OpenCCE Documentation

1 General Information

OpenCCE is an Eclipse plugin that provides tool support for Java Crypto APIs. The target audience is regular Java application developers, especially those with only little or no experience in cryptography.

The tool supports developers in two ways. First, a user may select a cryptographic task they intend to implement and OpenCCE generates a secure code snippet into the user's Java project in Eclipse. Second, in addition, OpenCCE statically analyzes any code that uses crypto APIs to ensure a secure usage.

This documentation is structured as follows. Section 2 illustrates how an end-user may interact with OpenCCE. Section 3 describes how crypto experts may contribute either new cryptographic tasks or primitives to OpenCCE. Lastly, Section 4 provides a tutorial for bridging C/C++ and Java code for contributors who prefer to implement their primitives in these languages.

2 How the End-user Uses OpenCCE

Application developers are the end-users of OpenCCE. They are supported by a code generation and a static analysis component. Code generation needs to be triggered explicitly, while static analyses run automatically in the background.

The code generation component provides wrappers for existing crypto APIs. These wrappers are task-based, that is, a wrapper implements a higher-level cryptographic task (e.g., communication over a secure channel or encrypt data using a given password) using one or more of the supported APIs. To trigger the code generation, the user has to click on the OpenCCE icon in the tool bar. Figure 1 shows the window that opens where the user can select one of the tasks they want to implement.

The user then has to answer a few high-level questions (e.g., How important is performance to you, Do you want to implement the server or the client?). Each answer translates into a requirement or a constraint that the solution must satisfy. As a very simple example, if the user is configuring a symmetric encryption task as shown in Figure 2, and specifies that security is important to them, a constraint that the key size of the cipher should be greater than or equal to 128 bits will be created. When all questions for a particular task have been answered, OpenCCE suggests a list of possible algorithms and their configurations, from which the user may select one. Figure 3 shows how OpenCCE displays the list. After selecting a configuration, the wrapper code that implements the task using the selected configuration is generated into the application developer's project.

The default mode of operation of OpenCCE is the *beginner mode* where the application developer answers high-level questions as explained above. An alternative mode of operation is the *expert mode* where the developer configures the individual properties of the algorithms that the selected task consists of. For example, a more experienced developer can select ">= 128" for the key size of a cipher.

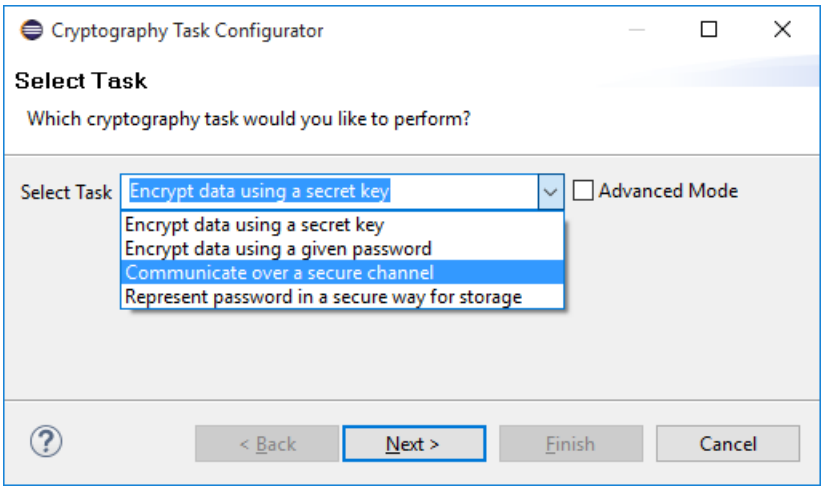

Figure 1: OpenCCE Task Overview

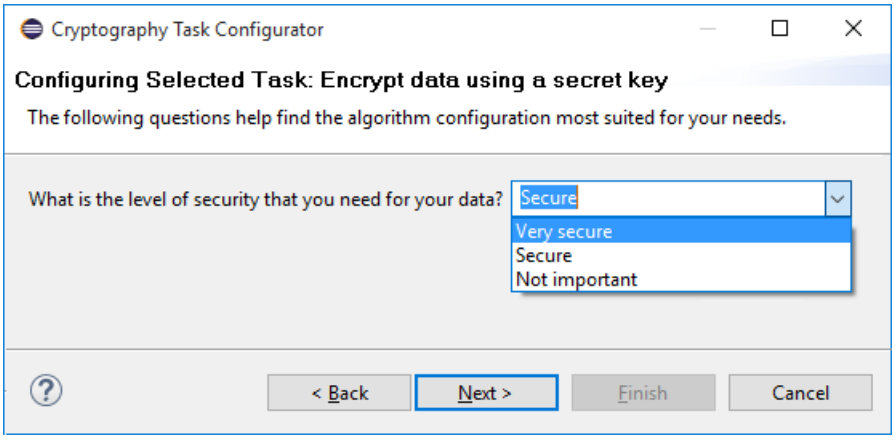

Figure 2: Configuration Question for Symmetric Encryption Task

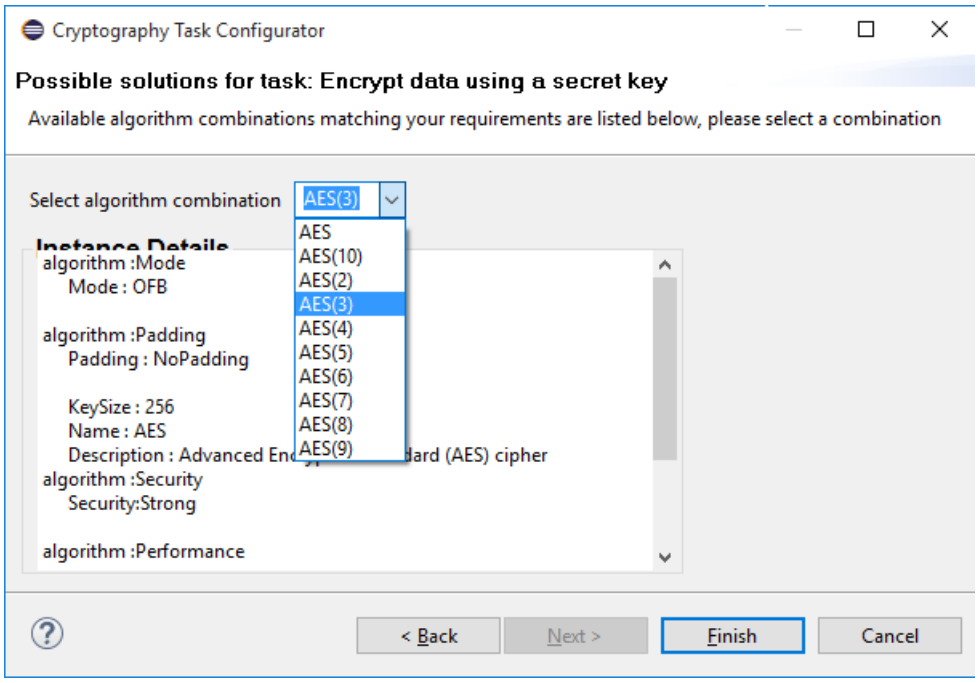

Figure 3: Algorithm Selection for Symmetric Encryption Task

The static analyses that OpenCCE runs in the background check the application developer's code for misuses of crypto APIs or related code (e.g., generation of key material). In case such a misuse is detected, OpenCCE generates an Eclipse error message.

3 How to contribute to OpenCCE

In general, crypto experts can contribute in two ways. First, they can contribute support for new primitives (e.g., encryption schemes, key agreement algorithms, digital signature algorithms). Second, they can integrate new cryptographic tasks that can be offered to users of OpenCCE. In this guide, the term cryptographic task refers to a more or less complex component that involves one or multiple cryptographic primitives. Examples for cryptographic tasks include file encryption, communication over secure channel, and user authentication mechanisms. Depending on how a task is integrated into OpenCCE, it can use primitives that have been integrated before, but primitives are not directly exposed to the end user. For easier distinction, this guide separates contributing crypto experts into primitives developers and task developers.

In order to integrate a new primitive/task, three components need to be provided.

- Implementation
- Usage example and rules
- Model describing the involved algorithms

The *implementation* should encompass the full functionality of the primitive/task. The *usage example* in general represents the way the functionality of the primitives/task should be accessed. To prevent developers from misusing primitive/task, *usage rules* describe how they are supposed to be used. These rules are translated into static analyses that are run when the application developer is writing the code, not only once it is run. This enables developers to correct their mistakes before deploying their application. These rules may also help application developers who do not use OpenCCE's code generation functionality, but want to implement a cryptographic task themselves as the static analyses might still alert them to API misuses. Finally, the *model* describes the used cryptographic algorithms and the attributes relevant to determine their security level. Throughout, the documentation shows only excerpts of the model. Contributors who are interested in a different part or the full model are referred to the OpenCCE maintainers.

3.1 Integrating Primitives

Implementation

Primitives need to be implemented as Cryptographic Service Providers (CSP) in Java. The JDK does not simply provide a set of implementations of cryp-

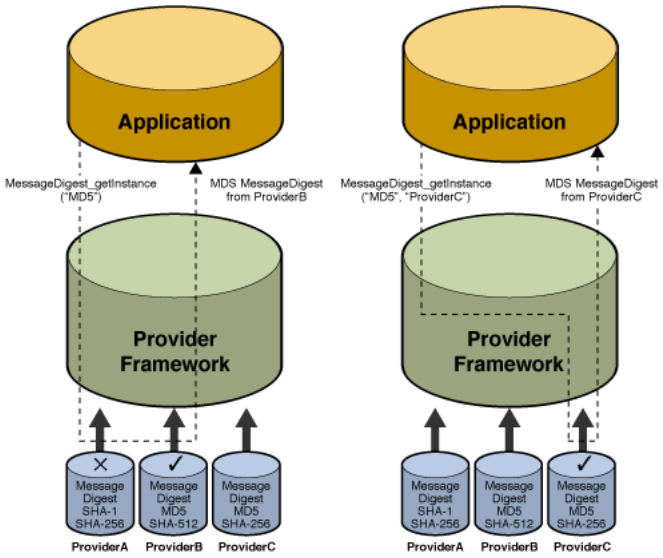

Figure 4: CSP Selection during Runtime

tographic primitives to its users. Instead, a set of standardized and algorithm-specific interfaces is provided by the Java Cryptography Architecture(JCA). These interfaces must then be implemented by CSPs in order to provide cryptographic primitives to an application developer. The goal of this design is to make the architecture around cryptography both independent of algorithms and implementations as well as easily extensible. It enables primitives developers to implement their own algorithms as CSPs and plug them into the JCA. One CSP may include multiple primitives of multiple algorithm types.

The JCA comes with a number of default providers that implement the most common cryptographic algorithms in several configurations. BouncyCastle, another cryptographic library for Java, can also be used as a CSP. It implements a wide range of cryptographic algorithms and configurations, including the ones implemented by the default CSPs.

During runtime, the Java Virtual Machine (JVM) maintains an ordered list of all plugged in CSPs. As illustrated in Figure 4, when an algorithm in a certain configuration is requested by some application code, the JVM iterates through the list and asks each provider if it supports the requested algorithm and configuration. In case several CSPs support the same configuration, the implementation of the first CSP in the list to support the configuration is selected.

Oracle provides extensive documentation on both JCA and CSPs as well as on how to implement a Cryptographic Service provider. Readers who wish to implement their primitives in C/C++ are referred to Chapter 4.

Usage Example and Rules

During the code generation, OpenCCE generates code into the application developer's project, that uses a CSP to perform a cryptographic operation (e.g. encryption, computation of a MAC). To ensure a secure usage of the implemented CSP, primitives developers should first provide a code snippet as a secure and correct usage example. OpenCCE can then use this snippet as the code it generates into the project as is shown in Listing 1. The Listing illustrates a simple encryption with Java Crypto APIs that avoids usual pitfalls (as described below).

```
1 public byte[] encrypt(byte[] secret, SecretKey key) {
2     byte [] ivb = new byte [16];
3     SecureRandom.getInstance("SHA1PRNG").nextBytes(ivb);
4     IvParameterSpec iv = new IvParameterSpec(ivb);
5
6     Cipher c = Cipher.getInstance("AES/CBC/PKCS5Padding");
7     c.init(Cipher.ENCRYPT_MODE, key, iv);
8     return c.doFinal(secret);
9 }
```

Listing 1: Secure Usage Example of Symmetric Encyption in Java

Furthermore, developers of primitives should provide usage rules for their CSP, which OpenCCE transforms into static analyses. The static analyses then makes sure the application developer's code is not breaking any of the specified usage rules.

To illustrate the necessity for such rules, consider the following two examples. The code snippet in Listing 2 is a simple example of a symmetric encryption using the JCA. The call in Line 11 instantiates an object of class `Cipher`, which is using the symmetric block cipher AES. Although not obvious at first glance, method `getInstance()` does in fact not expect a single algorithm, but instead a *transformation* consisting of cipher, block chaining mode and padding scheme. If the latter two are not provided by the developer (as in the code snippet below), the exact behaviour depends on the selected CSP. For the default CSP of the JCA, this means, that ECB is selected as block chaining mode by default. Unfortunately, ECB encrypts identical plaintext blocks to identical ciphertext blocks. Consequently, any encryption of more than one block using this configuration is deemed insecure. To prevent such misuses, contributors may specify how to use their CSP correctly and securely. Currently, there is no formal specification language to describe such usage rules. We thus ask primitives developers to simply describe their usage rules in English until the appropriate language is developed. For the above example, such a rule could look like this: "When calling the method `Cipher.getInstance()`, the passed parameter must be of the following form: 'Cipher/BCM/PAD'. Any call with fewer parameters is insecure."

```
10 public byte[] encrypt(byte[] secret, SecretKey key) {
```

```
11          Cipher ciph = new Cipher.getInstance("AES");
12          ciph.init(Cipher.Encrypt_Mode, key);
13          return ciph.doFinal(secret);
14  }
```

Listing 2: Insecure Usage Example of Symmetric Encyption in Java

One might argue that, in the above example, one does not need usage rules but instead only reasonable and secure default values. This is not always the case, however. Assume a lattice-based encryption scheme. Being a public key encryption scheme, a key pair is needed for a successful encryption. Listing 3 illustrates how to generate a key pair using the JCA. In Class KeyPairGenerator, there are two methods initialize() that can both be used to initialize the object. Both methods need to be implemented for each algorithm the CSP should support key pair generation for. The first one in Line 19 takes an algorithm-specific parameter object as input that can be defined to include any number of parameter objects of arbitrary type. However, the second method of that name in Line 20 takes an **int** parameter as input. Following the official JavaDoc of the method, this int parameter is generally interpreted as the key size of the key pair that is to be generated. Unfortunately, one cannot simply generate key pairs for lattice-based encryption schemes based on a single key length as the keys depend on multiple different parameters. Since the method needs to be implemented nonetheless, the CSP developer may throw an UnsupportedOperation Exception when this method is called. While this solves the problem for the CSP developer, it makes the provider easy to misuse. In this case, a rule such as "When KeyPair.getInstance is called and the name of a lattice-based algorithm is passed as a parameter, the method initialize(int) must not be called." might be added to prevent a misuse. With this rule, OpenCCE can point the developer to the misuse at compile time already and prevent the buggy code from being deployed.

```
15  public static void main(String .. args) {
16      KeyPairGenerator kpg = KeyPairGenerator.
17          getInstance("LP");
18
19      kpg.initialize(new LPAlgorithmParameterSpec(...));
20      kpg.initialize(1024);
21
22      KeyPair kp = kpg.generateKeyPair();
23  }
```

Listing 3: Insecure Usage Example for Key Pair Generation in Java

The actual description is being developed at time of writing this. The rules in English will inform this process and will be translated into that language once the process is finished by the language designers. Primitive developers who provided rules for their CSP in English will not need to provide them again in the description language.

Model

In addition to implementation and usage rules, the algorithms that are to be integrated into OpenCCE need to be modelled in the variability modelling language Clafer. This is necessary since developers of tasks might not implement their tasks by using algorithms directly (e.g., an encryption using RSA) but may only specify certain requirements an algorithm needs to comply to (e.g., encryption with an asymmetric cipher with a minimal key size of 4096 bit) and leave the selection of the actual algorithm to the user of OpenCCE. The algorithms hence work as basic building blocks for the supported tasks. In this case, OpenCCE determines a list of potential algorithms and configurations based on the variability model.

In general, the model consists of all algorithms and their configurations OpenCCE supports at a given point in time. At the time of writing, a basic model consisting of a number of cipher algorithms as well as stumps for other classes of algorithms (e.g., MACs, key agreement, and key derivation algorithms) are part of the model. An abbreviated model is shown in Listing 4. Lines 24 to 36 define the algorithm classes cipher and asymmetric cipher and their attributes. To integrate lattice-based schemes into this model, all lines in green need to be added. This extension can be separated into three different kinds of changes. First, class Cipher gets an additional attribute quantum (see line 30) to specify whether an algorithm is pre- or post-quantum. Second, a new class `LatticeBasedCipher` and its attributes are added (see lines 38 to 42). This class is a subclass of AsymmetricCipher and inherits all its attributes. Third, from Line 44 to line 61, an actual lattice-based encryption scheme is modelled. This algorithm is modelled as an instance of the `LatticeBasedCipher` class and all its attributes (e.g. `messageSize`, `cipherSize`) as well as the attributes it inherited from AsymmetricCipher and Cipher are defined.

```
24 abstract Cipher
25     name → string
26     description → string
27     security → Security
28     performance → Performance
29     secProperty → AttackModel
30     quantum → XQuantum
31
32 abstract AsymmetricCipher : Cipher
33     keySizePub → integer
34     keySizeSec → integer
35     performanceEnc → Performance
36     performanceDec → Performance
37
38 abstract LatticeBasedCipher: AsymmetricCipher
39     msgSize → integer
40     cipherSize → integer
41     n → integer
```

```
42      q → integer
43
44  LP: Lattice
45      [ name = "LP" ]
46      [ description = "Linder-Peikert Scheme" ]
47      [ msgSize = 1 || msgSize = 2 || msgSize = 3 ]
48      [ n = 1 || n = 2 || n = 3 ]
49      [ q = 20 || q = 40 ]
50      s → integer
51      [ quantum = post ]
52      [ n = 1 ⇒ q = 20 && s = 6 && security = Broken &&
53          cipherSize = 22 * msgSize ]
54      [ n = 2 ⇒ q = 40 && s = 9 && security = Weak &&
55          cipherSize = 30 * msgSize ]
56      [ n = 3 ⇒ q = 40 && s = 7 && security = Medium &&
57          cipherSize = 36 * msgSize ]
58      [ keySizePub = n * n * 25]
59      [ keySizeSec = log * 26 ]
60      [ performanceEnc = Slow ]
61      [ performanceDec = Fast ]
```

Listing 4: Clafer Model for Lattice Based Encryption Schemes

3.2 Integrating Tasks

Tasks are the main way users of OpenCCE interact with the tool. When a user wants to implement a cryptographic task (e.g., communicate over a secure channel, encrypt data using a password), they can open the tool and select the respective task. OpenCCE then guides them through a set of questions, lets them select one combination of algorithms and algorithm configurations, and generates the necessary code.

For this to work properly, task developers are required to contribute four components. In addition to the three general components - **implementation**, **usage example and rules**, and **variability model** - they also have to provide the **high-level questions** that OpenCCE asks the user in the beginning to configure the task. To simplify the extension of the model and the development of the questions, these steps may be conducted collaboratively with the OpenCCE developers. Shape and form of all of these components depend on the exact way the task is being integrated.

Implementation

The implementation for a task may be provided to OpenCCE as source code or a jar file. In this case, end-users get direct access to the code. If task developers do not wish to share the source code, they may provide access to the task's functionality indirectly through an API. Suppose a task developer wants to

offer long-term document archiving as a cryptographic task in OpenCCE. An archive requires the documents to stored on a hard-disk. To not require the user to manage these files on their own, the task developer offers their archive as a web service that stores the documents on their server. In this case, the task developer does not have to provide any implementation directly to OpenCCE.

A task developer may choose to let the application developer configure the task in terms of used algorithms among other things. This configuration is done through the dialogue system after the OpenCCE user selects a task they want to implement.

Usage Example and Rules

Regardless of how OpenCCE users access a task's functionality, task developers need to provide a usage example for the implementation. If the implementation is provided as source code or a jar file, the usage example simply illustrates how one may use the implementation. If the end-user has to access the functionality through an API, the generated usage code should reflect how the API should best be used. In the above example of a document archiving web service, the usage example should showcase how to use important calls like `addDocumentToArchive()`, `retrieveFileFromArchive()`, and `verifyDocument()`.

During the code generation, OpenCCE generates the usage example into the class the application developer has currently opened. A usage example for a symmetric encryption task is shown in Listing 5. At first, a secret key is generated using the `KeyGenerator` class. Then, the key and the plain text are passed as parameters to the actual encryption method, which is shown in Figure 2.

```
62 public void performEncryption(byte[] plaintext) {
63     KeyGenerator keyGen = KeyGenerator.getInstance("AES");
64     keyGen.init(256);
65     SecretKey key = keyGen.generateKey();
66
67     Enc enc = new Enc();
68     return enc.encrypt(plaintext, key);
69 }
```

Listing 5: Secure Usage Example for Symmetric Encryption Task

On top of that, task developers must provide usage rules for their usage example as well as the implementation if the user is not only accessing it through an API. Since there is no formal description language for this specification at the time of writing, rules in plain English suffice. These rules in English will serve as examples when designing the actual description language in English will not need to provide them again in the description language.

Consider the usage example for a symmetric encryption task from Listing 5 again. Since this code is generated directly into the application developer's

project they might alter the code. They could, for instance, generate the key as it is shown in Listing 6. In that listing, the key material is a fixed hard-coded string (see Line 72), while the class KeyGenerator is using a CSPRG for the key material. Since the key would no longer be random and could be retrieved by decompiling the respective Java class file, the encryption cannot be deemed secure anymore. A rule for the static analysis to prevent this misuse could look like this: "The key passed to the encrypt method must be created using the KeyGenerator class for the respective algorithm."

```
70 public void performEncryption(byte[] plaintext) {
71     SecretKey secretKeySpec = new SecretKeySpec("key"
72         .getBytes(), "AES");
73
74     Enc enc = new Enc();
75     return enc.encrypt(plaintext, key);
76 }
```

Listing 6: Insecure Usage Example for Symmetric Encryption Task

In terms of usage rules for the actual task implementation, consider Listing 2. It shows a possible implementation of method encrypt() that is being called in the previous code snippet and illustrates how to potentially implement an encryption in Java. This implementation of the symmetric encryption task, however, contains a misuse of the underlying API. For a more detailed description of the misuse, readers are referred to Section 3.1. As stated there, a rule such as "When calling the method Cipher.getInstance(), the passed parameter must be of the following form: 'Cipher/BCM/PAD'. Any call with fewer parameters is insecure." would prevent the misuse in the given case and OpenCCE would generate an error message.

Model and Configuration Questions

In general, task developers should let the the end user configure their implementation. For an appropriate configuration point, assume that for a task an asymmetric encryption needs to be performed. Instead of implementing an encryption using, say, RSA with a fixed key length, the developer may outsource this decision to the OpenCCE user. To ensure that OpenCCE presents the end-user with only appropriate algorithms to pick from, the requirements are specified in the model in the variability modeling language Clafer.

To accomplish this, the given task must be modelled top-down from the task to the algorithms, which serve as basic building blocks. Consider the symmetric encryption example in Listing 7. In Lines 77 to 90, cipher related algorithm classes are defined. The following lines define two instances of symmetric block ciphers, namely AES and DES. Finally, in Lines 108 to 111, the task element is defined as using one algorithm of type SymmetricBlockCipher. All components of the task must be modelled until they are mapped to one of the algorithm classes. In the case of the symmetric encryption task, this goal is already reached as it directly uses a symmetric block cipher. When defining more

complex tasks, several layers between the task element and the basic building blocks might be necessary.

```
77  abstract Cipher
78      name → string
79      description → string
80      security → Security
81      performance → Performance
82
83  abstract SymmetricCipher : Cipher
84      keySize → integer
85
86  abstract SymmetricBlockCipher : SymmetricCipher
87      mode → Mode
88      padding → Padding
89      [mode != ECB]
90      [mode = CBC => padding != NoPadding]
91
92  AES: SymmetricBlockCipher
93      [name = "AES"]
94      [description = "Advanced Encryption Standard"]
95      [keySize = 128 || keySize = 192 || keySize = 256]
96      [keySize = 128 ⇒ performance = VeryFast &&
97          security = Medium]
98      [keySize > 128 ⇒ performance = Fast &&
99          security = Strong]
100
101 DES: SymmetricBlockCipher
102     [name = "DES"]
103     [description = "Data Encryption Standard"]
104     [performance = VeryFast ]
105     [security = Broken ]
106     [keySize = 56 ]
107
108 SymmetricEncryption : Task
109     [description = "Encrypt data using a secret key"]
110     cipher → SymmetricBlockCipher
111     [cipher.security > Medium]
```

Listing 7: Clafer Model for Symmetric Block Ciphers

Lastly, if developers want to allow the end user of OpenCCE to configure their task implementation, they are also required to provide the questions OpenCCE can ask the user. These questions can include a precise specification of the expected security level, as is the case in Figure 2, and their answers may influence the variability model as well as the generated code.

4 JNI

Developers of cryptographic components who want to contribute to OpenCCE must provide their primitives as CSPs, but might nonetheless wish to implement their primitives in C or C++. In this case, the C(++) implementation needs to bridged to Java. There are multiple ways of doing this, but the tutorial below focusses on the Java Native Interface(JNI).

JNI can be used to bridge a C(++) and a Java program. The setup for JNI is different on different plattforms, although the general steps are the same. Using it on Windows is not recommended, as it leads to configuration issues, but the tutorial below nevertheless details the steps necessary to set JNI up on a Windows machine. Users of Linux and MacOS may skip step 0.

0. Install MinGW and add `$Path_to_mingw/bin` to PATH variable.

   MinGW is a development environment that provides windows with c and c++ compilers gcc and g++ among other things.

1. Install JDK and add `$Path_to_JDK/bin` to PATH Variable.

2. Create sample Java project with the following Java class (e.g. HelloWorld.java)

```
1  public class HelloWorld {
2      native String helloFromC();
3      static {
4          System.loadLibrary("ctest");
5      }
6      static public void main(String argv[]) {
7          HelloWorld helloWorld = new HelloWorld();
8          System.out.println(helloWorld.helloFromC());
9      }
10 }
```

Listing 8: HelloWorld.java

   `helloFromC()` is the name of the method the c(++) library needs to provide. `ctest` is the name of the library (i.e., the c(++) program). The library has to be part of Java's library path. As the current path of a java program is always part of the Java library path, it is easiest to have all files in the same directory for the first attempt. In the main method, the c(++) method `helloFromC()` is called and its return value is printed.

3. Open console and navigate to the JNI project folder.

4. Compile java program by running `javac HelloWorld.java`.

5. Run `javah HelloWorld` to generate header file.

6. Open the header file, which should look like this:

```
11 /* DO NOT EDIT THIS FILE - it is machine generated */
12 #include <jni.h>
13 /* Header for class HelloWorld */
14
15 #ifndef _Included_HelloWorld
16 #define _Included_HelloWorld
17 #ifdef __cplusplus
18 extern "C" {
19 #endif
20 /*
21  * Class:     HelloWorld
22  * Method:    helloFromC
23  * Signature: ()Ljava/lang/String;
24  */
25 JNIEXPORT jstring JNICALL Java_HelloWorld_helloFromC
26         (JNIEnv *, jobject);
27
28 #ifdef __cplusplus
29 }
30 #endif
31 #endif
```

Listing 9: HelloWorld.h

Line 25 show the method signature of the method the java class expects to be provided by the library. This method needs to be implemented for the interface to work.

7. Create a C(++) source file (e.g., ctest.c) and copy&paste the signature from the header file.

8. Write implementation for method.

```
32 #include <jni.h>
33 #include <stdio.h>
34
35 JNIEXPORT jstring JNICALL Java_HelloWorld_helloFromC
36   (JNIEnv* env, jobject obj) {
37
38    return (*env)->NewStringUTF(env,"Hello from C!\n");
39 }
```

Listing 10: ctest.c

Please note: The method signature in the header file does not contain any parameter names. If the signature is copy-pasted into the source file, parameter names must be added to make the program compilable.

9. Compile c(++) source file into library by running

```
gcc -o ctest.$FE -shared -I$Path_to_JDK\include
        -I$Path_to_JDK\include\$OS ctest.c -lc
```

   $FE is the platform-dependent file extension for the library. Please refer to `https://stackoverflow.com/questions/37203247` for the mapping. $OS is the operating system that the compiler is running on (e.g. 'win32' for Windows systems, 'linux' for Linux systems)
   Please note: If the path to the JDK includes a whitespace (e.g. C:/Program Files(x86)/...), compilation errors can be resolved by enclosing the path in quotation marks.
   Please note: In case of an error message along the lines of "unknown type name '____int64' ..." please follow the instructions on this website: `http://www.graphics-muse.org/wp/?page_id=147`

10. Run java program by running `java HelloWorld` in the console. If everything works, the program should print 'Hello from C!'.