Grant agreement no: 644729

European Commission

SAFEcrypto

Secure Architectures of Future Emerging Cryptography

Software Requirements Specification	
Deliverable	D6.1
Author(s)	Markku-Juhani O. Saarinen, Elizabeth O'Sullivan
Version	1.0
Status	Approved
Date	April 22, 2016
Classification	☒ **White – public** ☐ **Green – restricted to consortium members** ☐ **Yellow – restricted to access list given below** ☐ **Red – Highly sensitive information, access list only**
Access List	

Queen's University Belfast CSIT CENTRE FOR SECURE INFORMATION TECHNOLOGIES

HWCommunications Cyber Security and Resilience

RUHR UNIVERSITÄT BOCHUM RUB

RSA EMC²

THALES

Inria INVENTEURS DU MONDE NUMÉRIQUE

Università della Svizzera italiana

Executive Summary

This document describes the desired properties and features (“requirements”) for a suite of reusable, production quality software that implements lattice-based cryptography within the Horizon 2020 SAFEcrypto project. The document is mainly intended for guiding architecture and implementation.

As final algorithm and parameter selection is a continuous process, the actual algorithms and their precise properties are not defined in this document, nor is the architecture of the final software suite.

This document defines a uniform programming style and methodology that allows the software suite to meet the needs of various SAFEcrypto case studies and anticipated applications.

The software suite can be targeted for both resource-constrained embedded platforms and high performance multi-core architectures. The software is mainly written in performance-optimized C99 language, and is mainly self-contained in the sense that few external libraries are required. Assembly language optimizations can be added to “bottleneck” positions but all functionality must be available in highly portable C. We require the code to be written in such a way that it is easily maintainable for at least 15 years.

The Application Programming Interface (API) of the new suite is similar to that used by OpenSSL and derivatives such as BoringSSL. This helps to integrate the new public key algorithms to existing applications that already use this de facto standard API. External cryptographic primitives such as symmetric ciphers, MAC and hash functions, and true random numbers should be similarly accessed through an OpenSSL-like programming interface. Some of these primitives may be indigenously implemented within the SAFEcrypto suite for embedded targets, but eventually the project wishes to integrate the Lattice Cryptographic algorithms into a wider multi-purpose cryptographic framework such as BoringSSL.

This is a high-assurance software development project. During development, emphasis is on quality and correctness of code. An automated test system is continuously maintained for all components of the software suite. Automated tools, formal analysis, and code audits are performed for releases.

The resulting software suite and all of its components will be released as Open Source under a highly permissive license.

Table of Contents

1 Introduction 5
1.1 Purpose and Goals 5
1.2 Intended Audience 6
1.3 Scope: This is not an Architecture Document 6
1.4 General Description 6
2 Functional Requirements 7
2.1 Background 7
2.2 Functionality 7
2.3 Operating Environment 8
2.4 User Documentation 9
2.5 Assumptions and Dependencies 10
2.6 Application Programming Interfaces 10
2.7 Baseline Cryptographic Performance Targets 10
3 Nonfunctional Requirements 11
3.1 Portability and Implementation Language 11
3.2 Secure Implementation Techniques 11
3.3 Scalability 12
3.4 Configuration Management 12
3.5 Testability and Test Suites 12
3.6 Performance optimisation 13
3.7 Reliability, Safety, and Security Assurance 13
3.8 Licensing 14
4 References 16

Glossary

ABE	Attribute-Based Encryption
AES	Advanced Encryption Standard
API	Application Programming Interface
ASN	Abstract Syntax Notation
AVX	Advanced Vector Extensions, a SIMD instruction set extension for Intel architectures
BLISS	Bimodal Lattice Signature Scheme
DES	Data Encryption Standard
CERT	Computer Emergency Response Team (originally organized by Carnegie Mellon University)
DLP	Discrete Logarithm Problem (in finite fields)
DTLS	Datagram Transport Layer Security. A datagram mode of TLS.
ECC	Elliptic Curve Cryptography
ECDLP	Elliptic Curve Discrete Logarithm Problem (in elliptic curve groups)
ECDSA	Elliptic Curve Digital Signature Algorithm
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
GPL	GNU General Public License
IBE	Identity-Based Encryption
KAT	Known Answer Test
KEX	Key Exchange (protocol)
LGPL	GNU Library or “Lesser” General Public License
MAC	Message Authentication Code
NEON	SIMD instruction set extension for ARM architectures
NTRU	A Lattice-Based Public Key Cryptosystem
NTT	Number Theoretic Transform. A finite field analogue of FFT.
PKCS	Public Key Cryptography Standard (originally from RSA Data Security Inc.)
R-LWE	Ring Learning with Errors (a hard computational problem in cryptography)
RNG	Random Number Generator
RSA	A Public Key Cryptosystem named after Rivest, Shamir, and Adleman
SIMD	Single Instruction, Multiple Data (microprocessor architecture)
SOC	System-on-Chip
SSL	Secure Sockets Layer
TLS	Transport Layer Security (protocol)
X.509	A standard for Certificates and Public Key Infrastructure
XOR	Exclusive-or (A Boolean logic operation)

1 Introduction

1.1 Purpose and Goals

This document describes the desired properties and features ("requirements") for a suite of reusable, production quality software that implements lattice – based cryptography within the Horizon 2020 SAFEcrypto project. We refer to the resulting open source software deliverables simply as the "software suite".

In addition to the desired functional goals of this suite, we define a uniform programming style and methodology that allows the software suite to meet the needs of the various SAFEcrypto case studies and anticipated applications.

We define the goals and priorities of the software developed within this project as follows:

1. **Correctness.** The suite must implement its algorithms faithfully for all allowed inputs. A set of automatic scripts, test vectors and other data must be available for all components.
2. **Readability and documentation**. The code must be clear and understandable, and written in good, consistent style. All components must be clearly documented within the code. Programming interfaces and tools should also be externally documented.
3. **Flexibility.** The suite is designed for maximum modularity and allows easy configuration. It must be possible to create libraries that contain strictly only those components that are required for a given application or target platform.
4. **Code security.** Code must be robust enough so that it does not form a "weak link" in an application that utilizes the cryptographic functionality provided by the suite. Strict security practices for writing, auditing, and verifying code must be adhered to from the start.
5. **Cryptographically secure development methods**. In addition to general secure coding practices, cryptographic software requires special considerations to prevent leakage of confidential information in timing and physical attacks.
6. **Integration.** The effort required to integrate the suite with existing applications that utilize other cryptographic algorithms should be minimum. Furthermore, where standard Application Programming Interfaces exists for cryptographic primitives outside the scope of the project, these should be used.
7. **Portability**. All functionality and much of the code must work on a wide spectrum of target platforms, from low-end microcontrollers to high performance server systems. Target-specific optimizations should be made to core components (or bottlenecks) where the gained speedup or other advantage is high.
8. **Performance**. Algorithms must be implemented efficiently. For many arithmetic subtasks multiple algorithms are available; the project generally always implements the algorithm with best asymptotic complexity.
9. **Lifespan and maintainability**. Tools and techniques should be chosen in a way that one can expect the code to be easily maintainable for at least 15 years. Tools that experience frequent interface or syntax changes should be avoided.

1.2 Intended Audience

This document is primarily intended for developers working on lattice-based cryptographic software architectures and implementations within the SAFEcrypto project.

This document may also be referenced for information about requirements for software assurance methods, and the rationale for selection of various development tools and techniques.

1.3 Scope: This is not an Architecture Document

This Software Requirements Specification document specifies the properties and requirements for the main software deliverables within the Horizon 2020 SAFEcrypto project, Work Package 6: Lattice-Based Cryptographic Software Architectures.

In addition to desired functionality, we define the following goals for the software suite:

- **Portability**. General description of target systems and techniques to achieve portability.
- **Scalability**. Targets for scalability of implementations on different targets, from low-end embedded systems to high-end multicore server systems.
- **Configuration**. General techniques adopted for configuration of software components to fit various targets.
- **Testability**. Methods to ensure correct implementation and compilation.
- **Performance**. General performance targets.
- **Resilience**. Ability to maintain service in presence of faults.
- **Reliability**. Very low probability of failure of operations.

Detailed architecture of the software suite will be described in deliverable D 6.2. We only give a high- level description in this document. We cannot currently fully specify the algorithms, parameters, and implementation techniques provided by the software deliverables, as the science defining those is in a flux at time of writing. We wish the deliverables to reflect state-of-art at time of delivery, yet be implemented in a consistent way so that they all fit into the same framework.

1.4 General Description

The software described in this document is intended to satisfy requirements of use cases first outlined in D9.1 "Case Study Specifications and Requirements". The three main case studies are:

- Satellite Key Management
- Commercial Off-The-Shelf (COTS) in Public Safety Communication
- Privacy-Preserving Municipal Data Analytics

We refer to that document for more detailed requirements. Essentially at least one practical cryptographic construction is required for Digital Signatures, Authentication, Attribute-Based Encryption, and Identity-Based encryption; this is largely what dictates the functional requirements for the software suite.

At the time of writing we are not aware of any comparable, production-quality software suites that provides this functionality. We intend to develop the main bulk of the new software library source code within the SAFEcrypto project (possibly using existing external Open Source components where appropriate.)

2 Functional Requirements

2.1 Background

We define a set of software requirements for both resource-constrained embedded platforms and high performance multi-core architectures for use within the SAFEcrypto project and beyond.

In the context of the SAFEcrypto project, the objective of Work Package 6 is to develop a suite of software routines implementing the constructions identified during WP4 to address various use cases. However, the development of these constructions and their parameter selection is an ongoing process. Our purpose is to produce a highly portable, flexible, efficient, general framework for lattice cryptography that we expect to have a long lifetime and high industry impact.

2.2 Functionality

The primary function of the software suite is to support the use cases of the SAFEcrypto project with production-quality implementations of appropriate lattice-based cryptographic algorithms.

A secondary function is to provide a suite of algorithmic implementations that can be integrated into existing applications with relative ease. As such, the new lattice public key algorithms can be used as "drop-in-replacements" to more traditional algorithms such as RSA and ECDSA. The API calling convention of the new primitives should be consistent with current practices in standard C language toolkits (OpenSSL and BoringSSL).

The mechanisms for importing and exporting (portably serialising for transmission or storage) of public and private key information should follow the same outline as that for existing algorithms. One of the main support functionalities provided by the library is integration of new lattice primitives with the OpenSSL certificate processing facilities. All public key operations should be written in a way that supports existing (or future) standards in this field.

Integration with hardware implementations will be done through a generic mechanism that resembles the OpenSSL "engines". Applications should be – as far as possible – able to transparently use the same APIs for hardware cryptography as they would for corresponding software implementations. A functionally equivalent software implementation should be available for all hardware components. The kernel-level driver that interfaces the hardware component to user space may be application and operating-system dependant. On a "bare metal" embedded platform the driver may be incorporated into the suite itself.

Lattice-based cryptography is still an evolving field. The software suite should provide algorithms and security parameters that can be expected to provide long-term security according to latest research at time of release. Algorithm and parameter selection is addressed in Work Package 4 ("Lattice-Based Constructions") of the project.

If a major international standard in Lattice-based cryptographic primitives emerges, the software suite will provide a secure implementation of it.

<u>The main functionality is provided by C-callable primitive implementations of</u>:

1. **Lattice-based digital signatures**. The implementation must be able to create, export, and import public and private keys for Lattice-based digital signatures. Furthermore, it must be able to create signatures for arbitrary data, and verify them. Compact serialization (encoding and decoding) methods should be available for signatures and keys, preferably compatible with ASN.1 coding and X.509 PKI Certificates, in analogous fashion to PKCS #1 (the standard for RSA cryptography). The encoding will be handled via OpenSSL (or similar) APIs and mechanisms.

2. **Lattice-based public key encryption algorithms**. The implementation must be able to create, export, and import public and private keys for Lattice-based public key encryption. The methodology should be flexible and based on hybrid encryption methods in conjunction with appropriate symmetric ciphers such as AES. Compact serialization methods for encrypted messages should be available. Preferably the methods would be compatible with ASN.1 and S/MIME for message encryption, in analogous fashion to PKCS #1. The encoding will be handled via OpenSSL (or similar) APIs and mechanisms.
3. **(Optional) Lattice-based key-exchange algorithms (KEX)**. A key-exchange algorithm offers functionality that is similar to the Diffie-Hellman method for establishing a shared secret in a secure communications protocol. Efficient encoding of messages should be available, together with parameters that can be assumed to be secure.

<u>**Furthermore, the applications using the software suite must be able to demonstrate**</u>:

4. **Lattice-based authentication**. Authentication methods can be based on public key signature or encryption algorithms, or some other method which is based on lattice cryptography. Authentication will be integrated into TLS and DTLS protocols in conjunction with an appropriate KEX algorithm. It is possible that the satellite use case will require IPSec support; this requirement will be reviewed following a PKI Space Workshop arranged by the project.
5. **Attribute-based encryption (ABE)** and **Identity-based encryption (IBE)**. Case study applications and suitable methods of implementation are currently unknown. However, when suitable algorithms become available, the software suite should allow ABE and IBE to be implemented. The implementations may be separate demonstration programs.

The deliverable must be appropriately modular with clearly defined APIs for various mathematical algorithms and functions which are specific for lattice-based cryptography. A partial list of such functional requirements:

- **Non-Uniform Sampling**: Many of the older Lattice algorithms required very precise discrete Gaussian sampling. This requirement is lessening due to Rényi divergence. Some newer algorithms can use uniform or binomial distributions. The main challenge is constant-time implementation.
- **Ring Arithmetic using Number Theoretic Transforms**: NTT is used to speed up ring polynomial arithmetic to log-linear time. This also needs to be constant time. This subcomponent implements appropriate rings for R-LWE systems.
- **Matrix Algebra in small finite fields**: General matrix arithmetic over small prime fields to implement schemes that utilize general lattices.
- **Encoding and Compression:** In many lattice-based cryptographic schemes signatures, keys, and other stored and transmitted quantities do not come from uniform distribution and must be compressed and encoded in some standard way. Huffman codes and Arithmetic Coding are leading candidates for this work.

These algorithmic building blocks are discussed in more detail in separate Deliverable 5.1 "Evaluation Report of Efficiency of Lattice-based Constructions."

2.3 Operating Environment

The software suite must be designed to be portable to a wide range of target platforms, from high-end server systems to low-end microcontrollers and embedded systems (where possible).

Most of the software suite functionality (and code) must be available for the following primary targets:

1. **Minimal embedded target**: This microcontroller target has industrial unit price of approximately €5.00 for the self-contained SoC chip alone. Typically, a ``bare-metal'' (no operating system) ARM Cortex M3 or Cortex M4 based microcontroller system. Target has 32-bit hardware multiply and division, but no (IEEE 754) floating arithmetic.
2. **High-end multi-core target**: This high-performance system has acquisition price of approximately €2,000.00 for a single (rack-mountable) compute node. This is typically a 64-bit multi-core (and optionally multi-CPU) Intel and/or AMD - based system that has all of the common instruction set extensions (AES NI, AVX2, AVX3, etc.) The target runs Linux.

An additional mid-range target that has similar performance characteristics to current low-end consumer handsets is optional:

3. **An optional mid-range mobile target**. This target has capabilities similar to a contemporary mid-range smartphone and has a manufacturing cost of approximately €100.00. The target has a single- or dual core (32/64 - bit ARMv7 or ARMv8 architecture) with NEON and other standard SIMD instruction set extensions. The operating system is Linux Android.

We note that the primary low- and high-end targets have a 1:400 unit price difference, while the third target sits halfway between the two, with 1:20 price difference to both primary targets. The storage capabilities of the two primary targets differ by six orders of magnitude (1:1000000).

With 2016 technology, these requirements translate roughly to following technical specifications given below:

Table 1 Technical features of Primary Targets.

Target	**1. Resource-constrained Embedded target**	**2. High-performance multi-core target**	**3. Mid-range mobile target (OPTIONAL)**
Budget	€5.00 for the SOC chip.	€2,000.00 for system.	€100.00 to manufacture.
Operating system	Bare metal + libraries.	Linux or similar.	Android or similar.
Processor and its capabilities	ARM Cortex M3 or M4; 32-bit, no floating point, no cache.	High-performance Intel or AMD 64-bit with SIMD (AVX2 or AVX3).	Mobile ARMv7 (32-bit) or ARMv8-A (64-bit) SoC with SIMD (NEON).
Memory and storage	32kB SRAM, 1 MB flash memory.	32GB DRAM, 1 TB hard drive.	1 GB RAM, 16 GB flash memory.

2.4 User Documentation

Detailed documentation should exist for following:

- Architectural documentation giving details of which cryptographic algorithms have been implemented and the interdependencies between different modules.
- All aspects of the Application Programming Interface; function calls, conventions, and data structures used. These may be made available in the style of ``man pages'' or using automated tools such as Doxygen [1].
- Tutorials and other hands-on-documentation, especially for tasks such as configuring the software for different target platforms.

2.5 Assumptions and Dependencies

Implementation work is dependent on the availability and stability of appropriate Lattice-based post-quantum cryptographic algorithms. The specification of these algorithms is the main task of Work Package 4. Implementation work primarily focuses on new methods and algorithms that may reach standardised status, and their support functions. The patent and intellectual property status of some older algorithm suites (such as NTRU) may restrict their suitability and usefulness within the library.

2.6 Application Programming Interfaces

Where possible, the project makes primitives such as lattice-based digital signature algorithms and public key encryption available via an OpenSSL/BoringSSL - like API to make integration with standard cryptographic libraries and existing applications as easy as possible.

As Lattice-based cryptographic software frequently uses conventional cryptographic components, such as hash functions and symmetric encryption, we have chosen to utilise an OpenSSL/BoringSSL - like external API for those. Such an API is to be used for primitives such as:

- Symmetric cryptographic primitives such as hash functions, message authentication codes, stream ciphers, extensible-output functions, etc. will be accessed via an OpenSSL-like interface. For embedded targets some of these implementations may be provided via the SAFEcrypto suite itself to avoid unnecessary bloat.
- Cryptographically secure uniform random numbers are to be obtained via standard calling conventions from an OpenSSL or OpenSSL - like library. The conversion of uniform random numbers into non-uniform distributions (i.e. Gaussian Sampling) will be handled by the library itself without external dependencies.

2.7 Baseline Cryptographic Performance Targets

In comparison to current state-of-the-art implementations of conventional public-key cryptosystems (based on RSA problem, Discrete Logarithm (DLP), and Elliptic Curve Discrete Logarithm (ECDLP) - based primitives), SAFEcrypto's objective is to achieve a range of lattice-based architectures that provide comparable area costs, a 10-fold speed-up in throughput for real-time application scenarios, and a 5-fold reduction in energy consumption for low-power embedded and mobile applications.

We define the baseline conventional cryptography comparison to be the current OpenSSL implementation with the recommended key sizes from NSA Suite B [2]:

Algorithm	**Function**	**Parameters (Suite B)**
Elliptic Curve Diffie-Hellman (ECDH) Key Exchange [3]	Asymmetric algorithm used for key establishment.	Use Curve P-384 to protect up to TOP SECRET.
Elliptic Curve Digital Signature Algorithm (ECDSA) [4]	Asymmetric algorithm used for digital signatures.	Use Curve P-384 to protect up to TOP SECRET.
Diffie-Hellman (DH) Key Exchange [5]	Asymmetric algorithm used for key establishment.	Minimum 3072-bit modulus to protect up to TOP SECRET
RSA [4] [6]	Asymmetric algorithm used for digital signatures and encryption.	Minimum 3072 bit-modulus to protect up to TOP SECRET.

3 Nonfunctional Requirements

3.1 Portability and Implementation Language

The main body of the software must be portable between the targets defined in Section 2.3 . This multi-platform portability requirement, together with performance requirements, effectively dictates (plain) C as the main implementation language.

The main codebase is written in ANSI C99 (ISO/IEC 9899:1999) [7]. Use of some extensions such as C intrinsics and assembly language is encouraged for performance, but these code segments must be optional (via `#ifdef /#else` mechanism or similar) and plain C alternatives must exist.

1. Generally, utilize CERT SEI C Coding Standard [8] and Google / BoringSSL - type coding style and calling conventions [9].
2. Software must work on big- and little-endian systems, and on both 32- and 64-bit targets. Use explicitly-sized types from `stdint.h`: bytes are `uint8_t` (not `unsigned char`) 32-bit words are of `uint32_t`, etc. Sizes and related loop variables are of type `size_t` defined in `stddef.h`. Use appropriate macros to detect endianness.
3. C's Standard `assert` mechanism is allowed for basic sanity checks. The assert functionality is not intended for reporting error messages, and may be disabled in production builds.
4. As a rule, each software module should strictly use only those system services that it needs. Most cryptographic implementations do not require system services at all as some software components may have to be ported to ``bare metal'' targets. Therefore, the use of even the standard C library should be kept to a minimum.
5. Heap memory allocation should be avoided if possible, as memory management functionality may be restricted. Generally, the calling party should take care of memory management rather than the library itself. If needed, rather than `malloc()` and `free()`, use the wrappers `OPENSSL_malloc()` and `OPENSSL_free()`.
6. Library functions generally cannot assume an existence of standard input, output, or error streams. The library must be able to operate in an environment where there is no file system.
7. Due to embedded restrictions, we may not assume that all features of the floating point arithmetic and standard mathematics library are available.
8. Tag sensitive data and data structures so data flow and "taint" analysis is possible. The exact way this is done is defined in the Software Architecture document.

Ideally the same source code and Makefiles can be easily retargeted, simply by using appropriate open source cross-compiler toolchains, and GNU Build Tools.

3.2 Secure Implementation Techniques

We expect the implementation to follow CERT Software Engineering Institute C Coding Standards, as defined in [8]. The rules with a high severity risk assessment should be enforced and all rules with medium severity reviewed. Furthermore, we expect the implementation to note BoringSSL Coding Conventions [9].

Cryptographic software requires special implementation techniques. We refer to the Cryptographic Coding standard [10] for detailed instructions. Some of the basic requirements are:

1. Compare secret strings in constant time
2. Avoid branching controlled by secret data

3. Avoid table look-ups indexed by secret data
4. Avoid secret-dependent loop bounds
5. Prevent compiler interference with security-critical operations
6. Prevent confusion between secure and insecure APIs
7. Avoid mixing security and abstraction levels of cryptographic primitives in the same API layer
8. Use unsigned bytes to represent binary data
9. Use separate types for secret and non-secret information
10. Use separate types for different types of information
11. Clean memory of secret data
12. Use strong randomness

Furthermore, we require explicit validation of most input parameters and all data potentially coming directly from external sources

3.3 Scalability

The use cases of D9.1 "Case Study Specifications and Requirements" indicate a wide range of processing requirements for the implementations.

Implementations will scale across different target platforms from low-end embedded systems to high-end multicore server systems. Primarily the code should be targeted to 64-bit medium- to high-performance systems, but optimisations can be included for 32-bit targets. All data structures should be conservative in their memory usage.

Some applications such as web servers will maintain tens of thousands of context instances for cryptographic algorithms, so the code must be **fully thread-safe and have no global variables**. Also the size of contexts should be kept to a minimum so as to avoid exhausting the working memory in such cases.

3.4 Configuration Management

The project aims to use Open Source tools and compilers whenever possible.

We adopt the GNU Build System configuration techniques, also known as "autotools", which consists of Autoconf, Automake, and Libtool [11]. This means that there should be a configuration script that generates appropriate Makefiles for a target. For most target platforms configuration and installation of libraries should be effortless and follow the general style adopted in free software.

Also creation of packaged distributions and use of cross-compiler configurations should be relatively effortless. Scripts, tools, and instructions for generating all tables and constants should be included in the distribution; try to avoid magic constants and tables. These tables may have to regenerated for different parameters and tables.

3.5 Testability and Test Suites

All software components must be reachable via automated unit tests. In addition to functional tests (e.g. that a decryption operation correctly inverts encryption for given data) the software package must include an exhaustive set of known answer tests (KATs), which allow the algorithms to be tested against expected outcomes. These should cover, in addition to cryptographic algorithms and support libraries, related functionality such as encoding, decoding, and compression features.

For algorithms that use randomness as part of their operation, a special mode should be made available where the random source is entirely deterministic so that the outcomes can be verified with KATs. The formatting of KATs can be standardised based on plain ASCII files. If such test vectors are available in standards, those should be used.

The basic implementation correctness regression testing suite (which has a running time of few minutes at most) should be available with a simple instruction such as

```
$ make test
```

The distribution should also come with standardised tools and scripts to generate data on the throughput and performance of various software components. These can be used to tune implementation parameters.

The same regression testing methodology should also be applicable to hardware implementations that are accessible through the same APIs.

For some algorithms, it may be beneficial to implement test suites in another language, such as Go. This approach has been adopted in the BoringSSL project. For some functionality it may be advisable to test against Python or PARI/GP [12] scripts. We suggest including scripts that generate test data to be included in the distribution itself.

3.6 Performance optimisation

Performance optimisation should be performed on a best effort basis. Software should be no more than 15% slower than best-known implementation for any given algorithm. In algorithm parameter selection, security is the overriding priority before performance.

Asymptotically best algorithms are used for tasks such as ring arithmetic and parameter compression. Use of hand-coded assembly and utilisation of vector instructions (AVX2/AVX3 on Intel/AMD, NEON on ARM platforms) via C intrinsics is recommended.

Since we wish the code to be portable and to have a long shelf life, only the innermost loops and other bottlenecks should be optimised. A plain C implementation should be available for all components.

A trade-off is often required between software performance, side-channel security and source code lifetime -- constant time implementations are slower than maximally optimised implementations, and a constant time implementation is typically platform specific. In such a case the priorities are:

1. **Correctness**. A mathematically correct plain C implementation must always be available.
2. **Performance**. We are generally interested in the maximum performance of the scheme.
3. **Implementation size**. Implementation footprint is especially important for the embedded targets; sometimes it may be necessary to sacrifice some performance in order to make the algorithm usable on low-end targets.
4. **Constant-time and side-channel resistant operation**. Countermeasures may be optional as cryptographic algorithms are not always used in applications where an adversary can mount a timing- or other emissions-based attack.

3.7 Reliability, Safety, and Security Assurance

We list some required tools and guidelines related to reliability and quality of the code.

- **Use automated analysis tools.** Concentrated effort must be made to eliminate all errors that may lead to unpredicted behavior, memory leaks, and security vulnerabilities. Instrumentation, static, and dynamic analysis tools such as Valgrind [13] must be frequently used. Fuzzing and randomized testing can also catch a significant number of bugs.

- **Code reviews and Audits.** Regular code reviews and audits should be scheduled. Here production code is analysed by an experienced, independent reviewer to catch implementation bugs. Ideally 100% of code contained in releases has been independently reviewed. All release code must be reviewed by internal review team.
- **Security Assurance via Formal analysis.** For some key components it may be possible to utilize formal, automated reasoning tools such as Galois Inc's Crypto and Software Analysis Workbench (SAW) [14].
- **Handling failures.** There are cryptographic algorithms that may fail with a non-negligible probability. Such rare cases must be appropriately handled by the software library. Error conditions should propagate widely; Apart from debugging, an internal error condition should result in the failure of the entire session. Of course failure should be graceful and free all allocated resources, if possible.
- **Avoid information leakage in error messages.** In cryptographic applications it is usually advisable not to reveal any unnecessary information about private operations, even error conditions. Overly informative error messages may be utilized by an attacker as "Oracles" as was famously done with (SSL) protocol padding error messages in Bleichenbacher's PKCS #1 attack [15].

3.8 Licensing

The code produced within the project will be released with an Open Source license. Therefore, all software components, including externally produced components, must be licensed under an Open Source license. Open Source is defined by the Open Source Initiative as follows [16]:

1. **Free Redistribution**. The license shall not restrict any party from selling or giving away the software as a component of an aggregate software distribution containing programs from several different sources. The license shall not require a royalty or other fee for such sale.
2. **Source Code**. The program must include source code, and must allow distribution in source code as well as compiled form. Where some form of a product is not distributed with source code, there must be a well-publicised means of obtaining the source code for no more than a reasonable reproduction cost preferably, downloading via the Internet without charge. The source code must be the preferred form in which a programmer would modify the program. Deliberately obfuscated source code is not allowed. Intermediate forms such as the output of a pre-processor or translator are not allowed.
3. **Derived Works**. The license must allow modifications and derived works, and must allow them to be distributed under the same terms as the license of the original software.
4. **Integrity of The Author's Source Code**. The license may restrict source-code from being distributed in modified form only if the license allows the distribution of "patch files" with the source code for the purpose of modifying the program at build time. The license must explicitly permit distribution of software built from modified source code. The license may require derived works to carry a different name or version number from the original software.
5. **No Discrimination Against Persons or Groups**. The license must not discriminate against any person or group of persons.
6. **No Discrimination Against Fields of Endeavor**. The license must not restrict anyone from making use of the program in a specific field of endeavor. For example, it may not restrict the program from being used in a business, or from being used for genetics research.

7. **Distribution of License**. The rights attached to the program must apply to all to whom the program is redistributed without the need for execution of an additional license by those parties.
8. **License Must Not Be Specific to a Product**. The rights attached to the program must not depend on the program being part of a particular software distribution. If the program is extracted from that distribution and used or distributed within the terms of the program's license, all parties to whom the program is redistributed should have the same rights as those that are granted in conjunction with the original software distribution.
9. **License Must Not Restrict Other Software**. The license must not place restrictions on other software that is distributed along with the licensed software. For example, the license must not insist that all other programs distributed on the same medium must be open-source software.
10. **License Must Be Technology-Neutral**. No provision of the license may be predicated on any individual technology or style of interface.

Here are some licenses that comply with the definition.

- Apache License 2.0
- BSD 3-Clause "New" or "Revised" license
- BSD 2-Clause "Simplified" or "FreeBSD" license
- GNU General Public License (GPL)
- GNU Library or "Lesser" General Public License (LGPL)
- MIT license
- Mozilla Public License 2.0
- Common Development and Distribution License
- Eclipse Public License

4 References

[1] Doxygen, “Doxygen: Generate documentation from source code,” Doxygen, [Online]. Available: http://doxygen.org/. [Accessed 2016].

[2] NSA, “NSA Suite B Cryptography,” National Security Agency, August 2015. [Online]. Available: https://www.nsa.gov/ia/programs/suiteb_cryptography/. [Accessed April 2016].

[3] NIST, “NIST Special Publication 800-56A: Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography,” [Online]. Available: http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Ar2.pdf.

[4] FIPS, “FIPS PUB 186-4: Digital Signature Standard (DSS),” [Online]. Available: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf.

[5] T. Kivinen and M. Kojo, “RFC 3526: More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE),” May 2003. [Online]. Available: https://www.ietf.org/rfc/rfc3526.txt.

[6] RSA Laboratories, “PKCS #1 v2.2: RSA Cryptography Standard,” October 2012. [Online]. Available: http://www.emc.com/emc-plus/rsa-labs/pkcs/files/h11300-wp-pkcs-1v2-2-rsa-cryptography-standard.pdf.

[7] ISO/IEC/ANSI, Programming languages — C. International Standard 9899:1999, International Standardization Organization, 1999.

[8] CERT Software Engineering Institute, “SEI CERT C Coding Standard,” 2016. [Online]. Available: https://www.securecoding.cert.org/confluence/display/c/SEI+CERT+C+Coding+Standard.

[9] Google BoringSSL Team, “BoringSSL Style Guide,” Google BoringSSL Team, 2016. [Online]. Available: https://boringssl.googlesource.com/boringssl/+/HEAD/STYLE.md.

[10] CCS Group, “Cryptograpic Coding Standard,” CCS Group, 2016. [Online]. Available: https://cryptocoding.net/index.php/Cryptography_Coding_Standard.

[11] GNU Project, “Introducing the GNU Build System,” GNU Project, [Online]. Available: https://www.gnu.org/software/automake/manual/html_node/GNU-Build-System.html#GNU-Build-System. [Accessed 2016].

[12] H. Cohen and K. Belabas, “PARI/GP home,” 2016. [Online]. Available: http://pari.math.u-bordeaux.fr/.

[13] “Valgrind Home Page,” Valgrind Developers, 2016. [Online]. Available: http://valgrind.org/.

[14] “SAW (Software Analysis Workbench),” Galois Inc., 2016. [Online]. Available: https://galois.com/project/software-analysis-workbench/.

[15] D. Bleichenbacher, “Chosen Ciphertext Attacks Against Protocols based on the RSA Encryption Standard PKCS #1,” in *CRYPTO '98, LNCS 1462, Springer, pp. 1-12.*, 1998.

[16] Open Source Initiative, “The Open Source Definition,” 2016. [Online]. Available: https://opensource.org/osd.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644729

End of Document