**Comments in response to public comment period of Draft NISTIR 8214A**

Ronald Tse

Wai Kit Wong

Daniel Wyatt

Nickolay Olshevsky

****

RNP and Confium team

Ribose Inc.

Version 1.0

February 10, 2020

***Unrestricted***

Delivered to [nistir-8214A-comments@nist.gov](mailto:nistir-8214A-comments@nist.gov)



Table of contents

[1 Scope 4](#_Toc32553044)

[2 Normative references 4](#_Toc32553045)

[3 Terms and definitions 4](#_Toc32553046)

[4 Practical usage of new cryptographic families 4](#_Toc32553047)

[4.1 Background 4](#_Toc32553048)

[4.2 Key management 5](#_Toc32553049)

[4.3 Threshold cryptography 5](#_Toc32553050)

[4.4 Adoption challenges 6](#_Toc32553051)

[4.4.1 Overview 6](#_Toc32553052)

[4.4.2 Implementation and availability 6](#_Toc32553053)

[4.4.3 Secret storage management 7](#_Toc32553054)

[4.5 Architecture concerns 8](#_Toc32553055)

[5 Decoupling cryptographic primitives in trust stores 9](#_Toc32553056)

[5.1 Challenges 9](#_Toc32553057)

[5.2 Drivers for an extensible architecture 9](#_Toc32553058)

[5.3 Requirements for a trust store 10](#_Toc32553059)

[6 The Confium trust store 11](#_Toc32553060)

[6.1 Purpose 11](#_Toc32553061)

[6.2 Background 11](#_Toc32553062)

[6.3 Structure 11](#_Toc32553063)

[6.3.1 Overview and diagram 11](#_Toc32553064)

[6.3.2 Keystore layer 12](#_Toc32553065)

[6.3.3 Crypto-primitive layer 12](#_Toc32553066)

[6.3.4 Application layer 13](#_Toc32553067)

[6.4 Modular extension of cryptographic schemes 13](#_Toc32553068)

[6.4.1 Purpose 13](#_Toc32553069)

[6.4.2 Reuse of cryptographic components 13](#_Toc32553070)

[6.4.3 Interface isolation for applications 13](#_Toc32553071)

[6.4.4 Identification and organization of cryptographic schemes 13](#_Toc32553072)

[6.4.5 Example and diagram 14](#_Toc32553073)

[15](#_Toc32553074)

[6.5 Security requirements 15](#_Toc32553075)

[6.5.1 Keystore layer 15](#_Toc32553076)

[6.5.2 Security requirements of the crypto-primitive layer 16](#_Toc32553077)

[6.5.3 Security requirements of the public module repository 16](#_Toc32553078)

[6.6 Algorithms in the keystore layer 17](#_Toc32553079)

[6.6.1 Private keystore 17](#_Toc32553080)

[6.6.2 Public keystore 18](#_Toc32553081)

[6.6.3 Access control checking 18](#_Toc32553082)

[6.7 Crypto-primitive layer 19](#_Toc32553083)

[6.7.1 Third-party modules 19](#_Toc32553084)

[6.7.2 Public module repository 20](#_Toc32553085)

[6.8 Threshold cryptography 20](#_Toc32553086)

[7 Confium supports the NIST threshold cryptography project 21](#_Toc32553087)

[8 Confium feedback to NIST 8214A 22](#_Toc32553088)

[8.1 Threshold cryptography benefits to OpenPGP 22](#_Toc32553089)

[8.2 Alignment to NISTIR 8214A Figure 2 cryptographic modes 22](#_Toc32553090)

[8.2.1 Conventional 23](#_Toc32553091)

[8.2.2 Not-shared-IO 24](#_Toc32553092)

[8.2.3 Shared-IO 25](#_Toc32553093)

[8.2.4 Shared-I 26](#_Toc32553094)

[8.2.5 Shared-O 27](#_Toc32553095)

[9 Security analysis 27](#_Toc32553096)

[10 Development approach 28](#_Toc32553097)

[10.1 Community-driven requirements 28](#_Toc32553098)

[10.2 Open feedback and discussion 28](#_Toc32553099)

[10.3 Co-pilot development 28](#_Toc32553100)

[10.4 Iterative delivery 28](#_Toc32553101)

[11 Information about Ribose 29](#_Toc32553102)

[11.1 General 29](#_Toc32553103)

[11.2 Experience with NIST 29](#_Toc32553104)

[11.3 Commitment to open source and open standards 29](#_Toc32553105)

# Scope

This document describes the open-source Confium trust store project and its alignment to the NIST Threshold Cryptography standardization process. A proposal of offering an open-source framework to support the standardization process is provided.

# Normative references

Draft NISTIR 8214A

Towards NIST Standards for Threshold Schemes for Cryptographic Primitives: A Preliminary Roadmap

# Terms and definitions

keystore

component for storage and management of keys

portable data [ISO/NP 8000-100:2020, in development]

data conforms to applicable data specifications

RNP

open-source OpenPGP library developed by Ribose

Enprot

open-source secure document framework developed by Ribose

# Practical usage of new cryptographic families

## Background

Traditional encryption and signature schemes have been in use for decades to protect end-to-end communication. Recent advancements in technology and the diversification of use cases have introduced more complex security problems, many of which rely on encryption and signature schemes with certain security properties. Traditional schemes are no longer enough to cater for all different scenarios.

New cryptographic schemes are developed to fill this gap. For instance, searchable encryption schemes [2][3] allow a (third-party) cloud server to perform (encrypted) keyword search on encrypted data. Distance-preserving encryption [14] supports data analysis (by third party service providers) on encrypted data.

Threshold cryptography [5] [17] in particular, supports new modes of achieving security, as described in the Draft NISTIR 8214A, including cryptographic secure multi-factor authentication [15].

## Key management

Key management [10][8] is a recurrent topic in cryptographic research. Previous works focus on encryption and signature schemes for simple communication between different parties. [12] is a more recent work that develops a distributed cloud keystore. In this scheme, keys are divided into shares and the shares are stored by different cloud servers. As a result, no single cloud can see the actual keys, only the user able to collect all shares can recover the actual keys.

While numerous patents [15][7] exist on keystore-related technologies, they are not designed to be extensible. If a user would like to use a new encryption scheme in his application for his own purposes, it is not feasible without modifying the underlying source codes of the application.

Security problems have become increasingly complex in recent applications. For example, an encrypted database system developed in [9] uses multiple encryption schemes for encrypting data. Each plain data item results in multiple encrypted values, each encrypted by a special encryption scheme that supports certain functionality on encrypted data. Since encryption on data of different attributes requires different keys -- the many types of keys and number of keys both become concerns.

A keystore is an essential component to manage these keys securely. As commonly seen in research, [9] assumes there is a proper keystore to manage the keys securely, without providing details on how such a keystore is to be constructed.

In reality, implementing a secure keystore is hardly straightforward. For instance, [11] analyzes the security of the keystore in Android system, and discovers a vulnerability where an attacker can undetectably modify a stored key. A thorough security analysis of any keystore is crucial in ensuring the security of keystore and of secrets stored within the keystore.

## Threshold cryptography

A simple application that demonstrates the value of threshold cryptography is multi-factor authentication. Suppose there are three items to prove the identity of a user:

1. user password;
2. a one-time password (OTP) from the user’s phone; and
3. the user’s fingerprint.

A user needs to provide any two of the above items to login the system. This can be done using threshold cryptography with t = 2 and n = 3.

The advantages of such scheme are that:

1. even if one item, say the user password, is stolen by an attacker, the attacker cannot login the system; and
2. when a user loses one item, say the user forgets his password, the user can still be authenticated and an authenticated reset of the key(s) can be done.

These properties of threshold cryptography can clearly improve overall security and ease-of-use for the user.

While many threshold cryptography algorithms [4][1][17][5] have been developed, they all differ in the algorithms and/or security models to achieve thresholdization. There is no standardized interface of threshold cryptography today, and NIST is currently attempting to standardize algorithms in this field.

Without a common standardized interface, it is difficult for an application to make use of threshold cryptography:

* If an application selects to implement a particular algorithm of threshold cryptography, it may not fit the user’s needs that require another implementation of threshold cryptography with some other features.
* It is ineffective for the application to implement multiple algorithms for the user to choose.

Only with a standardized interface, we can separate the application logic from cryptographic tools. Applications can be developed according to the definition of the interface, and any implementation that conforms to the standardized interface can be selected by the user and run smoothly with the application.

## Adoption challenges

### Overview

Threshold cryptography introduces challenges that are not currently addressed by traditional cryptographic libraries and implementations. The two major areas include the implementation and adoption of cryptographic schemes in real-world applications, and interoperable facilitation of secrets utilized by those schemes.

### Implementation and availability

An application may use a number of cryptographic schemes in order to achieve different functionality.

For an application to utilize a new cryptographic scheme, short of directly implementing the scheme within the application, generally requires all the following conditions to be met:

1. The cryptographic scheme has been standardized by one or more standardization development bodies, such as NIST, ISO, ITU and IETF. The standardization timeline has a multi-year horizon and is largely driven by national agenda.
2. The (standardized) cryptographic scheme has been adopted by one or more standard cryptographic libraries, such as OpenSSL, LibreSSL, mbtls, BoringSSL. At this stage stakeholders of the scheme will have to contribute and implement this scheme for the major cryptographic libraries, each with different requirements, security implications and timelines. This stage is often a multi-year process.
3. The standard cryptographic library that implements the cryptographic scheme has been adopted by an operating system vendor or distributor, such as Apple, Microsoft, IBM and Oracle. There is a typical lag between step 2 and 3 of at least a year or more.

These steps are mostly sequential – success in step 1 leads to step 2, etc. They lead to an adoption timeline, even if the cryptographic scheme and its use cases are fortunate enough to garner long-term and widespread support, of at least 5 years to over a decade in an optimistic sense.

All of the above factors leading to the success of a cryptographic scheme are not directly linked to its purpose or technical merit, and are often completely out of the control of the creator and researchers of a cryptographic scheme. Herein lies the difficulty in incorporating threshold cryptography in real applications.

For example, [13] developed a threshold signature scheme as an extension of the traditional RSA signature scheme, such that the signature is generated by multiple parties instead of one party. The method of signature verification is identical to traditional RSA, meaning that the verifier only needs to understand traditional RSA without the need to implement the scheme described in [13]. An application that can practically adopt such threshold signature scheme is likely to be a decade out, even the its mechanisms are based on existing, widely available, cryptographic primitives.

### Secret storage management

Keystores are essential in the operation of encryption and signature schemes as they rely on the protection of secrets. Every primitive may define new types of secrets with different key lengths, properties and operations.

Traditional keystores, such as Oracle JKS1, assume that individual cryptographic schemes are independent, and thus each cryptographic scheme is implemented as a separate module without being extensible.

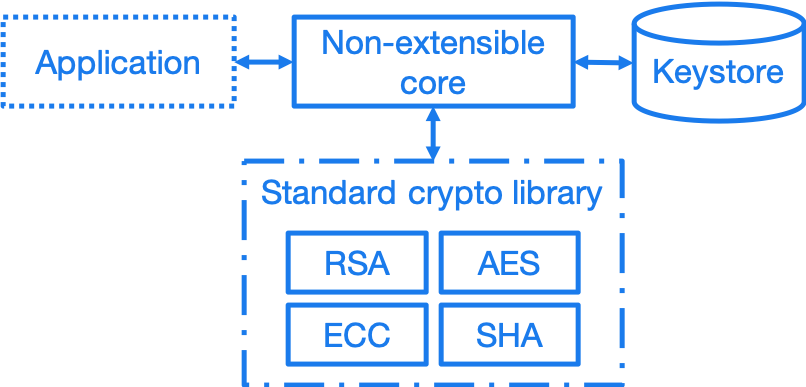
In addition, traditional keystores also rarely provide an interoperable way for others to obtain the generated public keys. The user application, and often the user of the application, has to resort to out-of-band mechanisms to obtain the public keys of others in order to import them into one’s own keystore. This process is opaque to users of the applications, and may introduce more security issues (such as improper sharing of secrets) compared to the enhanced security provided by the adoption of such cryptographic scheme.

In the realm of threshold cryptography, many cryptographic schemes are extensions of some others, and the integration between them and traditional keystores will be clumsy at best.

Moreover, today’s keystores often rely on proprietary secret protection, leading to unwanted lock-in in the storage or keys, reduced resilience in face of application failure, ultimately increasing security risks of the organization. With the advent of multiple threshold cryptography schemes, an open, interoperable keystore will be necessary to manage the various types of secrets.

## Architecture concerns

Existing architecture of standard cryptographic libraries simply do not provide the flexibility necessary for the adoption of threshold cryptography.



*Figure 1. A non-extensible trust store*

Taking RNP as an example application that utilizes a trust store, the conventional architecture of having an application interface directly with the crypto-primitive layer, where the application handles trust management. Under this approach, nearly all responsibility of security lies with the application itself. In fact, this architecture binds the application developer to low-level cryptography, and results in an inflexible, fragile stack.

A screen shot of a smart phone

Description automatically generated

# Decoupling cryptographic primitives in trust stores

## Challenges

Traditional cross-platform trust stores are challenged in many ways, including:

1. Inability to support smart cards and other trust mediums. This is usually controlled solely by the operating system;
2. Inability to retrieve keys from external sources. Traditional trust stores are inward-looking.
3. Types of secrets stored entirely depend on underlying cryptographic libraries. e.g. addition of plaintext padding in an updated version can screw the whole stack.
4. Inability to adopt or extend to future cryptographic families. New cryptographic primitives like threshold cryptography and searchable encryption are out of scope of traditional trust stores.
5. Cryptographers are unable to contribute or influence the key types and mechanisms of the trust store.

Confium aims to address these challenges into opportunities.

## Drivers for an extensible architecture

There are several drivers in creating a trust store with an extensible architecture.

1. User responsibility. It is unfair to the application developer to manage low-level cryptographic mechanisms and keep them up to date all the time, often needing to fix source code due to changes in the underlying cryptographic library;
2. Supporting new cryptographic mechanisms. Searchable encryption (keyword search on encrypted data), attributed-based encryption (crypto-enabled access control), threshold cryptography and post-quantum cryptography are new cryptographic families, of which may not have a generalized interface. Without this interface, it is impossible for applications to adopt them.
3. Provide platform for developers and researchers to implement the state-of-the-art algorithms.

## Requirements for a trust store

A trust store needs to support management of user identities as well as trust towards external identities.

For user identities, the trust store is expected to handle the following tasks:

* storing user’s identity secrets
* manage recipient public keys
* publish public keys (for key servers or web of trust)
* support smartcards/hardware

It is also expected to manage trust towards external identities, including:

* manage validity of external public keys (key servers)
* retrieval of external public keys
* trust level of external public keys
* key signing
* web of trust

The design goals of the Confium trust store include:

1. Provide an extensible architecture to support usage of new cryptographic families. This includes providing “cryptographic provider plugins” to bridge existing cryptographic libraries, and “cryptographic storage plugins” to support different types of keys and parameters.
2. Allow decoupling of dependencies between cryptographic design, implementation, distribution and adoption (at the control of the user).
3. Platform-independent, interoperable key storage for better confidentiality and integrity. This also allows better ease of use, backup and restore capabilities.
4. Utilizes a standardized and accessible key storage format to facilitate interoperability and data portability.
5. Secure storage of secrets with compartmentalized internal security.
6. User applications can control extension activity. For example, using a secret key stored on a locally available, single smartcard device should be possible, even if a cloud key storage module is disabled.
7. Performance and able to serve multiple applications at once.
8. Cross-platform on major operating systems.

# The Confium trust store

## Purpose

Confium is an extensible secure keystore framework for threshold cryptography and beyond. It aims to be an interoperable trust store, that is platform independent and relies on international standards to ensure its contents are portable.

## Background

Confium is an offshoot of RNP, an open-source, open-licensed (BSD) OpenPGP library and toolkit. Both Confium and RNP are developed by Ribose.

RNP is designed to be a high-performance OpenPGP library, fully compliant to RFC 4880 and compatible with GnuPG.

Its benefits have been well demonstrated by Mozilla’s Thunderbird’s decision of embedding RNP to power Thunderbird’s email security capabilities, in order to support its client installations numbering over 20 million worldwide.

## Structure

### Overview and diagram

Confium provides a new keystore framework consisting of 3 layers:

1. the application layer
2. the crypto-primitive layer; and
3. the keystore layer.

The architecture of the secure keystore framework is provided in Figure 2.

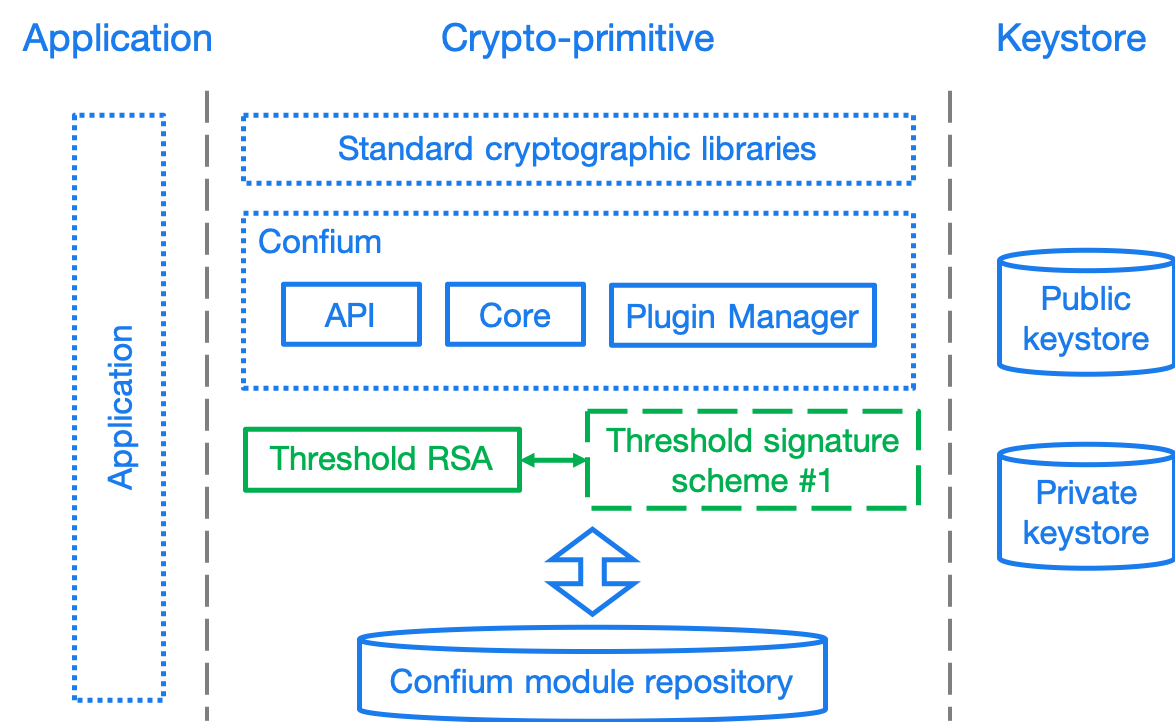


Figure 2. Architecture of the secure keystore framework

### Keystore layer

The keystore layer is responsible for managing keys. There are two separate storage spaces at the keystore layer:

1. The private space is for holding private parameters, such as private keys.
2. The public space is for holding and broadcasting public parameters, such as public keys.

### Crypto-primitive layer

The crypto-primitive layer contains a set of modules, each implementing an individual cryptographic scheme, e.g., AES, RSA. This layer interfaces with the keystore layer to access and store private and public parameters.

The crypto-primitive layer is extensible through its “plugin manager”, where third-party cryptographers (developers) could contribute modules implementing new cryptographic schemes, primitives and keystore mechanisms. These cryptographic plugins are meant to be publicly available and downloadable through a public cryptographic scheme repository.

This layer supports threshold cryptography through the threshold cryptographic module, for which support of multiple threshold cryptographic schemes can be implemented in form of plugins.

### Application layer

The application layer represents any application that make uses of modules provided in the crypto-primitive layer. For example, Mozilla’s Thunderbird represents an application that relies on the crypto-primitive layer.

## Modular extension of cryptographic schemes

### Purpose

The Confium framework provides a mechanism of identifying and facilitating reuse of cryptographic schemes. By allowing potential multiplexing and adoption of multiple cryptographic libraries, authors of cryptographic scheme modules can be free from fear of an updated underlying cryptographic library accidentally breaking compatibility.

### Reuse of cryptographic components

The relationship between cryptographic schemes can be considered as follows. Take threshold RSA [13] and RSA as example. The signing process in threshold RSA uses the threshold cryptography approach. The verification of the signature is the same as usual RSA. The module implementing [13] should utilize the existing RSA implementation for shared calculations including signature verification and multiplicative homomorphic computation to minimize code duplication.

### Interface isolation for applications

Cryptographic schemes are considered to be in the same family where they share characteristic properties, e.g., searchable encryption, threshold cryptography. A common interface can be generalized from the majority of the schemes in the family. By sharing a common programming interface, an application can be developed to bind to such interface without specifying the concrete cryptographic schemes. This promotes isolation between implementation of cryptographic scheme and the application, providing a customizable option to the user, and this decoupling frees up the application’s focus on application logic.

### Identification and organization of cryptographic schemes

Object-oriented programming (OOP) is a paradigm well understood by cryptographic engineers and developers alike. The relationships demonstrated above indicate that the connection between the threshold RSA and vanilla RSA can be considered fitting in the OOP model, allowing effective sharing of code, cutting down redundancy as well as re-implementation risks.

The Confium framework identifies each cryptographic scheme module by a package name and version. Dependencies between cryptographic schemes are explicitly declared.

In addition, Confium provides a Foreign Function Interface (FFI) allowing multiple programming languages to utilize its core. This enables cryptographers to easily prototype cryptographic algorithms using higher-level languages, and when the algorithm moves into production, providing them the ability to use another language with better optimizations.

### Example and diagram

require "confium/ffi"

require "confium/openssl/1.1.1"

class ClownRsa::Key

def generate

process OpenSSL::RSA.generate\_key

end

def sign(data)

...

Figure 3 Sample code implementing a RSA-derived scheme in Ruby

include <confium/ffi.h>

include <confium/openssl/1.1.1.h>

namespace ClownRSA {

namespace Key {

RSA generate() {

const int kBits = 1024;

const int kExp = 3;

return OpenSSL::RSA\_generate\_key(kBits, kExp, 0, 0);

Figure 4 Sample code implementing a RSA-derived scheme in C++

## 

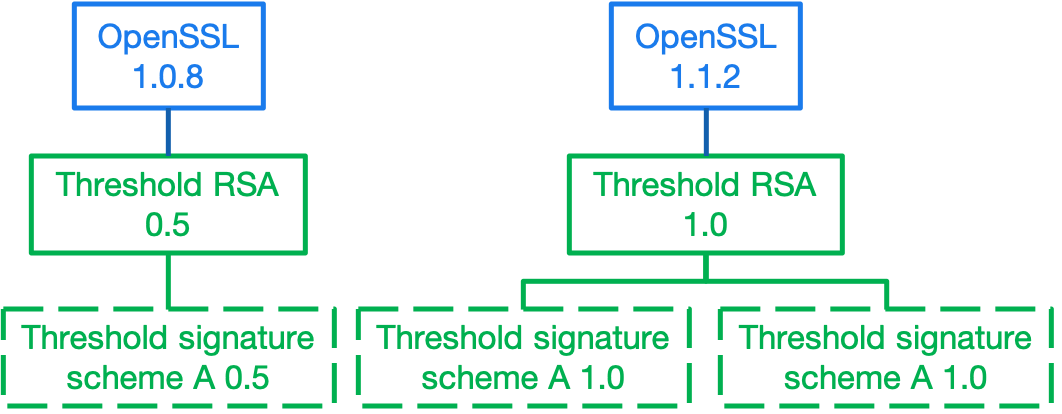


Figure 5. Example of cryptographic scheme versions and versioned dependencies

## Security requirements

### Keystore layer

#### General

The keystore layer provides helper procedures to store and retrieve parameters securely. Generally speaking, there are two aspects of security that the framework needs to cater for: confidentiality and integrity.

#### Confidentiality

Each module in the crypto-primitive layer has a private space in the keystore for keeping the private parameters. The extending module extending a parent module may have access to certain data or procedures of the parent module.

The framework provides three levels of visibility:

1. private: only the module itself can access;
2. protected: the module and its extending modules can access; and
3. public: there is no restriction on the access.

#### Integrity

The framework needs to ensure private data stored inside the keystore cannot be modified by outside parties without detection.

This can be achieved using signature schemes; but module dependencies may render signature verification is not straightforward.

Take for example, an extending module that only implements key generation, and inherits all other procedures from the RSA module. Keys generated by the extending module are signed by the extending module. The signature verification procedure in RSA using the public key signed by the extending module can be executed because the process does not leak any information.

However, consider another scenario. Suppose a malicious module extends the AES module. We need to perform encryption within the AES module using the private key generated and signed by the malicious extending module. This procedure should not be done as it is risky to use an insecure key to perform encryption.

Our initial framework provides five levels of trust that can be declared by the module on each parameter of each procedure:

1. self: only parameters signed by the module itself can be used;
2. ancestor: only parameters that are signed by the module or its ancestors can be used;
3. descendant: only parameters that are signed by the module or its descendants can be used;
4. relative: parameters that are signed by the module, its ancestors or descendants can be used; and
5. public: no verification on the parameter is required. If one of the parameters of the procedure does not match the declared level, the procedure is not executed.

### Security requirements of the crypto-primitive layer

It is the module developer’s responsibility to make sure an implemented module is secure, e.g., by configuring the visibility of parameters properly according to the initial model described above.

We assume the implementation of any module can be open source. Confidentiality of implementation is not a concern, but integrity of the implementation is of major concern -- malicious modifications of the module’s source code can lead to leakage of private parameters. Cryptographic hashing [22] can be used for integrity verification prior to execution.

### Security requirements of the public module repository

The public module repository in our framework is responsible for providing a list of available modules and their authenticity information. The repository is decentralized in nature in our initial model such that it is not dependent on a single party. Authentication of the history of changes to the repository, i.e., adding, updating or removing modules, can be done using the blockchain technique [14].

## Algorithms in the keystore layer

The keystore is compartmentalized for every separate cryptographic scheme, associated to the combination of module and application identifiers.

### Private keystore

A module in a particular application can freely put and get keys in its private space.

In order to support micro-management of access to individual keys, each individual private key can be associated to the combination of module identifier and a key identifier provided by the application. Only when the module identifier and key identifier matches, the private key can be retrieved.

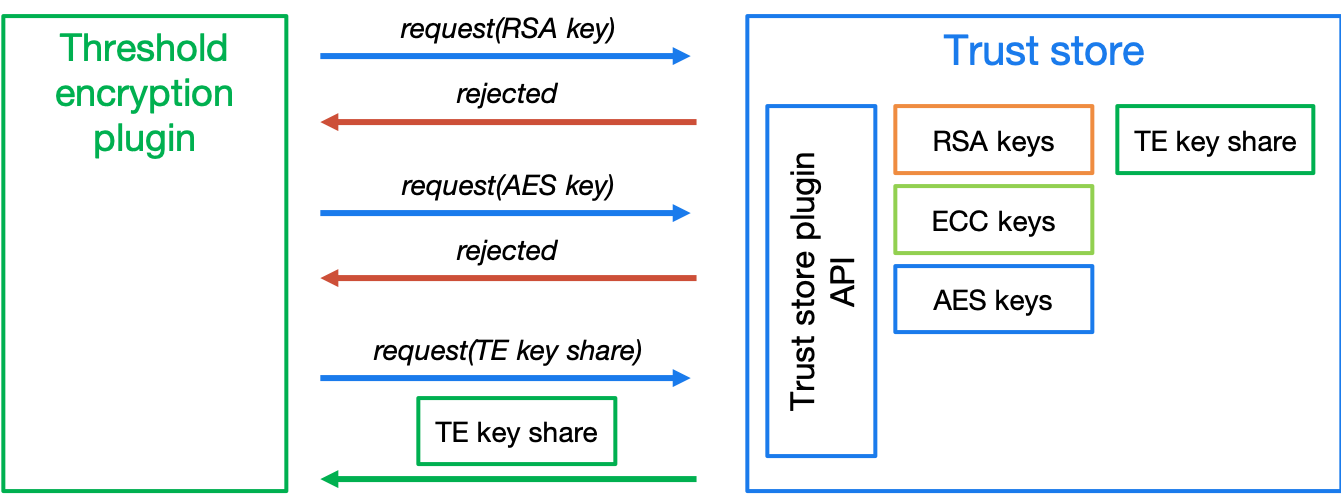


Figure 6. Plugins can only access their own secrets

The keystore has its own master key to encrypt and sign the contents in the keystore. This is to ensure an attacker cannot bypass the keystore to obtain or modify the plain data in it. This algorithm is to generate the master key of the keystore and depends on which encryption and signature schemes are used.

The trust store keeps plugin keys contained and secure via the plugin API:

1. Plugin and trust store generates a secret key *SK* known to both
2. Trust store returns challenge on request of key
   1. *challenge* = *keyRequest(keyId, pluginId)*
   2. Each key (or parameter) is associated with the *keyId* and *pluginId*
3. *key = keyResponse(respond)*

A picture containing text

Description automatically generated

Figure 7. Trust store keeps plugin keys contained and secure via the plugin API

### Public keystore

In public key cryptographic schemes, the public key is supposed to be known by other parties. This is an important part of the keystore as it has been a challenging task to ship one’s public key to another via the Internet. A typical man-in-the-middle attack is practical in many scenarios to let one obtain a forged public key and so the rest of the cryptographic scheme fails.

To address this challenge, there is a public keystore in the keystore layer, that facilitates distribution of public keys. An identity-based signature scheme [6] is used, where the public key in identity-based schemes is the user’s unique information, such as the email address. To upload a new public key, the identity and its signature are also provided to the public space. A key-value store database can be used to store the parameters and provide efficient search in large number of parameters.

### Access control checking

Attribute-based encryption [16] can be incorporated to support complex access policy based on the attributes set on the data items and the users. Our framework also limits on whether a specific procedure can be executed with five levels of trust settings. Note that the procedure can also be executed outside the framework as the module in cryptographic layer is usually open source.

We use access control on private data in the keystore to enforce the execution policy. Private parameters can be obtained only when the all the trust settings of all parameters are satisfied. The procedure cannot be run if some parameters are not obtained. Thus, the problem of enforcing execution policy is transformed to the problem of access control on private data, which can be handled by access control techniques.

## Crypto-primitive layer

### Third-party modules

Modules in the crypto-primitive layer are developed by different developers. The layer here needs to provide a general enough model specification for developers to follow. The model allows the developer to declare a new module or interface that conforms to our framework.

An ideal model is that the developers only need to focus on implementation of the cryptographic schemes and the keystore framework will take care the rest, e.g., secure key storage, authentication procedures. However, the framework cannot understand all the details of the modules automatically, e.g., the access control policy required by the module on each private parameter. Some settings are delegated to the module developer. For example, in our initial model, the developer needs to declare the access policy of each private parameter from our three available policies: private, protected, and public. The model here needs to balance between two factors: control and focus. If there are more settings, the developers can gain more control but on the other hand need to put more efforts on the settings.

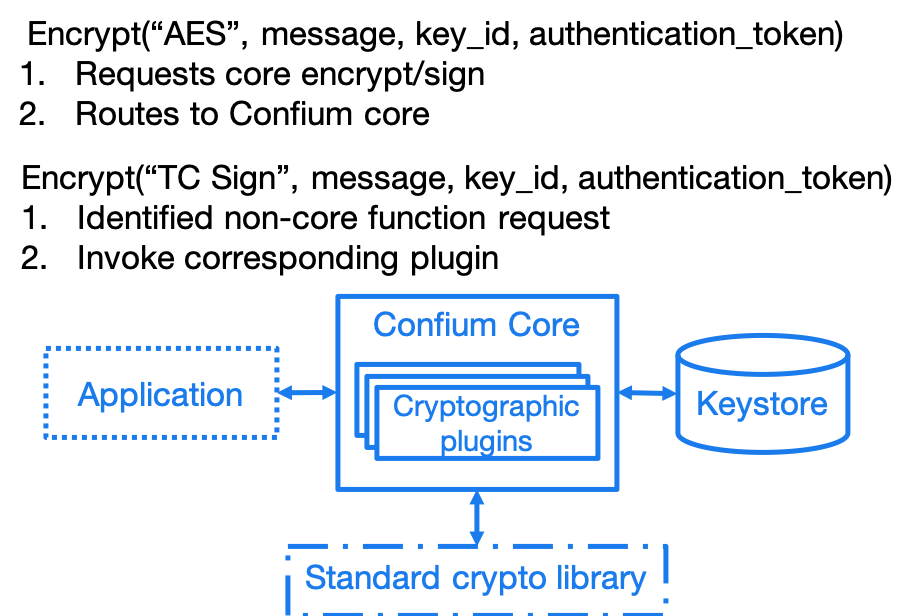


Figure 8. Differentiated treatment of core and non-core schemes

Developers may be distracted from the core implementation of the module. It is not developer friendly if there are too many available settings. There are in general three ways to handle the design on the settings

1. the setting is compulsory;
2. the setting is available but optional. A default option is used if it is not specified by the developer; or
3. there is no setting, and all are forced to use the default setting.

In order to achieve a good balance between control and focus, our team will communicate with people from the industry, e.g., the developers in Thunderbird, to refine our model. There are many design issues in this model, e.g., the format of module identifier, the way to declare inheritance. These issues are trivial but are time-consuming to get a common consent from people in the industry.

### Public module repository

The public module repository is the counterpart to Confium just like how CTAN and CPAN are module repositories for LaTeX and Perl.

Installation of modules must be a direct choice of the user.

Consider the example in an email client. When a user receives an email with a signature signed by a module that is not yet installed on the user’s computer, the user needs to find and install this module in order to verify the signature.

The typical user may not know where and how to find and install such module. The public module repository can automate this search and install process.

When the application sees that it requires a particular module, it can connect to the module repository and download and install the module automatically after the user permits the action.

The repository must enforce careful authentication to prevent malicious parties from modifying existing modules. This can be done by:

* integrating signature mechanism into the repository management mechanism
* certification of modules with code-signing certificates and reviews.

## Threshold cryptography

Confium aims to support new cryptographic families and threshold cryptography is one of them. The threshold cryptography module is implemented in Confium’s crypto-primitive layer, where threshold algorithms could:

1. depend on existing cryptographic algorithms for calculations, such as threshold RSA to RSA
2. have access its own private keystore, and the public keystore of the Confium keystore layer
3. have access to hardware modules exposed by Confium
4. access network interfaces if the scheme is an interactive one.

A screenshot of a cell phone

Description automatically generated

Figure 9. Threshold cryptography module in Confium

# Confium supports the NIST threshold cryptography project

As developers of Confium, we strongly commend and fully support NIST’s effort in the standardization of threshold cryptography.

We believe that the goals of Confium fully support and align with the current standardization efforts, especially in the areas of:

* Providing a common platform for cryptographers to develop prototype to production algorithms and schemes
* Allows cryptographic testing in a sandbox to real-world deployment
* Makes assessment easier by providing a level-playing field
* Potentially common primitives (e.g. networking) allows better focus on what’s important

**We would like to contribute effort in providing Confium to NIST as an open-source implementation test-bed for threshold cryptographic schemes, and are willing to work with NIST in ensuring that the test-bed meets the requirements set by NIST.**

**As a gesture of commitment, Confium will implement a proof of concept to demonstrate the capabilities of the test-bed. It will be a 2-out-of-3 threshold RSA signature scheme, where the secret key is shared across 3 parties, and any pair of them is able to sign or decrypt, but without the secret key ever being recombined.**

# Confium feedback to NIST 8214A

## Threshold cryptography benefits to OpenPGP

The following clauses refer to Appendix A of NIST 8214A (Draft).

1. Threshold cryptography could help store OpenPGP secrets in multiple shares, allowing the private keys to be recoverable. (A.2 Protection of secrets at rest)
2. An OpenPGP identity key can be stored in multiple shares such that only when multiple factors are provided the key could be used (A.3 Confidential communication)
3. An OpenPGP identity key can be distributed across secure environments (A.6 Distribution of trust across secure environments)

As an example of 2), a user may want to keep the private key in 3 shares (iCloud keychain, computer, Yubikey), where 2 must be present to utilize it. This would allow the user to recover the key even if the computer is lost, but the iCloud keychain and Yubikey are still present.

The user could also change the scheme to have 4 portions where 2 must be present, with the ability to store an extra share in another USB key which is stored in a safe to enable recovery in case 2 of the 3 existing factors are lost.

## Alignment to NISTIR 8214A Figure 2 cryptographic modes

The following sections demonstrate how the Confium architecture aligns to the threshold cryptographic modes listed in NIST 8214A Figure 2.

In the following diagrams:

1. The dotted box represents the scope of the Confium project. This includes an interface (API) with the user application. The application can request authentication to utilize the trust store’s features. (does it have access rights to use this key in keystore to encrypt?)
2. The Core cryptographic engine for handling standard encryption / signature using the standard cryptographic libraries.
3. A plugin interface and a manager for managing and interacting with user-installed plugins (e.g., registration of new plugin)

### Conventional

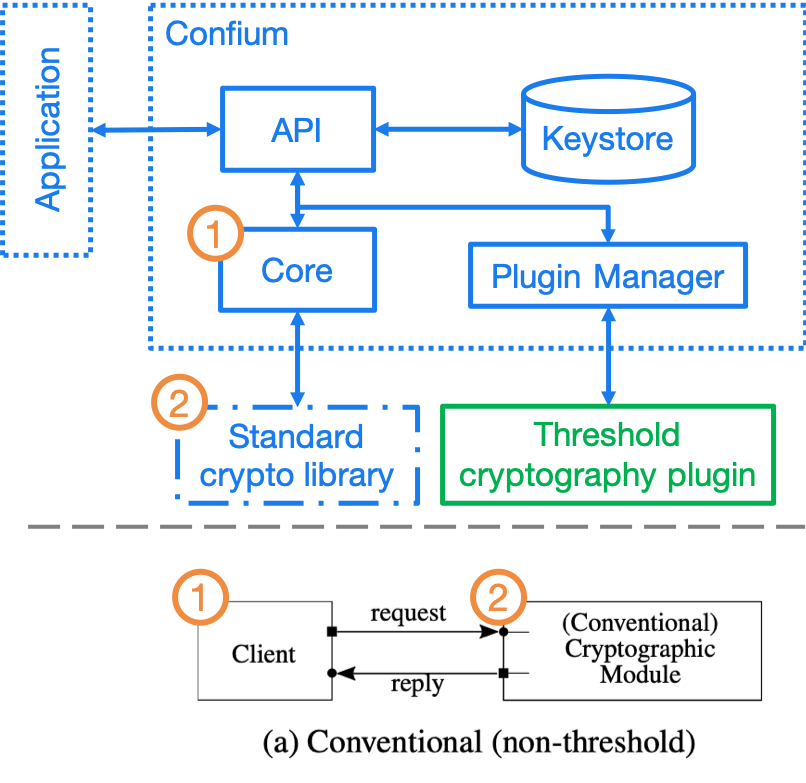


Figure 10. Mode (a)

### Not-shared-IO

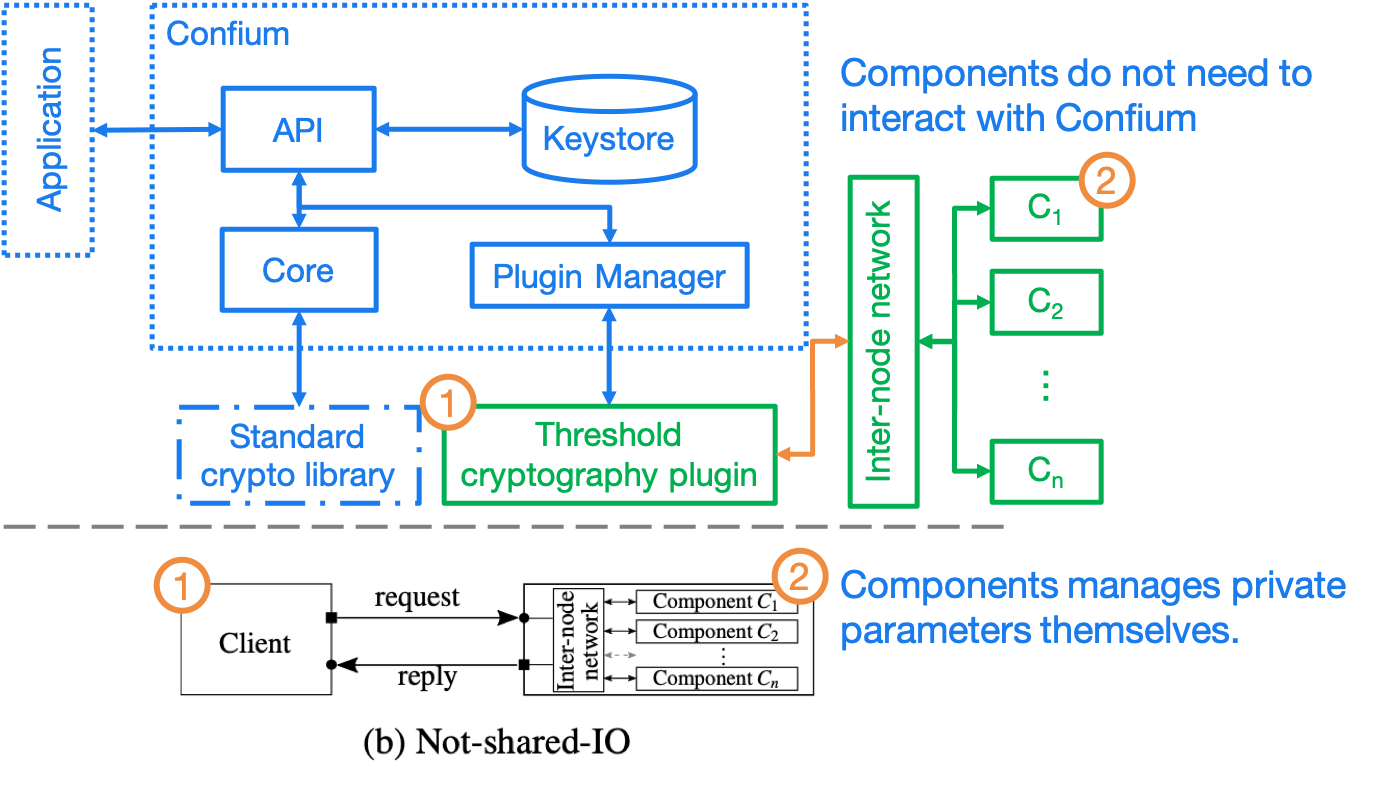


Figure 11. Mode (b)(i) Component not storing keys in Confium

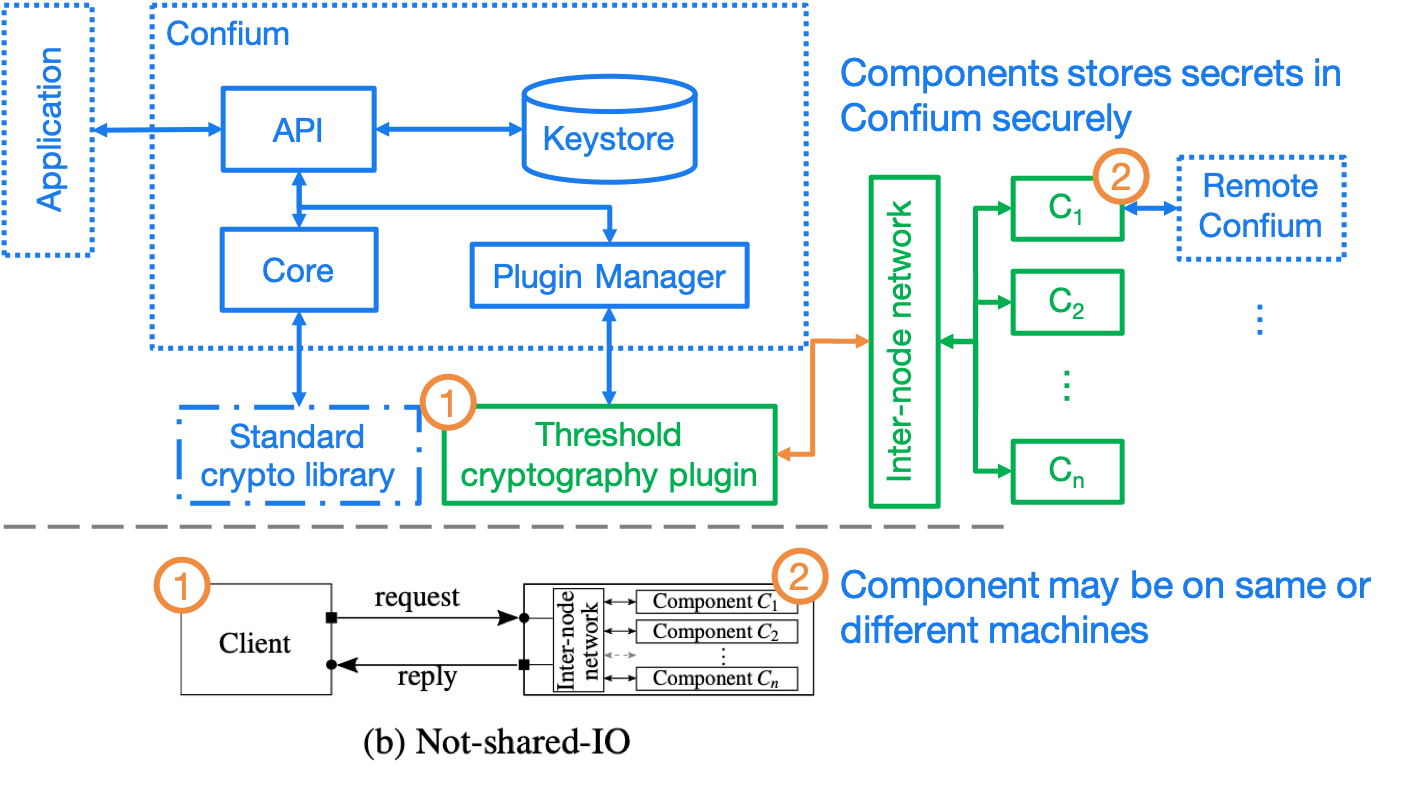


Figure 13. Mode (b)(i) Component storing keys in Confium

### Shared-IO

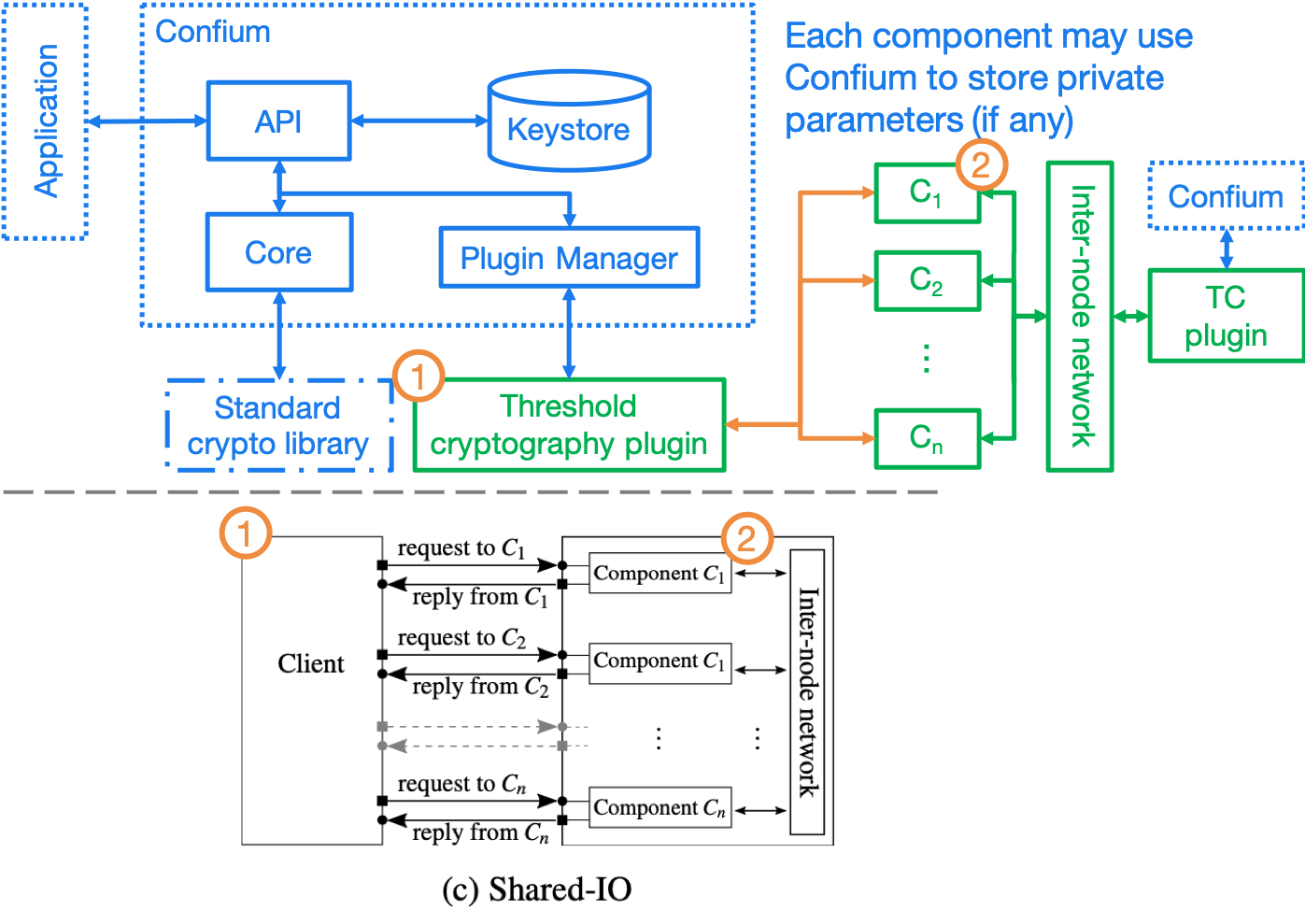


Figure 12. Mode (d)

### Shared-I

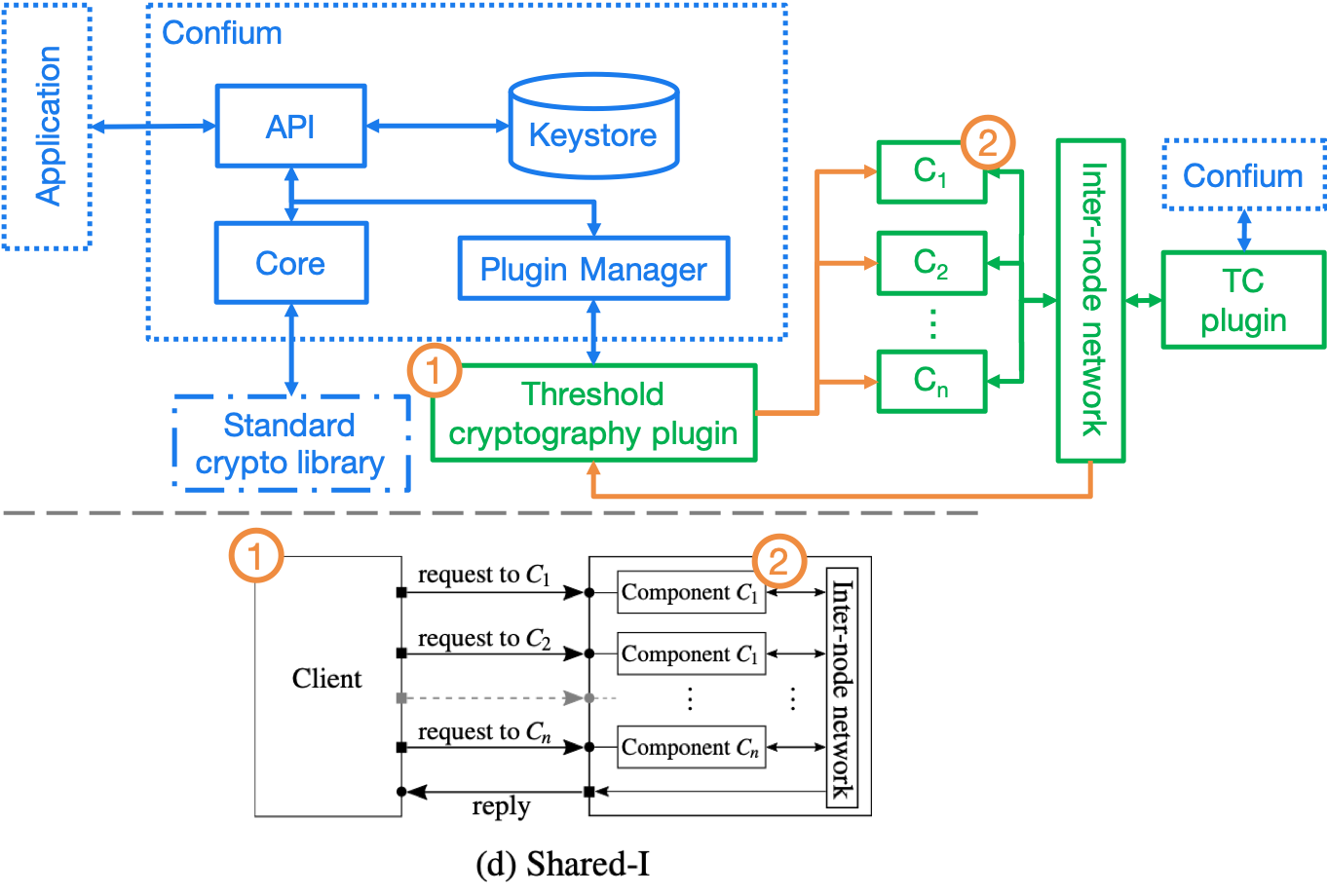


Figure 14. Mode (d)

### Shared-O

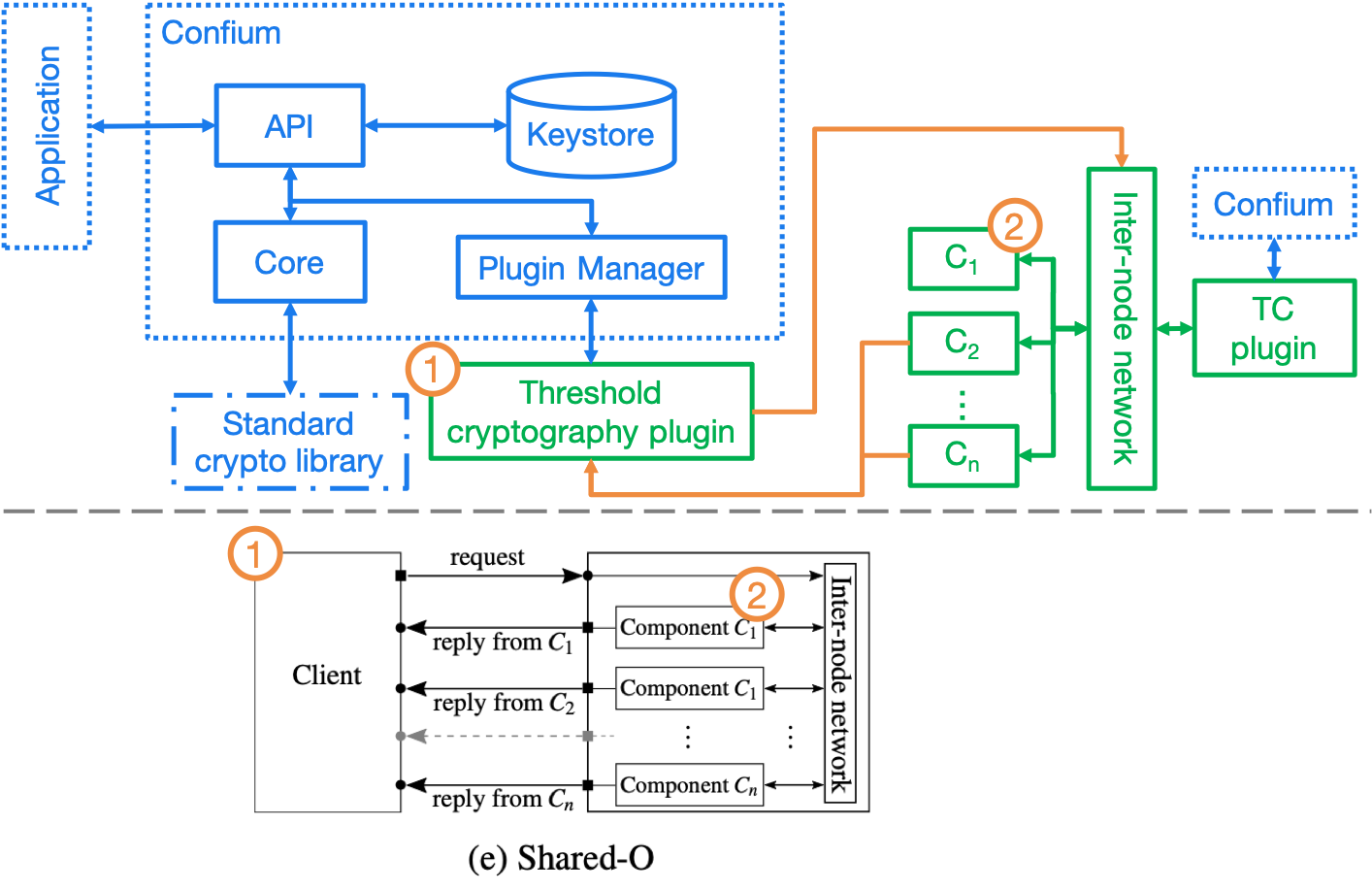


Figure 15. Mode (e)

# Security analysis

We plan to perform security analysis of the Confium framework in two ways:

* theoretical analysis; and
* empirical studies.

Theoretical analysis is usually a security proof with regards to an attack model. The attack models will be formulated, and with analysis performed after that.

In addition, security analysis on the impacts of side-channel attacks [21] will be done. Side-channel attacks refer to the attacks based on implementation of the algorithms, e.g., the time and power consumed to perform a cryptographic action can be used to derive certain information about the private keys in some scenarios. There are many side-channel attacks in practice. We will evaluate the threats of these attacks theoretically and empirically.

A technical report about the results of our empirical study will be delivered.

# Development approach

Ribose is strongly committed to open source development, and the development approach of Confium adheres to the following principles:

1. Community-driven requirements
2. Open feedback and discussion
3. Co-pilot development
4. Iterative delivery

## Community-driven requirements

The intended user community of Confium will drive the implementation requirements of Confium. This community includes application developers that depend on Confium, such as Mozilla Thunderbird, cryptographers of threshold cryptography schemes, as well as other stakeholders.

Requirements and feedback for implementation will be solicited and clarifications sought from this community, ensuring that the resulting deliverable meets stakeholder expectations and their goals.

## Open feedback and discussion

Discussion and feedback for the Confium framework will be performed in public to ensure requirements and any changes to reach as many pairs of eyes as possible to ensure quality and appropriateness.

## Co-pilot development

Confium is developed from scratch as open source software but with development mainly the responsibility of Ribose engineers. The implementation will adopt a co-pilot model allowing a smaller group of key stakeholders to prioritize necessary functionality. The co-pilots here are intended to be Mozilla and NIST.

## Iterative delivery

Confium’s iterative development approach produces continuously functional deliverables that are improved iteratively. This allows early adopters to engage early and interact with new functionality to ensure they meet the expectations of cryptographers and application developers who use Confium.

# Information about Ribose

## General

Ribose Group Inc. (BVI, with its wholly owned subsidiary, Ribose Inc., based in Delaware) is a pioneer of interoperable and cybersecurity technologies across user-centric systems and applications. It has been named a Deloitte Technology FAST 20 Rising Star and Red Herring Global 100 company, and has received CSA’s Enterprise Award for Security Innovation of the Year.

Ribose is the world’s first organization to be certified to the NIST Cybersecurity Framework by BSI, with its experience presented at the inaugural NIST Risk Management Conference in 2018.

Ribose is certified by BSI to ISO/IEC 27001, ISO/IEC 27017, ISO/IEC 27018 and ISO/IEC 27701, ISO/IEC 22301, ISO/IEC 20000-1, ISO 9001, CSA STAR Certification, amongst other schemes.

## Experience with NIST

Ribose is the world’s first organization to be certified to the NIST Cybersecurity Framework by BSI, with its experience presented at the inaugural NIST Risk Management Conference in 2018.

Ribose’s Metanorma framework is currently being piloted by NIST’s Computer Security Division for the publication of the Special Publications 800 series and Cybersecurity Whitepaper documents.

Ribose has contributed to NIST’s “Publication Identifier Syntax” initiative and currently assists NIST’s DLMF LaTeXML project in maintaining their official Snap and Homebrew packages.

## Commitment to open source and open standards

Ribose operates the official Geodetic Registry for ISO (ISO 19127) that powers transformations between international geodetic models used by global navigation satellite systems.

Metanorma is Ribose’s semantic standardization document system. Parts of the framework is being standardized at ISO as ISO 36100, ISO 36200 and ISO 36300. Metanorma is used by a number of international standardization bodies in publishing their standards, including ITU-T, ITU-D, OGC and CalConnect, accepted by IETF, ISO and IEC, and is currently being piloted by NIST’s Computer Security Division and the United Nations Innovation Lab.

Relaton is Ribose’s framework for interoperable citations, providing bibliographic information from standard development bodies including NIST, ISO, IEC, ITU, IETF amongst others. Relaton also powers CalConnect’s Standards Repository forms an integral part of Metanorma.

Coulomb is Ribose’s desktop application framework for editing structured data, currently in used by ITU to publish its Operational Bulletin. Glossarist and Geolexica, Ribose’s terminology management client and server frameworks, are used by ISO TC 211, OSGeo to manage official terminology.

# Bibliography

[1] D. Boneh, X. Boyen, and S. Halevi. Chosen ciphertext secure public key threshold encryption without random oracles. In CT-RSA, 2006.

[2] R. Chen, Y. Mu, G. Yang, F. Guo, X. Huang, X. Wang, and Y. Wang. Server-aided public key encryption with keyword search. IEEE Trans. Information Forensics and Security, 11(12), 2016.

[3] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Im- proved definitions and efficient constructions. Journal of Computer Security, 19(5), 2011.

[4] C. Delerable ́e and D. Pointcheval. Dynamic threshold public-key encryption. In CRYPTO, 2008.

[5] C. Hazay, G. L. Mikkelsen, T. Rabin, T. Toft, and A. A. Nicolosi. Efficient RSA key generation and threshold paillier in the two-party setting. J. Cryptology, 32(2), 2019.

[6] X. Hu, H. Lu, H. Xu, J. Wang, and Y. Yang. An efficient identity-based proxy signature scheme in the standard model with tight reduction. In CISIS-ICEUTE, 2015.

[7] R. Keshava, S. Katti, S. Vepa, and H. Sastry. Key store service, Feb. 8 2018. US Patent App.

15/608,708.

[8] D. Mazieres, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In ACM SOSP, 1999.

[9] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: processing queries on an encrypted database. Commun. ACM, 2012.

[10] S. Rafaeli and D. Hutchison. A survey of key management for secure group communication. ACM Computing Survey, 35(3), 2003.

[11] M. Sabt and J. Traore ́. Breaking into the keystore: A practical forgery attack against android keystore. In ESORICS, 2016.

[12] T. J. E. Schwarz and D. D. E. Long. Clasas: A key-store for the cloud. In IEEE/ACM MASCOTS, 2010.

[13] V. Shoup. Practical threshold signatures. In EUROCRYPT, 2000.

[14] C. Tex, M. Scha ̈ler, and K. Bo ̈hm. Distance-based data mining over encrypted data. In IEEE ICDE, 2018.

[15] M. Webster and S. Pohja. Method for server assisted keystore protection, Sept. 19 2017. US Patent 9,768,960.

[16] G. Xu, H. Li, Y. Dai, K. Yang, and X. Lin. Enabling efficient and geometric range query with access control over encrypted spatial data. IEEE Transactions on Information Forensics and Security, 14(4), 2019.

[17] X. Yan, Y. Lu, L. Liu, S. Wan, W. Ding, and H. Liu. Chinese remainder theorem-based secret image sharing for (k, n) threshold. In ICCCS, 2017.