IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661, p-ISSN: 2278-8727, Volume 22, Issue 1, Ser. II (Jan - Feb 2020), PP 57-68
www.iosrjournals.org

Object-Oriented Cryptography Hierarchy Realizations

Tony Karavasilev[1], Svetoslav Enkov[2]
[1](*Computer informatics, Plovdiv University "Paisii Hilendarski", Bulgaria)*
[2](*Computer informatics, Plovdiv University "Paisii Hilendarski", Bulgaria)*

Abstract: *The role of cryptography has become a major part of every modern software and network architecture. Most programming languages, software frameworks and application libraries do not provide a cryptography hierarchy and do not even implement a basic cryptography model, which makes the development of a software system or communication network even harder from a security point of view. This paper presents an object-oriented solution for providing all the main classifications of cryptography algorithms via a hierarchy of classes that offer frequently used formats and security features for both in-transit and at-rest data security. It also presents the important role of all three main types of cryptographic algorithms for the security and development of any state-of-the-art system. The purpose of the developed and published online practical realization of a cryptography hierarchy, for a language that has no embedded cryptography model, is to emphasize and show the need for such components and their usage for authentication, integrity verification, secure storage and safe data transfers. This article defines the main building blocks of the further creation of a secure object-oriented cryptography model for standardized security techniques and protocols in any programming language.*
Keywords*: cryptography, security, object-oriented hierarchy, cryptography model, at-rest, in-transit, network communication, verification, identification, authentication, authorization, data, hash functions, symmetric encryption, asymmetric encryption, digital signature, authenticated encryption, digital envelope, key exchange, message digestion, digest, salt, secret key, key management, initialization vector, block, mode of operation, padding standard, public key, private key, key pair, MAC, HMAC, KDF, HKDF, PBKDF2, MD5, SHA-1, SHA-2, SHA-3, Whirlpool, Argon2, Bcrypt, AES, CAMELLIA, RSA, DSA.*

Date of Submission: 28-01-2020 Date of Acceptance: 13-02-2020

I. Introduction

The classical cryptography definition was confined to the art of designing and breaking of algorithms for message encryption or data hiding via sophisticated secret code schemes. In contrast, modern cryptography or cryptology is the practice and study of techniques for secure communication over an insecure channel, secure storage or privacy problems and the prevention of illegal data alteration. In general, this consists of the construction and the analysis of security protocols to prevent third parties from stealing, manipulating or reading data both in-transit and at-rest. [1]

The practical applications of cryptography include securing computer passwords, sensitive information storage, data integrity checks, system authentication, high-quality pseudo-random data generators, key exchange algorithms, digital signatures, chip-based payments, digital rights management, data erasure methods, military-grade secure network communications and cryptocurrencies. Cryptography has become a major irreplaceable tool for the information technology sector in the building of software applications, hardware systems and network communications. The best cyber security practices completely rely on the current state of the established modern cryptography primitives, protocols and techniques. [2]

In modern cryptography, a cryptosystem is a suite or combination of cryptographic algorithms needed to implement a particular security service that achieves confidentiality, integrity, non-repudiation and/or authentication. There are three main types of cryptographic algorithms: [3]

- Hash functions;
- Symmetric key ciphers;
- Asymmetric key ciphers.

The strong definition of a hash function is any one-way function that can map data of arbitrary size to fixed-size values in a certain domain. A cryptographic hash function, on the other hand, is a cryptography suitable function that does not allow duplicated output values for different input data and the function result must always be practically infeasible to invert back or recover to its initial raw state. The output of hash functions is a hash value or message digest. Frequently, the usage of hash functions is combined with salting techniques that consist of prepending or appending long strings to the input message to make it longer before processing. This reduces the risk of some well-known attacks like brute-force guessing or searching pre-generated digest lookup tables. There are certain subtypes of hash functions that are parameterized to provide greater security and include the usage of one or more parameters like a secret key, a context string or an algorithmic cost value. They are mostly used for secure message authentication, password hashing, key generation, key expansion and hardware-resistant or time-consuming input processing. Cryptographic hash functions have many security applications like integrity checks, message authentication, confidential password storage and digital signature digestion. The most famous algorithms from this type are MD5, SHA-1, Whirlpool, Bcrypt and Argon2.

In contrast, symmetric key algorithms are designed to hide a message via the process of encryption with the option to retrieve it later via the process of decryption but only if we have the chosen secret key used by the current cryptosystem. In other words, they use the same cryptographic key for both the encryption of plain data and the decryption of cipher data. The most important thing is that the key must always be kept a secret between all conversation participants, needs to be agreed in advance and to be as complex as possible. Some of these algorithms can only process a message as a whole with exactly one pass and are classified as stream ciphers. With the development of modern hardware, these algorithms have become obsolete and insecure. The most famous algorithms of this subtype are historical Caesar and Vigenère ciphers. On the other hand, most of the modern algorithms process data by dividing it into chunks of fixed size that are called blocks. This is the reason why they are classified as block ciphers. This quality opens the ability to include different manipulations and operations between the blocks and add an extra initialization vector block that can play the role of a semantic second secret key. If we do not have both the initialization vector, secret key and the block mode of operation processing logic, then we can not retrieve a message encrypted with this type of algorithm. They also have a need of filling up the last block of data to the fixed chunk size via padding of extra bytes. The simplest example is padding the last block with null bytes or zeroes until the fixed length is reached. Of course, there are other more secure ways of doing padding like the PKCS standards. The most famous block ciphers are AES (Advanced Encryption Standard), 3DES (Triple DES) and DES (Data Encryption Standard). These cryptographic symmetric algorithms are mainly used for data encryption purposes but are also frequently used with message authentication codes (MAC). The practical application includes the encryption of sensitive data, files, databases and whole disk storage devices.

The asymmetric cipher cryptography type also provides a way to encrypt and decrypt data like the symmetric type. The special property of this type of cryptosystem is that it uses two different keys, one for encryption and one for decryption. The key for encryption is called a public key and can be known and publicly available to the whole world. On the other hand, the decryption key must be kept a total secret and is called a private key. It is important to note that this type of cryptography allows one private key to have multiple derived public keys without breaking the security of the whole cryptosystem. In addition, this type provides a way of generating secure digital signatures via the usage of hash functions and the inversion of the semantical role of the two keys. This is done via the generation of a message digest, the encryption of the produced hash value with the private key that you possess, the ability of all authorized institutions to decrypt the value with their personal public keys and generate a fresh digest of the received message to compare it with to the obtained one. This is only possible because both encryption and decryption are done via exponentiation and modulus calculations, thus the roles of the keys may be swapped for digital signature generation purposes only. The usage of hash functions here is important because it protects the digest while in-transit and can securely prove that the received message data has not been tampered with. Most of these algorithms are used for key exchange algorithms, secret key encryption and digital signature generation. The biggest disadvantage of asymmetric algorithms is that they are limited to processing information smaller than their key size, in contrast with the unlimited processing power of symmetric ciphers. Note that this type of algorithm also uses data padding but only to the used key size. The most famous algorithms here are RSA (Rivest–Shamir–Adleman), DSA (Digital Signature Algorithm) and DH (Diffie–Hellman Key Exchange).

It is important to note that the Transport Layer Security (TLS) and the Secure Sockets Layer (SSL) network protocols take advantage of all three subtypes of cryptography. The asymmetric algorithms are used for key exchange and the safe transport of a secure pseudo-random secret key for symmetric ciphers. The symmetric algorithms are then used to encrypt all traffic send between both sides and the hash functions are used for integrity checks and authentication purposes. Note that, those two communication network protocols also ensure the usage of

the most secure cryptographic algorithms supported by each individual client and take care of compatibility issues per connection. [4] [6]

In addition, many modern protocols combine the usage of cryptography and steganography techniques. Steganography consists of the concealing of a message or a file within another message or a file that is both publicly accessible and clearly visible, hiding the true message in plain sight. The main difference here is that steganography does not attract attention to the encrypted messages since they look like completely readable and plain information. It is very common that the message is encrypted with a symmetrical cryptographic algorithm before being injected via a steganography technique into another public message to prevent and secure the message from accidental discovery or unauthorized extraction. [5]

Most programming languages, software frameworks and application libraries do not provide a cryptography model or even a simple class hierarchy for the different types of cryptography primitives. This makes the proper usage and the creation of cryptographic protocols truly problematic. In most cases, there are only simple classes providing a small variety of algorithm standards with limited output formats, configuration options and not well-enough documented features. They tend to have a non-unified interface for processing information, a huge difference in the required parameters for algorithms of the same type and have limited or no support of polymorphism. This creates many disadvantages, for example:

- Slows down development;
- Boosts the risk of anti-pattern usages;
- Prone to insecure algorithm configuration;
- Lacks the support of subtyping and polymorphism;
- Not suitable for all scientific and security use cases;
- The default cipher parameters tend to be totally insecure;
- Not easily injected into other class services or containers;
- Not enough cipher output or signature digestion formats are available;
- No object-oriented class hierarchy exists or only global functions are available;
- Security, performance and thread safety may vary across different language versions.

The main purpose of this article is to provide an object-oriented solution for the implementation of all the main cryptography primitives used for encryption, message digestion, authentication, key exchange and secure end-to-end communications. This paper includes a practical implementation of an object-oriented class hierarchy for digestion, symmetric and asymmetric cryptosystem realizations that support frequently used data transportation and storage output formats for a language that has no embedded cryptography model. This research is also a part of ongoing work in developing an advanced PHP object-oriented software framework for cryptographic services that provide a full cryptography model. [12]

II. The theoretical object-oriented solution

In most programming languages, the provided functions or classes only support basic cryptography primitives or standards and they tend to have an insecure default configuration. The first step of migrating to a fully object-oriented solution is to build sophisticated wrapper classes, create a proper subtyping definition and define unified interfaces for processing information. The next sections provide a detailed analysis of the used approaches.

Building a suitable class hierarchy for hash functions

The realization must carefully split reusable code into an abstract base class for message digestion, different derived abstract classes for each subtype of hash functions and force the desired implementations via unified interfaces or inherited abstract methods. The base abstract representation of hash functions must force all derived objects to implement at least binary data digestion and simple salting features. It must provide a constructor method specification for forcing the same type of instance creation for all derived classes. After that, we must carefully define more abstract subtype classes that extend the base conceptual representation class and add up all specific functionalities per each type.

In the case of hash functions, the most professional approach is to add a representation for simple message authentication codes (MAC), a keyed hash message authentication code (HMAC) type and a key stretching or derivation specification (KDF, HKDF and PBKDF). It is a good idea to further extend the key stretching abstraction in two separate abstract subclasses for key expansion algorithms and for hardware-resistant digestion. Note that not every sub-classification of message digestion is suitable for the secure hashing of authentication passwords or

implements a time-attack safe digest verification feature. This is why it is a good idea to keep this functionality away from the base class and move it to an optional interface specification that can be implemented only for those subtypes that are recommended for password hashing. You can further increase code reuse via creating mix-in or trait reusable implementations for each interface definition, especially if some features can be fully reused for completely different cryptographic types. Note that, it is a good idea to define the digest formats in a separate interface specification for further reuse in other components that use hash values. Figure 1 shows an example of a hash algorithm hierarchy as a Unified Modelling Language (UML) class diagram.

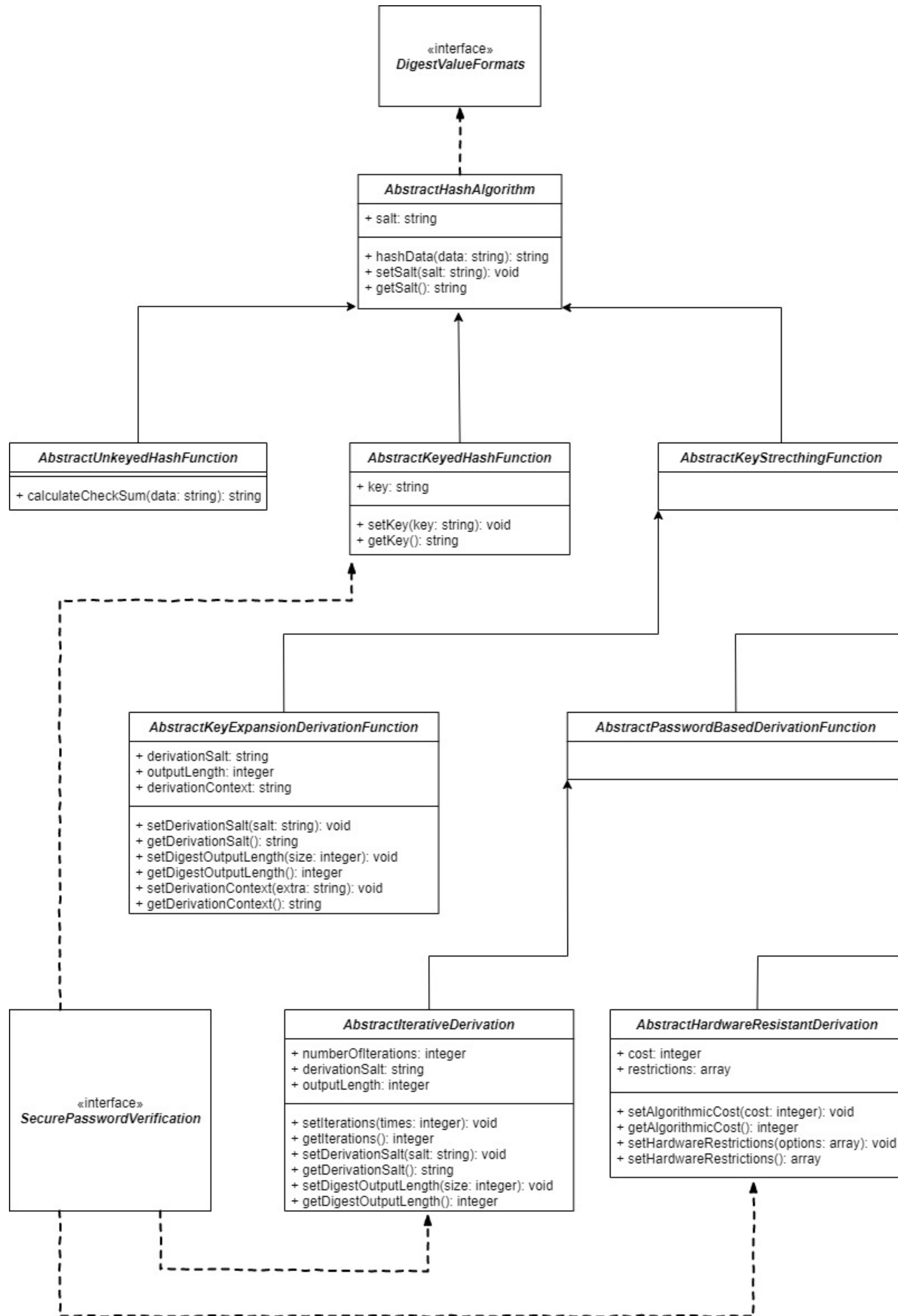

Figure 1. UML Diagram for an Example Hash Algorithm Hierarchy

This hierarchy can become more sophisticated as a realization, for example adding a file digestion interface or creating abstract subtypes for each algorithm family standard you need to implement. In some cases, even a less complex hierarchy may be enough for service injection purposes but subtyping is clearly mandatory.

Building a suitable class hierarchy for symmetric ciphers

The realization of the symmetric cipher hierarchy is less sophisticated than the one for hash functions. The base abstract representation of this type of cryptography must force all derived objects to implement at least binary data encryption, cipher data decryption and secret key manipulations. In addition, it must provide a constructor method specification for forcing the same type of instance creation for all derived classes.

In the case of symmetric algorithms, the most professional approach is to add two abstract derived classes for stream codes and for block ciphers. The stream scheme definition must provide a one pass binary or letter transformation. The block cipher type must include initialization vector manipulations, a variety of final block padding techniques and easy switching between different block cipher modes of operation. Defining interfaces for special features like output cipher data formats is a good idea and can be used for asymmetric encryption algorithms as well. The basic example of a symmetric algorithm hierarchy, as a detailed UML class diagram, can be seen on Figure 2.

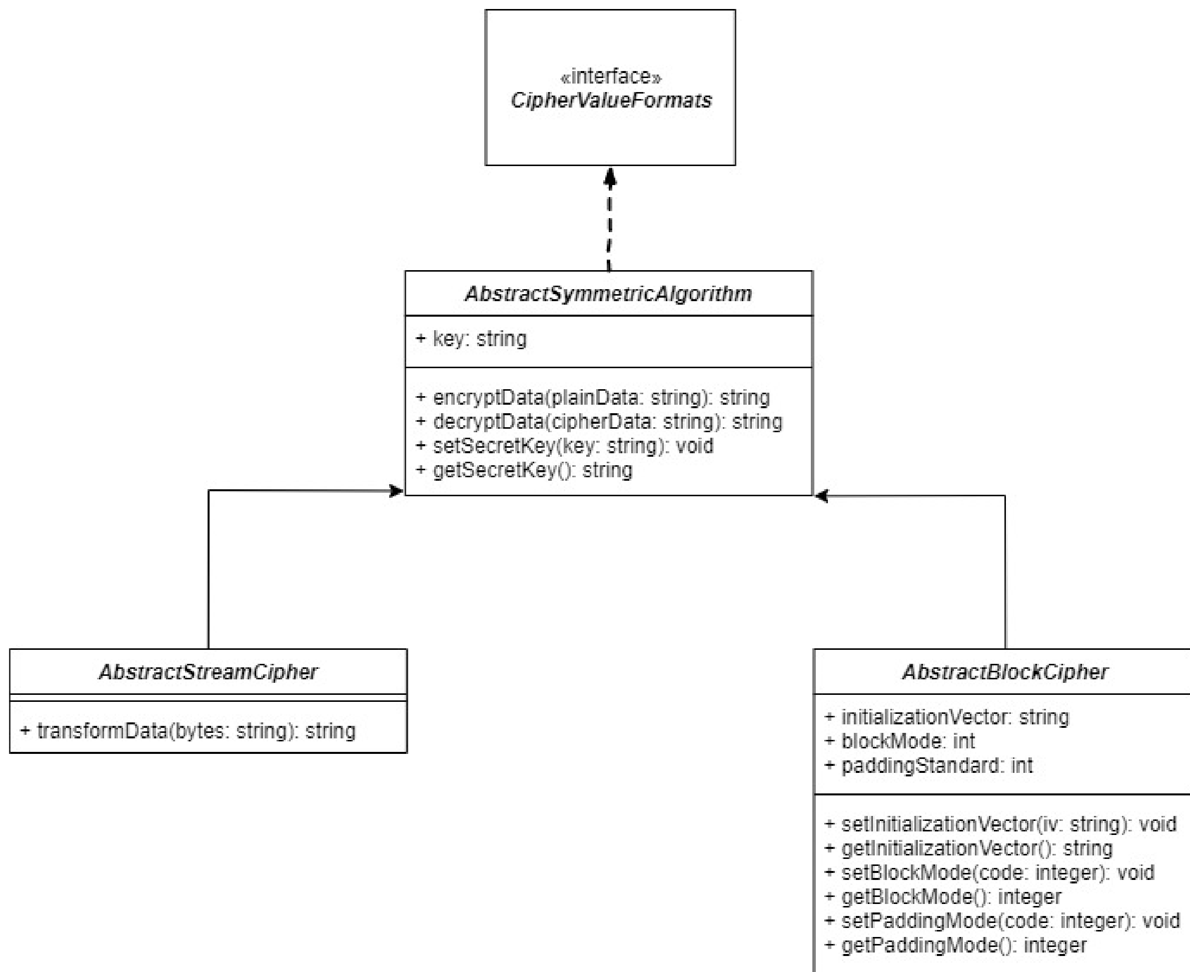

Figure 2. UML Diagram for an Example Symmetric Cipher Hierarchy

This hierarchy provides the necessary minimum of subtyping to achieve the basic type of polymorphism and permits easy injection of symmetric cryptography primitives into container classes. Adding extra interface specifications for secondary features like file encryption or multiple passes of data concealing is recommended.

Building a suitable class hierarchy for asymmetric ciphers

The creation of the asymmetric cipher hierarchy differs from the one for symmetric algorithms because not all subtypes are suitable or do not support any data encryption functionalities. The base abstract representation of this type of cryptography must force all derived objects to implement key pair manipulations and a constructor method specification for forcing the same type of instance creation for all derived classes

The most professional approach is to add two abstract derived classes, one for algorithms that support data encryption and one for data signature generation. Of course, some algorithms can support both features but it is not a good practice to define a single object with multiple processing functionalities, hence it will break the single responsibility coding principle. It is much better to define two objects that extend the two different abstractions and name them including both the standardized name and their semantical usage. In addition, key exchange algorithms can be defined either as a third abstraction representation or as a container class, which uses asymmetric encryption services vie object injection. Also, remember to specify interfaces for special features like output cipher data formats and digital signature formats. The basic example of an asymmetric algorithm class hierarchy, as a comprehensive UML class diagram, is shown on Figure 3.

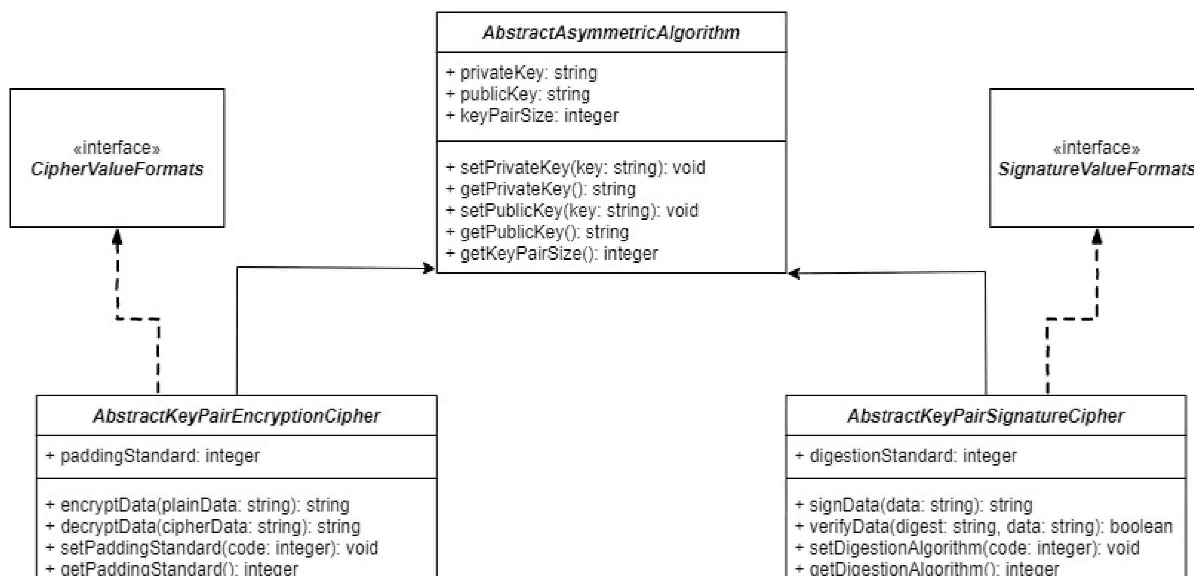

Figure 3. UML Diagram for an Example Asymmetric Cipher Hierarchy

Note that, this hierarchy can become more complex depending on the different standardized algorithms that are included in the specific software realization. It is recommended to keep the key agreement algorithms as a cryptographic protocol container object for the easier injection of the different subtypes of asymmetric ciphers.

Creating optional output formats for digests, cipher data and signature information

Most algorithm implementations in programming languages support only the raw byte output format and the frequently used hexadecimal representation. When providing a software solution for cryptography types, we must define interfaces with the specification of all supported output formats per each primitive. All classes must support raw bytes as an output format but may add representations like Base64, Base64URL, lowercase hexadecimal, upper case hexadecimal or even a compressed binary string.

Usually, the technical realization of converting the output formats is simple enough but needs a vast number of validation cases for each representation. It is a good idea to use the uppercase or the lowercase hexadecimal format for hash functions and digital signatures. The Base64 or Base64URL representation is suitable for both symmetric and asymmetric encryption algorithms, depending on the at-rest and in-transit requirements. Raw bytes or compressed strings are great for file or database storage use cases. Keys and initialization vectors benefit the most of the Base64 format for storage and the Base64URL representation for transportation. Asymmetric key pairs may have an extended complex format with subsections and can be converted to Base64 encoded strings for easier key pair importing and exporting. The output format options are almost unlimited and depend entirely on the system needs or the software developer's taste.

Creating native language realizations for incompatible or missing algorithms

Most programming languages have a minimum of basic cryptography features in their core. Usually, the cryptography software libraries that come pre-bundled with the language contain the custom implementation of more advanced features and all software frameworks take advantage of them. Sadly, over time some of the algorithms that software systems use are removed because of security concerns and introduce crucial technical migration problems. In addition, modern new algorithms and standards take too long to be implemented in such low-level libraries but are desperately needed for the system's security. This leaves the developer with a lot of compatibility issues and security problems to mitigate.

Before the developer can migrate the system to a new language version, software framework, cryptography library or put new innovative cryptography standards into use, the correctly implement native algorithm realizations with full backward compatibility must be created. The implementation must simulate the same behavior between different versions of the programming language and the cryptography library. Before a new algorithm is implemented in a low-level library, most people write native realizations, which are slower by definition but provide the necessary functionality on time. The implementation must fallow the official algorithm specification or the standard's pseudo-code without introducing bugs or unwanted behaviors to the algorithm. The usage of shared memory locks for all used resources and the definition of the main logic in static memory will reduce thread safety and parallel processing issues. It is important to note, that the usage of 3rd party implementations of cryptography algorithms instead of embedded ones, may lead to even more severe bugs, security breaches, compatibility failures and performance issues. When creating such algorithms in native code, always check the official algorithm definition, completely test the whole implementation meticulously, be careful of system integration issues and set up a secure default configuration for the algorithm object instancing.

III. The software solution implementation

With the wide case usage of cryptography primitives, the need for a practical implementation of the discussed theoretical hierarchy solution is clearly immense. The created software solution for all three main cryptography types is introduced as a part of a PHP object-oriented software framework for cryptographic services. The framework also includes a component for the secure generation of secret keys, asymmetric key pairs, tokens, passwords and initialization vectors. By default, the generator object produces data via a cryptographically secure pseudo-random number generator (CSPRNG) primitive object, which is part of the framework along with a quasi-random (QRNG) and a pseudo-random number generator (PRNG). [11] The next sections give a detailed description of the implementation, which completely relies on the previously discussed approaches. The complete source code is available and published online at the framework's GitHub repository. [12]

The class hierarchy implementation for hash functions

The realization of the hash algorithm hierarchy consists of a similar subtyping arrangement as discussed before. A few features that are more complex were added to the objects, for example, a huge variety of salting modes, file hashing and object hashing. The implementation divides subtyping to MAC, HMAC, HKDF, PBKDF2 and hardware-resistant KDF representations. The message digestion algorithms implemented are the MD5, the SHA-1, the SHA-2 family variations (224, 256, 384 and 512 bit), the SHA-3 family variations (224, 256, 384 and 512 bit) and the Whirlpool standards. Each of these is available for MAC, HMAC, HKDF and PBKDF2 standard hash value calculation. The hardware-resistant HKF functions implemented are Bcrypt, Argon2i and Argon2id.

The supported digestion formats are raw bytes, lowercase hexadecimal, uppercase hexadecimal, Base64 and Base64URL. The secure hashing of passwords and time-attack safe digest verification has been defined by an interface only implemented in the suitable for password digestion HMAC, PBKDF2, Bcrypt, Argon2i and Argon2id standards. It is planned to add more message digestion standards like the RIPEMD family of algorithms in the final version of the framework.

The class hierarchy implementation for symmetric ciphers

The implementation of the symmetric algorithm hierarchy consists of an identical subtyping class allocation. Advanced features were added to the objects, for example, file encryption, file decryption, object encryption and object decryption. The implementation divides subtyping to stream and block ciphers representations. The symmetric encryption algorithms implemented are the AES block family variations (128, 192 and 256 bit), the block CAMELLIA family variations (128, 192 and 256 bit) and the RC4 (128 bit) stream standard.

The supported block modes for AES and CAMELLIA are ECB, CBC, CTR, CFB and OFB. The supported padding algorithms are the PKCS7 standard and the unofficial zero-fill concatenation technique. The stream cipher

data encryption and decryption procedures are done via an 8-bit binary internal transformation. The provided cipher data formats are raw bytes, lowercase hexadecimal, uppercase hexadecimal, Base64 and Base64URL. It is planned to add more symmetric encryption standards like the 3DES (Triple DES) algorithm in the final version of the software framework.

The class hierarchy implementation for asymmetric ciphers

The practical fulfillment of the asymmetric algorithm hierarchy consists of a bit modified version of the discussed subtyping class placement and naming. Extra features were added to the encryption object type, for example, chunk processing, file encryption, file decryption, object encryption and object decryption. A few more complex features were added to the digital signature object type, for example, file signature generation, file signature verification, object signature generation and object signature verification.

The implementation divides subtyping to the RSA encryption standard and the DSA signature algorithm. The asymmetric encryption algorithms implemented are from the RSA family (1024, 2048, 3072 and 4096 bit). The digital signature standards implemented are from the DSA family (1024, 2048, 3072 and 4096 bit). It is important to say that the key pair generation component provides a way of generating keys with size between 384 bits and 15360 bits (dividable by 128 bits). This is enabled because of two simple reasons. The first is that the generated key pairs can be decoded from the default framework Base64 format and used in other applications and programming languages in their raw representation. The second reason is that a software developer can easily extend the RSA or DSA abstractions to create a new object and configure the wanted key pair size by just overriding one integer constant to the desired value. No realizations based on a small key pair size like those that use a 384-bit length were added because of security reasons. The non-typically used huge sizes like those that use an 8192-bit length are not included because of the long key pair generation time, which causes the inability of the unit test suite to finish on time on both the local development machine and the remote continuous integration server.

The supported padding standards for the RSA algorithm types are PKCS1 and OAEP. The available internal message digestion functions used when the DSA algorithm generates signatures are the SHA-1 and the SHA-2 family variations (224, 256, 384 and 512 bit) standards. The implemented digital signature formats are raw bytes, lowercase hexadecimal, uppercase hexadecimal, Base64 and Base64URL. In addition, although it is not recommended because of security and performance reasons, you can enable the chunk-processing feature to be able to conceal input data bigger than the key size for all RSA objects, which is otherwise not possible. The key agreement cryptographic protocols are not specified as a subtype because they can be easily defined via container objects that can inject RSA instances, which are by default designed with the capability of importing non-related private and public keys to simulate crossed keys between distant parties. It is planned to add more asymmetric standards like the DH key exchange algorithm in the final version of the framework.

IV. The importance of object-oriented hierarchies for cryptographic protocols

The next sections explain the significance of the implemented cryptography primitives and their object-oriented hierarchies. They show standard use cases for cryptographic and security protocols, as well as explain the dependency injection principle. In addition, there is a complete implementation guide included for each of the given practical examples.

The dependency injection software design principle

In the context of object-oriented programming, the dependency injection principle is a specific form of decoupling of software modules and causes a huge improvement of the overall code reusability. This approach states that the high-level components in an application should not depend on the low-level ones and that both should rather depend on abstractions, instead of strict implementations. This technique manages to provide a simple way to separate the creation of an object from its general usage. When the application architecture supports polymorphism and has a well-defined hierarchy of objects, we can easily create container-based classes for high-level logic that can easily reuse and inject some of our components as services of some kind. This basic decoupling approach is the general definition of the dependency injection technique. [9]

In the case of cryptography primitives, we can design container classes for standardized cryptographic protocols and techniques, which can easily inject and use different algorithms from the same subtype or interface specification. The previously discussed hierarchies allow all abstract subtype representations to be injectable into any class that inherits a container specification abstraction or implements some kind of service setter interface. The next sections describe the usage of this principle to define standard cryptographic protocols and as container objects.

The authenticated encryption and decryption technique

The process of authenticated encryption (AE) provides confidentiality by symmetrically encrypting the data and forces authenticity by creating a message authentication tag over the encrypted data. The tag is a secure message digest code of the cipher data and ensures the data is not accidentally altered or maliciously tampered with during its transmission or storage. [3] [8]

There are different types of authenticated encryption like generating a digest of the plain data instead of the cipher one or hiding the tag at the end of the plain data before encrypting it. Some standards use dedicated modes to realize this feature, like the GCM and the CCM block modes that are defined and usable for the AES family algorithms. A more unified and secure approach is to prefer the CTR block mode for authenticated encryption, in combination with a keyed hash message authentication code (HMAC). It is recommended that the key for the symmetric algorithm and the message digestion function are not the same. The initialization vector must also be very different from the two used keys. Avoiding the usage of the basic MAC algorithms, which do not use or demand a key for hashing, is a good idea although some standard implementations use them by default. The basic example of authenticated encryption and decryption, as a detailed UML flow diagram, is shown on Figure 4.

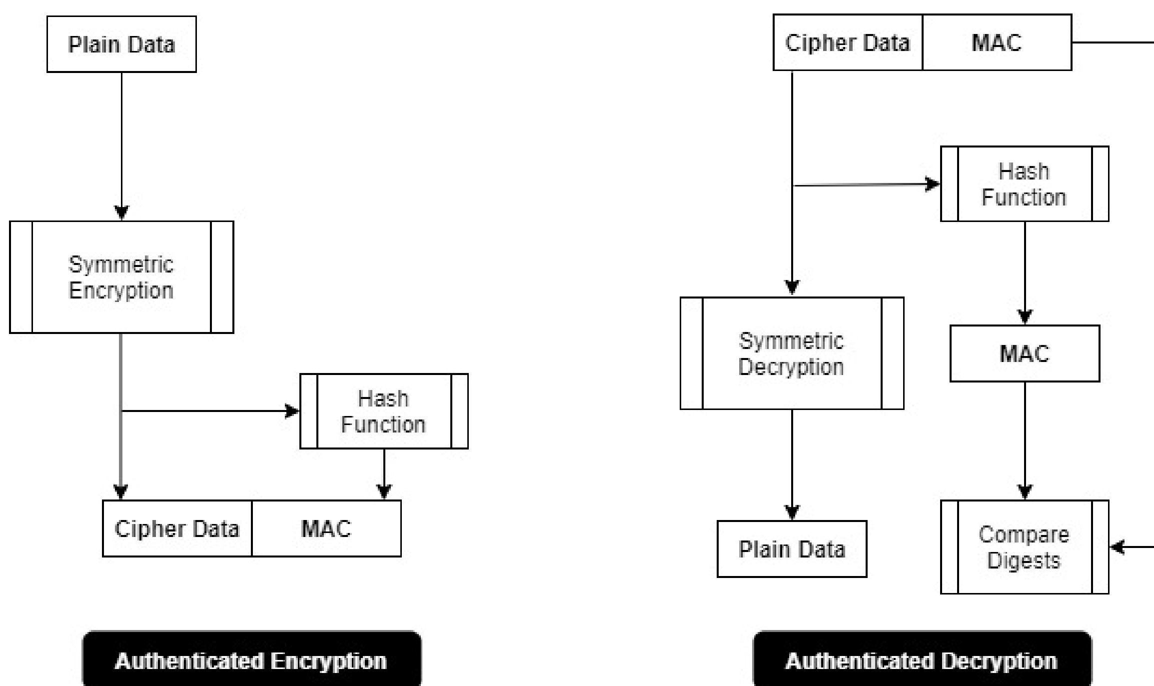

Figure 4. UML Diagram for an Example Authenticated Encryption and Decryption

Creating a simple dependency injection container for the implementation of the authenticated encryption technique that is able to inject both hash functions and symmetric encryption ciphers is clearly easy because of the existence of an object-oriented class hierarchy for each of the types. Using the implemented keyed hash message code and symmetric block cipher abstractions we can easily control the supported types for injection. In addition, the creation of a proper constructor specification with parameters for the two service types and at least a few setter methods for easier further switching is strictly mandatory.

The cryptography secure envelope protocol

The technique of sending cryptography secure message envelopes consists of the simultaneous usage of asymmetric and symmetric encryption algorithms. It is important to note that the encryption and decryption with asymmetric key pairs are computationally expensive and typically, communication messages are not directly encrypted with them. Instead, the asymmetric encryption is used to conceal a generated pseudo-random symmetric "session" key for secure transportation reasons. The raw generated key is required for the actual symmetric encryption of the plain message before dispatching the envelope. The envelope structure consists of the encrypted representations of the message and the session key. In addition, an encrypted value for the initialization vector may

also be included or we can derive one directly from the generated key via a cryptographic key derivation function. The cryptographic envelope protocol has two main processes defined as the sealing and the opening operations. [10]

The sealing procedure starts with the generation of a random session key and initialization vector via a cryptographically secure pseudo-random number generator (CSPRNG). The next step is to encrypt the message for transportation with a symmetric encryption algorithm with the generated session key and session IV. The next step is to encrypt both the session key and initialization vector with an asymmetric public key. Note that the second party must possess the appropriate private key for the envelope cryptographic protocol to work as expected. The final step is to build a transportation friendly string structure contain the encrypted message, session key and session IV. Finally, the envelope has been constructed and successfully sealed for sending to the interested party.

After an envelope has been received, the opening procedure starts with the proper parsing of the chosen transportation format and the setup of the previously agreed algorithm configurations like the block mode and the padding standard options. The first step is to decrypt the session key and session IV via the owned private key. Next, we can use them to do the symmetric decryption of the received cipher message. The final step is to semantically parse and process the plain message.

Some systems use cryptography envelopes for both side communications, which means each side has a public key for encryption and an unrelated to it private key for decryption. Some systems may even include a message authentication code inside the envelope structure to ensure the data's authenticity. Figure 5 shows an example of asymmetric envelope sealing and opening as a detailed UML flow diagram.

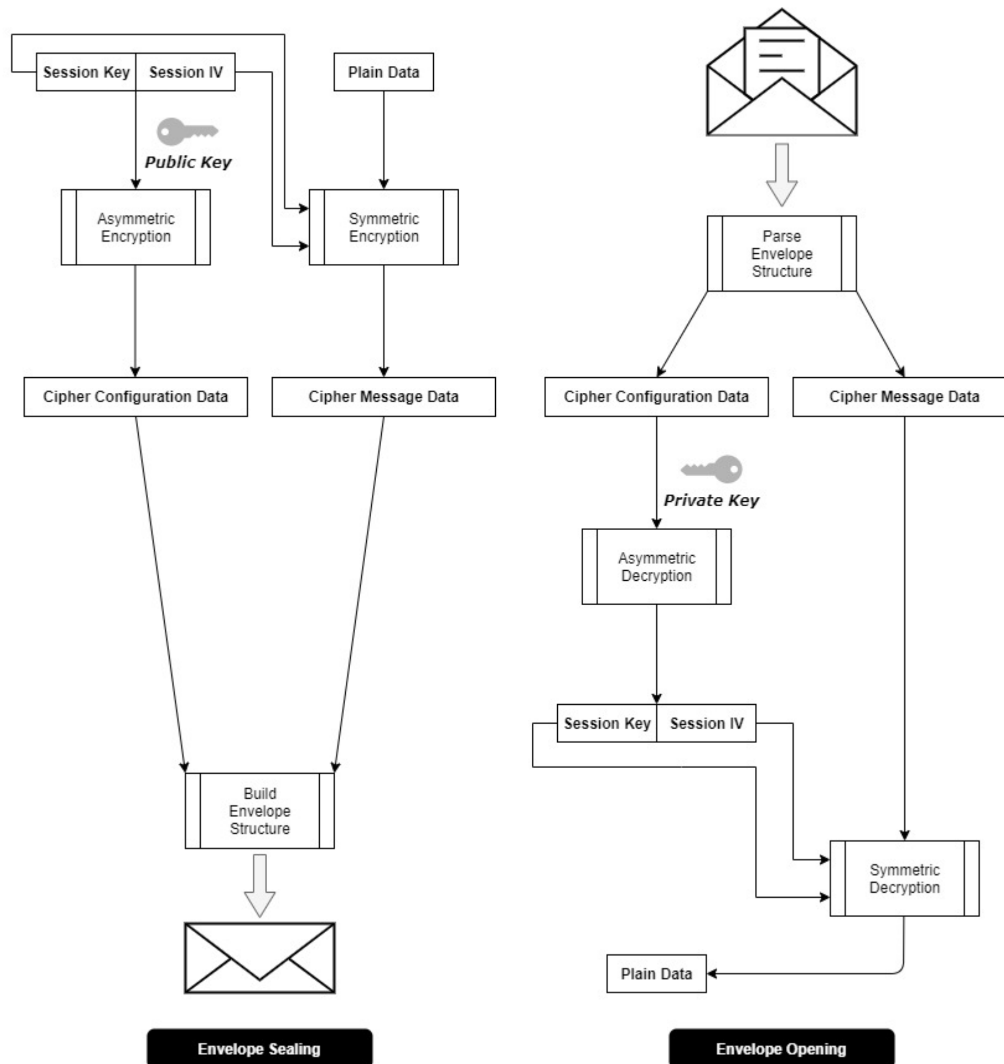

Figure 5. UML Diagram for an Example Envelope Sealing and Opening

The implementation of a basic dependency injection container for the sealing and the opening of secure envelopes that can inject both symmetric and asymmetric cryptography primitives is quite effortless because of the existence of an object-oriented class hierarchy for each encryption method. The service injection control is easily achievable with the implemented asymmetric and symmetric cipher abstractions. It is obligatory to set up a proper constructor specification with parameters for the two service types and at least a few setter methods for subsequent dependency switching.

The password-based authentication method

The definition of an authentication protocol is a type of computer communications scheme designed for the transfer of authentication data between two entities for given identification information. The purpose of such protocols is to verify the identity of a user, a system or a network that requests access to a resource or a service via the supplied request authentication material. It is important to say, that there are many different ways of identification like unique numbers, user names and physical card chips. The point of using an authentication process is to prove that the recognized valid identification information really belongs to the client-side trying to use it. Note that, the authorization process or access control depends totally on the context of the used system and is used only after the authentication process finishes successfully on the remote server-side. [6] [7]

The authorization type depends on the amount of information required to verify the connection's identity. The first subtype asks for "something a user knows", for example, a password or PIN code, and is defined as single-factor authentication (SFA). The second way of authenticating is the two-factor authentication (TFA) process and in addition to the first factor, it adds a requirement for "something a user has", for example, a signed digital certificate for remote network access or a previously verified phone number. The last defined type is the multi-factor authentication (MFA) that requires the previous factors and "something a user is", for example, biometric information like a fingerprint or a retina scan. The goal of MFA is to create a layered defense and make it more difficult for an unauthorized entity to access a system, network or physical device. If even one factor is compromised or broken, there is at least one more barrier to breach before obtaining control.

It is important to note that the above types of authentication cover the case of one client entity requesting to connect to another server entity and can be referred to as one-way authentication. The case when each party needs to be authenticated in front of each other is a mutual authentication or a two-way authentication. In addition, the authentication process is usually triggered once per session on the connection's initialization, but some systems may require a continuous authentication process. One example is the checking of biometrics via sensors or cameras that are watching for behavior changes such as non-typical writing style, keystroke dynamics or even facial expressions.

The most commonly used type is the single-factor password-based authentication. The only requirement to prove the client's identity is to supply a secret passphrase or a token string that only the user knows. The frequently used implementations of this authentication protocol include:

- The sending of a plain passphrase or a calculated secure hash value of it;
- The sending of a decrypted message, which was previously encrypted with a secret key;
- The sending of a decrypted message, which was previously encrypted with a public key.

The first type requires that both communication parties know the secret passphrase or token string that the user knows and has configured on the remote system. The second one is based on the idea that both conversation sides possess the same secret key for a symmetric encryption algorithm. The last method requires that the client-side has the private key locally configured, the remote server-side has the public key correctly set up and both of the keys are from the same asymmetric key pair assigned to the current user. The basic use case interactions of the different password-based authentication types are shown on Figure 6 as a comprehensive UML sequence diagram.

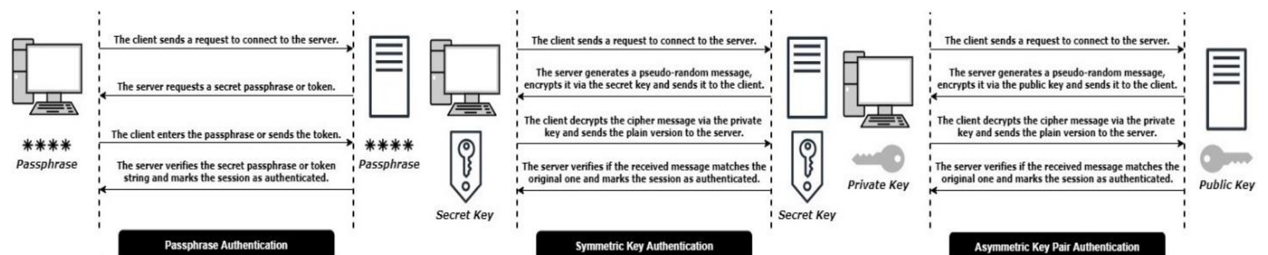

Figure 6. UML Diagram for Password-based Authentication Interactions

The implementation of a dependency injection container for password-based authentication types is easy because of the object-oriented hierarchies for symmetric and asymmetric encryption. The hash function primitives can also be used for tightening the security of the simple passphrase authentication type by using them for the concealing of the raw passphrase and preventing password leaks and identification theft. In addition, with the implemented abstractions we can easily control the supported types for service injection. The realization of a proper constructor specification with parameters for the required type of services and at least a setter method for further switching is indisputably essential. This type of container is also suitable for two-way and continuous authentication.

V. Conclusion

This paper has created an object-oriented solution that provides the three main cryptography types via sophisticated class hierarchy realizations. It accurately identifies the proper use cases for each algorithm category and summarizes the best approaches in combining them for the creation of secure communication protocols and advanced security techniques.

The article also defines all the fundamental cryptography primitives needed for the further development of a secure object-oriented cryptography model in any programming language. In addition, it successfully confirms the importance of cryptographic algorithms, protocols and techniques for the development of any state-of-the-art software or hardware system.

Acknowledgements

University of Plovdiv Paisii Hilendarski, 24 Tzar Asen, 4000 Plovdiv, Bulgaria

Author Contributions

Both authors contributed equally to this work.

References

[1]. A. Menezes, P. van Oorschot, S. Vanstone. Handbook of Applied Cryptography. 1996. ISBN: 9781439821916.
[2]. S. Vaudenay. A Classical Introduction to Cryptography: Applications for Communications Security. 2005. ISBN: 9780387254647.
[3]. J. Katz, Y. Lindell. Introduction to Modern Cryptography: Principles and Protocols. 2007. ISBN: 9781584885511.
[4]. B. Schneire. Applied Cryptography: Protocols, Algorithms, and Source Code in C. 1995. ISBN: 9780471117094.
[5]. J. Fridrich. Steganography in Digital Media: Principles, Algorithms, and Applications. 2009. ISBN: 9780521190190.
[6]. J. Andress , S. Winterfeld. Cyber Warfare: Techniques, Tactics and Tools for Security Practitioners. 2003. ISBN: 9780124166721.
[7]. E. Gilman, D. Barth. Zero Trust Networks: Building Secure Systems in Untrusted Networks. 2017. ISBN: 9781491962190.
[8]. J. Aumasson. Serious Cryptography: A Practical Introduction to Modern Encryption. 2017. ISBN: 9781593278267.
[9]. M. Seemann, S. van Deursen. Dependency Injection Principles, Practices, and Patterns. 2019. ISBN: 9781617294730.
[10]. M. Rashed, S. Ullah, R. Yasmin. Secured Message Data Transactions with a Digital Envelope (DE) - A Higher Level Cryptographic Technique. 2013. DOI: 10.13140/RG.2.1.3372.4963.
[11]. T. Karavasilev. Object-Oriented Pseudo-Random Data Generator Realizations. 2019. DOI: 10.1088/1757-899X/618/1/012032.
[12]. T. Karavasilev (Т. Каравасилев). TonyKaravasilev/CryptoManana. 2020. DOI: 10.5281/zenodo.2604693.

Tony Karavasilev, et al. "Object-Oriented Cryptography Hierarchy Realizations."
IOSR Journal of Computer Engineering (IOSR-JCE), 22.1 (2020), pp. 57-68.