   CONSOLA DE DEPURACIÓN ...   Filtro (por ejemplo, text, !exclude)   Launch Java Tests - $(s
198792544551062l9cb8309007c0d01a842889b2cc946d21920f24dc2d18138f
15e7ae9a87da66122a1a9253c2948a28c43a2dbb5063bdbe4af1c644db6edcbd
2 true
```

Table 12 Test to validate a hash

7.2. Phenotools-validator

To finish with the validations, it was decided to use a tool provided by GA4GH called Phenopacket-tools [37]. This tool provides a validator named *Phenotools-validator* that checks if a Phenopacket is correctly created or not by checking for missing elements that need to be specified in the schema or incorrectly defined fields. The only requirement for its use is that the schema is provided in JSON format.

In this case, the schema obtained by means of the `ExportPhenopacket()` function was used, as detailed in section 6.3.2. Then, in order to perform such validation, the following command must be executed:

```
$ pfx-tools validate path_to_jsonFile
```

where `path_to-jsonFile` is the location of the Phenopacket JSON file that wants to be validated.

Finally, as Figure 32 shows, a positive result indicated with an "OK" was obtained. It is important to note that the tool has a method to check the age field and although the schema used had this value encrypted, it could be validated correctly.

```
judit@judit-VirtualBox:~/phenopacket-tools$ pfx-tools validate
17a1a6ad-2ea1-40ee-9308-1401fa096c0c.json
17a1a6ad-2ea1-40ee-9308-1401fa096c0c.json - OK
```

Figure 32 Phenopacket has been successfully validated

8. Conclusions and future development

Over the past few years, medical data sharing has been essential, not only because such data can help with disease research, but also because it makes it possible to share new treatments for rare diseases.

Organizations such as Global Alliance for Genomic and Health are working on a daily basis to improve standardized data sharing around the world. Among the many standards that they have developed is Phenopackets, a new file format designed to exchange phenotypic information between researchers and healthcare personnel. However, this format provides no mechanism to ensure data privacy.

This is a challenge because, as the beginning of this document shows, the healthcare sector is one of the most popular targets for attacks that lead to a data breach in their environment. The reason for this is that the data used is treated as protected health information and can be monetized by attackers after it has been stolen.

All these factors led to the study of the Phenopacket schema in order to add security features that would allow the information to be exchanged securely, preventing an attacker from accessing confidential information in the event an attacker takes it.

In the first part of the project, a study and analysis of the schema that defines the Phenopacket was carried out. Once its structure was known, it was decided to create a reduced schema with the aim of being a format that would allow the exchange of new treatments and improve the patient's quality of life. In addition, when designing the schema, it was also considered which fields could include sensitive data and were completely discarded for the project's schema unless they were a requirement of the schema itself.

Moreover, when dealing with so many different fields, a classification was needed to differentiate which data presented a higher risk in a data leak, differentiating between two types of data: Personally identifiable Information (PII) and Protected Health Information (PHI). PII is sensitive information that identifies an individual, being the Age and karyotypic sex fields the ones included. Regarding PHI, it was found that the data included differed by countries, however, it was concluded that all data that could be part of a medical record and whose information could be relevant to a patient's re-identification would be categorized as PHI.

At the same time, the data were also divided into two categories: restricted and confidential. This distinction made it possible to apply different levels of security according to the assigned category, in this case, confidential data did not pose a risk to the individual, yet the sensitivity of the data required minimal security of application. In contrast, PHI assigned as restricted data required a higher level of protection. This process was not easy, as many of the data processed were unknown and there was not enough information about how to classify them.

To conclude the research part, an investigation was conducted to select the techniques that will guarantee the confidentiality, integrity, and authentication of the schema. This research resulted in the three security features applied in the project, consisting of encryption algorithms to ensure the privacy of the restricted data, a hash function to provide integrity of the confidential data and, lastly, digital signature to authenticate the creator of the Phenopacket.

However, the choices of algorithm within these techniques are quite extensive, so the use and reputation of the algorithm in the security community was considered when choosing a particular algorithm. Based on the information gathered, the best approach for encryption was to use hybrid encryption that gets the best of both types of encryptions. Meanwhile, the digital signature chosen was ECDSA, since it provided greater security than others currently available, and for hashing Keccack256 was the selected one, being the function used by several blockchain projects.

Based on the exposed, a second phase was performed, which led to the development of the security mechanism with the objective of providing security to the Phenopackets schema for its exchange.

The development process was guided by an Agile Scrum methodology, which allowed each feature to be implemented individually along with the tests and variations that were necessary as the project progressed. First of all, the Phenopacket was defined using Protobuf format, which in turn exported the schema into different Java classes that were later used to build the project. The fact that the classes were created automatically meant learning the methods involved in those classes as the security features were designed. Finally, the project of this Master's thesis was created in a Java environment that includes the designed schema as well as the security features with the verification tests that allowed it to check its correct operation.

Despite being a first version with basic features to fulfill the objectives, it has been observed that it allows the Phenopacket to be exchanged securely, providing privacy and authenticity to the format. However, it can be said that it lacks certain functionalities such as allowing the user to design the Phenopacket directly through a terminal.

In the course of developing the project, several obstacles were also encountered, spending more time on certain tasks than expected. On the one hand, there was a problem in protecting the karyotypic sex of the individual, which was defined as an enumeration of fixed values and could not be modified. Consequently, it was not possible to apply the agreed level of security, in this case encryption, concluding not to protect the field since the information it provided was not commonly known as gender or sex. On the other hand, not all the security mechanisms considered in the early stages of the project could be applied, either because of the structure of the Phenopacket itself or because of the final scope of the project. However, the schema has been designed in such a way that any other element not proposed in the classification may also be protected by the mechanisms developed if requested by the user.

Furthermore, it would have been convenient to improve the storage of files and key sets, since to facilitate implementation, they were stored in the project's resource folder, being a potential threat to the privacy of the user if the project falls into malicious users.

It would also have been interesting to develop a web application that would allow interaction with the user either to customize the format according to his/her requirements or to choose more easily the technique he wishes to use to send the data securely. Unfortunately, time constraints did not allow to investigate this line of work and therefore it remains as a possible future work to be done.

It can be concluded that this project can be a first step in improving data exchange, where it has been demonstrated that a variety of security techniques can be used to ensure the confidentiality, integrity, and authenticity to the exchange of phenotypic information, turning the Phenopackets into a more secure file format.

Bibliography

[1] "World Health Organization," 11 03 2020. [Online]. Available: https://www.who.int/director-general/speeches/detail/who-director-general-s-opening-remarks-at-the-media-briefing-on-covid-19---11-march-2020. [Accessed 09 August 2022].

[2] WHO, "Coronavirus Disease (COVID-19) pandemic," World Health Organization, [Online]. Available: https://www.who.int/europe/emergencies/situations/covid-19. [Accessed 09 August 2022].

[3] World Health Organization, "Laboratory testing strategy recommendations for COVID-19: interim guidance," World Health Organization, 21 March 2020. [Online]. Available: https://apps.who.int/iris/handle/10665/331509.

[4] World Health Organization, "Launch of the WHO Academy and the WHO Info mobile applications," World Health Organization, 13 May 2020. [Online]. Available: https://www.who.int/news/item/13-05-2020-launch-of-the-who-academy-and-the-who-info-mobile-applications.

[5] WHO, "Listings of WHO's response to COVID-19," World Health Organization, 29 01 2021. [Online]. Available: https://www.who.int/es/news/item/29-06-2020-covidtimeline. [Accessed 09 August 2022].

[6] T. Hulsen, "Sharing Is Caring—Data Sharing Initiatives in Healthcare," *International Journal of Environmental Research and Public Health,* vol. 17, p. 3046, 2020.

[7] World Health Organization, "WHO reports fivefold increase in cyber attacks, urges vigilance," World Health Organization, 23 April 2020. [Online]. Available: https://www.who.int/news/item/23-04-2020-who-reports-fivefold-increase-in-cyber-attacks-urges-vigilance. [Accessed 11 August 2022].

[8] "Attacks targeting healthcare organizations spike globally as COVID-19 cases rise again," 2021. [Online]. Available: https://blog.checkpoint.com/2021/01/05/attacks-targeting-healthcare-organizations-spike-globally-as-covid-19-cases-rise-again/. [Accessed 08 July 2022].

[9] "Cost of a data breach Report," 2021. [Online]. Available: https://www.ibm.com/downloads/cas/OJDVQGRY. [Accessed 08 July 2022].

[10] "X-Force Threat Intelligence 2022: Healthcare executive summary," 2022. [Online]. Available: https://www.ibm.com/downloads/cas/GLPOVRLP. [Accessed 07 July 2022].

[11] “Health Sector Cybersecurity: 2021 Retrospective and 2022 Look Ahead,” [Online]. Available: https://www.hhs.gov/sites/default/files/2021-retrospective-and-2022-look-ahead-tlpwhite.pdf. [Accessed 07 July 2022].

[12] Verizon, "4 Industries Report," 2022. [Online]. Available: https://www.verizon.com/business/resources/reports/dbir/2022-data-breach-investigations-report-dbir-industries.pdf. [Accessed 10 July 2022].

[13] S. Alder, "March 2022 Healthcare Data Breach Report," 2022. [Online]. Available: https://www.hipaajournal.com/march-2022-healthcare-data-breach-report/. [Accessed 07 July 2022].

[14] "agencia española protección datos," aepd, [Online]. Available: https://www.aepd.es/es. [Accessed 03 June 2022].

[15] "Trustwave global security report," 2018. [Online]. Available: https://trustwave.azureedge.net/media/15350/2018-trustwave-global-security-report-prt.pdf?rnd=131992184400000000. [Accessed 09 July 2022].

[16] "Phenopackets: Standardizing and Exchanging Patient Phenotypic Data," GA4GH, 2019. [Online]. Available: https://www.ga4gh.org/news/phenopackets-standardizing-and-exchanging-patient-phenotypic-data/. [Accessed 01 March 2022].

[17] "Genomics informatics — Phenopackets: A format for phenotypic data exchange," ISO/DIS 4454(en), 2021. [Online]. Available: https://www.iso.org/obp/ui/#iso:std:iso:4454:dis:ed-1:v1:en.

[18] ISO, “Genomics informatics — Phenopackets: A format for phenotypic data exchange,” ISO 4454:2022, 06 July 2022. [Online]. Available: https://www.iso.org/standard/79991.html. [Accessed 12 August 2022].

[19] A. A. G. A. C. A. A. E. B. M. A. D. R. E. H. G. L. G. T. M. S. T. S. G. M. R. D. 1. G. P. A. G. Petr Danecek, "The variant call format and VCFtools," in *Bioinformatics*, vol. 27, 2021, p. 2156–2158.

[20] medRxiv, "The GA4GH Phenopacket schema: A computable representation of clinical data for precision medicine," 2021. [Online]. Available: https://doi.org/10.1101/2021.11.27.21266944 .

[21] "PhenoTips," Gene42 Inc., 2015. [Online]. Available: https://phenotips.com/. [Accessed 07 July 2022].

[22] "Phenopackets," [Online]. Available: http://phenopackets.org/. [Accessed 09 July 2022].

[23] "Phenopacket-schema Documentation," Global Alliance for Genomics and Health, 2021. [Online]. Available: https://phenopacket-schema.readthedocs.io/en/v2/. [Accessed 03 March 2022].

[24] "Human phenotype ontology," [Online]. Available: https://hpo.jax.org/app/. [Accessed 07 July 2022].

[25] "Health Insurance Portability and Accountability Act of 1996," PUBLIC LAW 104-191, 1996.

[26] GDPR, "General Data Protection Regulation," 2018. [Online]. Available: https://gdpr-info.eu/. [Accessed 07 July 2022].

[27] S. Harvey, "Classifying Data: Why It's Important and How To Do It," KirkpatrickPrice, 2020. [Online]. Available: https://kirkpatrickprice.com/blog/classifying-data/. [Accessed 07 July 2022].

[28] "5 Types of Data Classification," Indeed, 2021. [Online]. Available: https://www.indeed.com/career-advice/career-development/data-classification-types. [Accessed 01 July 2022].

[29] HIPAA Security Series, "4 Security Standards: Technical Safeguards," 2007. [Online]. Available: https://www.hhs.gov/sites/default/files/ocr/privacy/hipaa/administrative/securityrule/techsafeguards.pdf?language=es. [Accessed 09 July 2022].

[30] "Keccak-256 Hash Function," MoneroDocs, 2022. [Online]. Available: https://monerodocs.org/cryptography/keccak-256/. [Accessed 15 August 2022].

[31] B. I. G. Seroussi, G. Seroussi and N. Smart, Elliptic curves in cryptography, vol. 265, Cambridge university press, 1999.

[32] CISA, "Security Tip (ST04-018)," 2020. [Online]. Available: https://www.cisa.gov/uscert/ncas/tips/ST04-018. [Accessed 09 July 2022].

[33] "Protocol-buffers," Google, 2008. [Online]. Available: https://developers.google.com/protocol-buffers/. [Accessed 07 July 2022].

[34] "Protocol Buffers - Google's data interchange format," GitHub, 2019. [Online]. Available: https://github.com/protocolbuffers/protobuf. [Accessed 15 June 2022].

[35] "Tink," Google Developers, [Online]. Available: https://developers.google.com/tink. [Accessed 30 August 2022].

[36] "Bouncy Castle Crypto APIs," Legion of the Bouncy Castle, 2013. [Online]. Available: https://bouncycastle.org/. [Accessed 30 August 2022].

[37] "Phenopacket-tools," GitHub, 2022. [Online]. Available: https://github.com/phenopackets/phenopacket-tools. [Accessed 20 August 2022].

[38] G. N. Colfax, "Confidentiality and Privacy in Healthcare," Department of Health and Human Services, 2017.

[39] "Phenopackets v2.0 expands utility to provide a more complete medical picture," GA4GH, 2022. [Online]. Available: https://www.ga4gh.org/news/phenopackets-v2-expands-utility-to-provide-a-more-complete-medical-picture/. [Accessed 03 March 2022].

[40] S. W. A. D. Ben Lutkevich, "Protected health information (PHI) or personal health information," SearchHealthIT, 2021. [Online]. Available: https://www.techtarget.com/searchhealthit/definition/personal-health-information. [Accessed 08 July 2022].

[41] "Repository for the GA4GH phenopacket schema," GitHub, 2018. [Online]. Available: https://github.com/phenopackets/phenopacket-schema. [Accessed 03 March 2022].

[42] T. Pornin, "RFC 6979. Deterministic DSA and ECDSA," IETF Trust, August 2013. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6979.

[43] "Keccak," keccak.team, 2022. [Online]. Available: https://keccak.team/keccak.html. [Accessed 15 August 2022].

[44] S. Nakov, "Digital Signatures," Practical Cryptography for Developers, 2018. [Online]. Available: https://cryptobook.nakov.com/digital-signatures. [Accessed 10 August 2022].

Appendices

A. Phenopacket data model

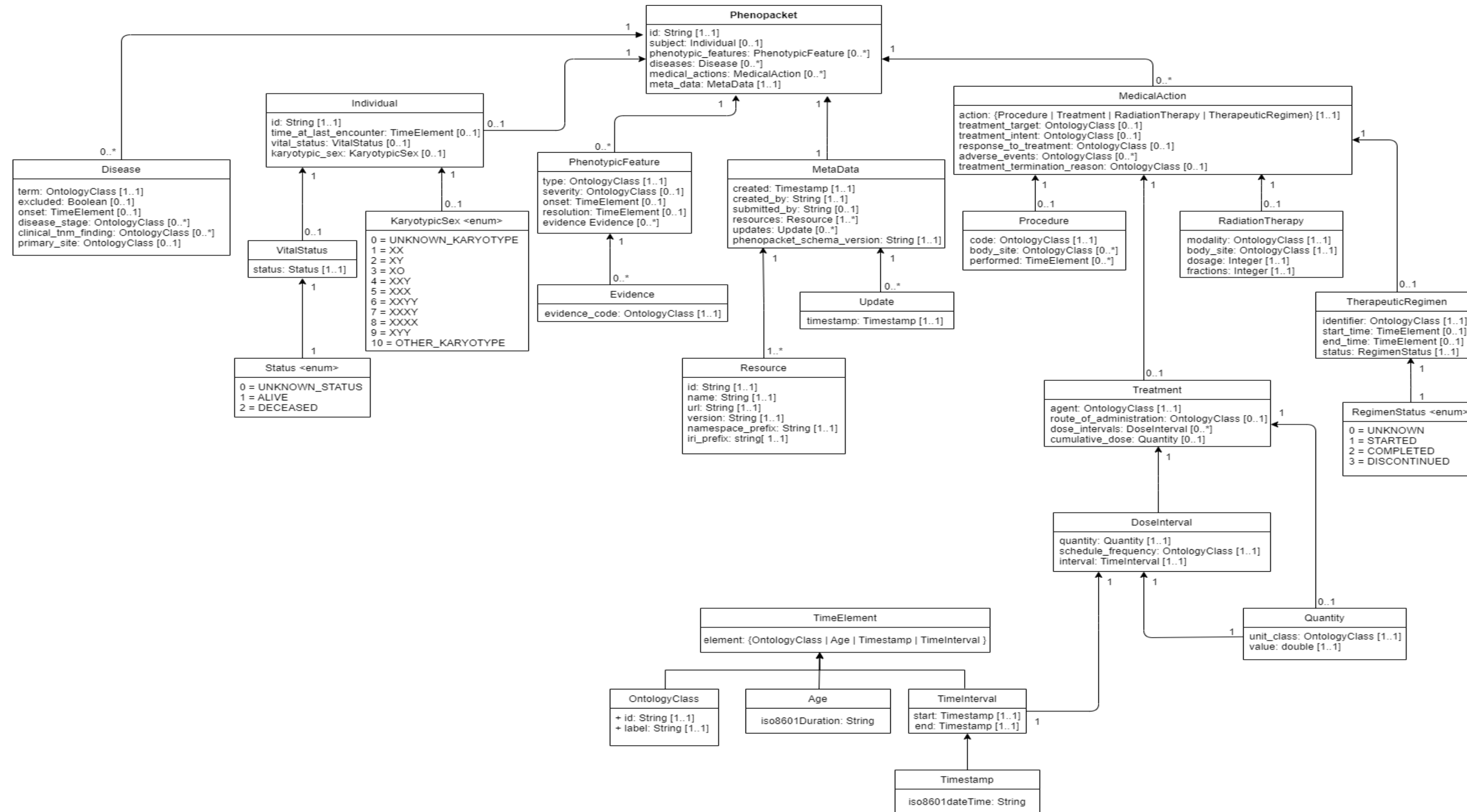


Figure 33 Phenopacket schema diagram

B. Project folder structure

The project consists of several folders to organize the classes involved in the Phenopacket implementation of the classes in charge of the security features. Specifically, it can be differentiated between the **src** folder and the **test** folder, the first one is composed of the different classes implemented during the development phase, while the test folder contains the different verification tests created to check the features created.

In the *src* folder it can be found:

- **Phenopacket** folder – contains the *SecurePhenopacket* class in which the creation of the whole Phenopacket is implemented
- **Phenopacket/examples** folder – includes two Phenopackets cases for use during testing
- **Phenopacket/schema** folder – there are all the methods to create the blocks and elements that make up the Phenopacket
- **Phenopacket/securityMechanisms** folder – three security mechanisms as well as the *ExternalResources* class are included in this folder

Then, the *test* folder has the same structure to facilitate its understanding. In this case, the following folders are:

- **/Phenopacket/phenopacket_schema** folder – tests associated with creating elements and Phenopacket are located in this folder
- **/Phenopacket/securityMechanisms** folder – all the created verification test are implemented here

To help the reader understand this description, the following figure represents the complete project structure:

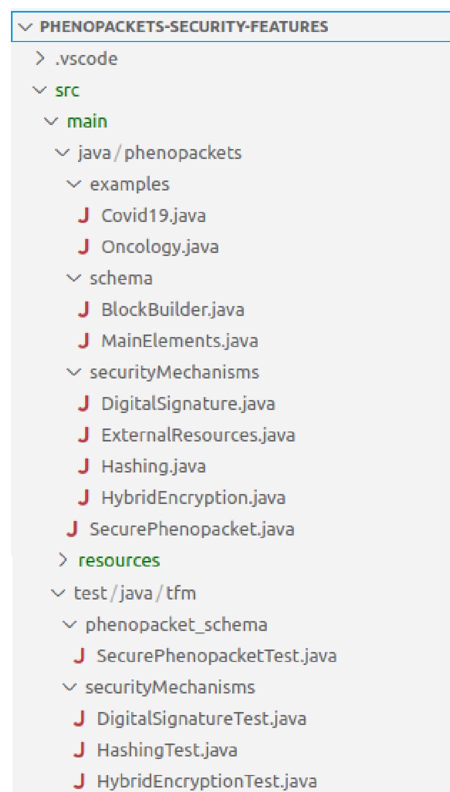


Figure 34 Project folder structure

C. Covid19 example


```
{
  "id": "17a1a6ad-2ea1-40ee-9308-1401fa096c0c",
  "subject": {
    "id": "P172062",
    "timeAtLastEncounter": {
      "age": {
        "iso8601duration":
"AQD5SysEPd7+EDI7Cf9vtMJQZyJfjD+LVR6hk72iqHVW/tFS5e325FKwtyjy/Db7rcwABeeSFd8HsF
G/z5HGSAvXtaP3l3vlFpYiFUnH+TJ0iTI0MGPdg9PApKhTALfin2mnuVlZ"
      }
    },
    "vitalStatus": {
      "status": "DECEASED"
    },
    "karyotypicSex": "XY"
  },
  "phenotypicFeatures": [{
    "type": {
      "id": "NCIT:C27009",
      "label": "Myalgia"
    },
    "severity": {
      "id": "HP:0012828",
      "label": "Severe"
    },
    "onset": {
      "timestamp": "2020-03-18T00:00:00Z"
    },
    "resolution": {
      "timestamp": "2020-03-20T00:00:00Z"
    },
    "evidence": [{
      "evidenceCode": {
        "id": "ECO:0006017",
        "label": "author statement from published clinical study used in manual
assertion"
      }
    }]
  }, {
    "type": {
      "id": "NCIT:C2998",
      "label": "Dyspnea"
    },
    "severity": {
      "id": "HP:0012828",
      "label": "Severe"
```

```
    },
    "onset": {
      "timestamp": "2020-03-18T00:00:00Z"
    },
    "resolution": {
      "timestamp": "2020-03-20T00:00:00Z"
    },
    "evidence": [{
      "evidenceCode": {
        "id": "ECO:0006017",
        "label": "author statement from published clinical study used in manual
assertion"
      }
    }]
  }],
  "diseases": [{
    "term": {
      "id": "NCIT:C2985",
      "label": "Diabetes Mellitus"
    },
    "excluded": true,
    "onset": {
      "age": {
        "iso8601duration":
"AQD5SysECZuvhyYNhbrsxzgSIR8i9C38nXAb3IpU76OhLg8URhW9qyEuJIry5rtwPDQ0qBoDhSqNZ8
M+kEa4D/kOoCz6XLTBjjgu7bqJWZLDhq4xjRC+eky0y1uVsZcgNWMci/LU"
      }
    },
    "diseaseStage": [{
      "id": "NCIT:C27971",
      "label": "Stage IV"
    }],
    "clinicalTnmFinding": [],
    "primarySite": {
      "id": "UBERON:0000948",
      "label": "heart"
    }
  }, {
    "term": {
      "id": "NCIT:C34830",
      "label": "Cardiomyopathy"
    },
    "excluded": false,
    "onset": {
      "age": {
        "iso8601duration":
"AQD5SysECZuvhyYNhbrsxzgSIR8i9C38nXAb3IpU76OhLg8URhW9qyEuJIry5rtwPDQ0qBoDhSqNZ8
M+kEa4D/kOoCz6XLTBjjgu7bqJWZLDhq4xjRC+eky0y1uVsZcgNWMci/LU"
```

```
        }
      },
      "diseaseStage": [{
        "id": "NCIT:C27971",
        "label": "Stage IV"
      }],
      "clinicalTnmFinding": [],
      "primarySite": {
        "id": "UBERON:0000948",
        "label": "heart"
      }
    }],
    "medicalActions": [{
      "procedure": {
        "code": {
          "id": "NCIT:C80473",
          "label": "Left Ventricular Assist Device"
        },
        "bodySite": {
          "id": "UBERON:0000948",
          "label": "heart"
        },
        "performed": {
          "timestamp": "2016-01-01T00:00:00Z"
        }
      },
      "adverseEvents": []
    }, {
      "treatment": {
        "agent": {
          "id": "NCIT:C722",
          "label": "Oxygen"
        },
        "routeOfAdministration": {
          "id": "NCIT:C38284",
          "label": "Nasal Route of Administration"
        },
        "doseIntervals": [{
          "quantity": {
            "unit": {
              "id": "NCIT:C67388",
              "label": "Liter per Minute"
            },
            "value": 2.0
          },
          "scheduleFrequency": {
            "id": "NCIT:C64597",
            "label": "Immediately"
```

```
        },
        "interval": {
          "start": "2020-03-20T00:00:00Z",
          "end": "2020-03-22T00:00:00Z"
        }
      }]
    },
    "adverseEvents": []
  }],
  "metaData": {
    "created": "2022-08-08T00:27:16.662Z",
    "createdBy": "Judit C.",
    "submittedBy": "Judit C.",
    "resources": [{
      "id": "ncit",
      "name": "NCI Thesaurus OBO Edition",
      "url": "http://purl.obolibrary.org/obo/ncit.owl",
      "version": "http://purl.obolibrary.org/obo/ncit/releases/2019-11-
26/ncit.owl",
      "namespacePrefix": "NCIT",
      "iriPrefix": "http://purl.obolibrary.org/obo/ncit.owl"
    }],
    "updates": [{
      "timestamp": "2022-08-08T00:27:16.662Z"
    }],
    "phenopacketSchemaVersion": "2.0"
  }
}
```