Verificatum

Programming Philosophy behind the Verificatum Mix-Net

VMN Version 3.1.0
2022-09-10

The Verificatum mix-net is an implementation of a provably secure El Gamal-based mix-net. This document describes the programming philosophy. For additional information about the mix-net, please visit http://www.verificatum.org.

Contents

1 Introduction

In this document we describe the *philosophy* behind the Verificatum mix-net implementation and how this leads to certain decisions about programming languages, code standards, testing etc, and not the architecture of the code or any technical details of the code itself.

2 Choice of Language

Verificatum AB has experience in programming and teaching functional, imperative, and object-oriented programming languages, so several languages were considered before an informed decision on the programming languages was made and the implementation started.

A basic guiding principle when writing security critical code is that other people should be able to read and audit the code. For the choice of programming languages, this means that we can not choose a programming language that is too obscure. Having many potential readers of the code is a security feature that must be carefully weighed against the possibility of various forms of automated analysis and optimizations of the code.

Script languages such as Python were discarded early due to the lack of strong typing mechanisms and fast moving evolution of both the languages and associated libraries. Trading stability for fast updates and responsiveness to new suggestions is reasonable for some applications, but not for mix-nets that are security critical and expected to converge to a state that is nearly static. Python is for example not even backwards compatible between major versions.

Pure imperative languages such as C are perfect for writing fast, but structurally simple code. The running time of mix-nets implemented in such languages can be reduced substantially using special optimizations that make no sense in general purpose libraries, or even in cryptographic packages such as OpenSSL [22]. To implement such optimizations on top of an existing library without loosing too much due to overheads is virtually impossible without using the language used to implement the original library. Fast arithmetic code is commonly implemented in C, so this is a natural language for low level arithmetic code.

On the other hand imperative languages lack the object-oriented features we expect from a modern programming language. These features are vital to modularize and manage the complexity of larger software packages. It may seem natural to use C++ on top of the fast low-level routines written in C, but C++ is well known to be a difficult language to master. There are several features that if incorrectly used may cause the exception handling system to fail, or leak memory. Another natural alternative is C#, but we considered this to be too tightly coupled with Microsoft Corporation. Java seems to be the most natural choice for writing the logical code and the user interface of the mix-net, since it runs on a variety of platforms and is understood by many programmers. An additional advantage of Java is that Verificatum AB know the developers of the Oracle Java virtual machine. This allows asking deep questions about its behavior.

Historically, the execution speed of applications written in Java was significantly lower than those written in languages without a garbage collector that are not interpreted, but this issue has been resolved for many settings. Furthermore, the majority of the work in a mix-net is isolated in a few subroutines, which may be implemented in a low level language. Thus, any claims that Java is too slow to be used to implement a mix-net are wrong.

Languages such as Haskell, Ada, or Scala have various advantages that improve the correctness of code, but few programmers master these programming languages and even fewer would understand the annotations needed to use the advanced tools for code analysis available. Furthermore, correctness of executions are guaranteed using zero-knowledge proofs anyway, so at best these tools can help verify that randomness is used correctly during execution to preserve privacy. Thus, we do not consider using them to be a sensible choice in practice.

To summarize, the two languages chosen to implement VMN are: (1) Java for logic and user interface, including processing of configuration files, and inputs and outputs, and (2) C for fast arithmetic code in a few carefully chosen places. Using Java Native Interface (JNI) the two can be combined.

3 External Dependencies

Early on in the project it was decided to keep external dependencies to a bare minimum. Any piece of software already depends on the operating system and various standard libraries as well as on the firmware and hardware on which the operating system runs. Every additional external dependency is an additional potential security risk, both due to honest mistakes by developers of the library and due to malicious programming. Thus, each external dependency should be carefully considered and the list of all dependencies should be made explicit. Given our choices of C and Java as programming languages we can not avoid that VMN depends on the following components.

- The GNU build system (Autotools) consisting of Autoconf [4], Automake [5], and GNU Libtool [14], including the GNU M4 macro interpreter [15], used to configure, build, make, and install (relocate) software written both in C and Java.
- The GNU Compiler Collection (GCC) [13] and the GNU C Library (glibc) [12] used to compile C code for low level arithmetic.
- The Java compiler, the Java Class library, and the Java VM, used to compile, package, and execute the Java protocol code.
- The shell (`/bin/sh`) used to provide wrappers of compiled Java programs.
- The Unix-like operating system used to execute the mix-net. This handles memory allocation, process book keeping, access to file system, network access, and so on.
- The BIOS, hardware, and possibly re-programmable firmware and microcode of the hardware.

VMN is confirmed to run on FreeBSD, Ubuntu, and CentOS, but it should run on any system that provides recent versions of the above. In many Unix-like operating systems the default system shells (`/bin/sh`) are derivatives of the (almost) POSIX compliant Almquist's shell (a derivative of the Bourne shell), e.g., FreeBSD and Debian (and its derivatives such as Ubuntu). Shells, and more generally script languages, are a common source of security problems. System shells are generally considered to be more secure than more expressive shells such as Bash or C shell. Thus, we deliberately use the system shell in our wrappers, despite that this makes our scripts slightly more complex.

The above dependencies already give rise to security risks, but there is no such thing as a completely secure computer system. We accept the following two additional dependencies to gain substantial efficiency improvements, but we leave a pure Java implementation as an option.

- The GNU Multiple Precision Arithmetic Library (GMP) [17] for implementing modular arithmetic needed to implement arithmetic in a subgroup of the multiplicative group of a field, and for implementing elliptic curves over prime order fields.
- The handcrafted elliptic curve code for the standard NIST curves P-224, P-256, and P-521 found in the OpenSSL Cryptography and SSL/TLS Toolkit (OpenSSL) [22].

GMP is the fastest general purpose arbitrary precision integer arithmetic library in the world and it has an advanced testing framework with exceptional coverage. The code quality is also exceptional. Furthermore, the original author Torbjörn Granlund is a has helped us write our addon GMP Modular Exponentiation Extension library. This gives us direct access to deep knowledge about the library and strong trust.

OpenSSL is infamous for its source code, despite that it has been significantly improved during the last decade, and it has been plagued with bugs over the years. Thus, for security critical applications it is prudent to avoid it as an important component.

However, parts of the code have high quality and this is the case for the handcrafted implementation of elliptic curves. In fact, these implementations are exceptionally clean and well commented. They are written by employees of Google, namely Emilia Käsper and Adam Langley, but they are based on an implementation of Dan Bernstein, who is a world leading expert on elliptic curves and a very skilled programmer. Dan has kindly provided valuable advice regarding elliptic curves. Emilia has swiftly answered all of our questions about her code. Because of all of the above we feel comfortable to include this code in our source.

We do not link against OpenSSL as a library for three reasons:

1. The library embeds the implementations of the curves statically[1], so it is presently technically impossible.

2. We do not want our software to depend on a full installation of the large and complex OpenSSL package to let us use a few routines. Too many people can commit code to the OpenSSL code base for this to feel comfortable.

3. Embedding highly optimized code in optimizing special-purpose algorithms without loosing notably in efficiency due to data conversion overhead costs is non-trivial. We were unable to preserve the advantages of the curve implementations in our code without simply copy it into our code and use macros to embed it into our routines side-by-side with our own implementation.

Projects implemented in Java normally uses a Java-oriented building system such as Maven [3] or ANT [2]. However, we think it is more natural to use GNU Make [16], since we need the GNU Autotools anyway and GNU Make is the traditional way to manage, build, and install software on Unix-like operating systems. Furthermore, far more programmers are comfortable with GNU Make than any of the building tools available for Java, and using different building tools for the two languages would add an additional unnecessary dependency.

4 Coding Standard

No explicit coding standard of Verificatum AB is defined in a separate document, but the code mostly follows the original Java coding standards [7] (now no longer maintained) and Google's coding standard [18]. These two standards are similar. Whenever we use a different approach, we use it consistently.

The main difference between our coding standard and other high quality coding standards is that we enforce documentation at more levels, e.g., we require that *every* class, method, variable, and constant is documented with a Javadoc comment. Furthermore, we use comments within the code to explain anything that is not obvious.

[1]This software engineering approach is quite unusual, perhaps it could even be described as wrong, but there may be a good reason for it that escapes us.

5 Code Analysis

We take code quality very seriously and we have an uncompromising development approach. Thus, we do not hesitate to re-write large parts of the software for the sole purpose of making the code simpler and better organized. We have done this several times in the past, and as a result the code has converged into a form that is perfect for our purposes.

An important goal for Verificatum AB is that the code for VMN is easy to read and audit by humans, but an equally important goal is that the software is easy to maintain and extend.

We also want to be able to use automated analyzers of the code. There are several such tools that are more or less stringent and complete. We do not aim at using such tools at the expense of human readability, i.e., we think that: readability by humans is the first priority, that annotating the whole code to allow full and strict automated analysis is unrealistic, and that this would reduce the readability of the code.

Optimized code tends to be more complex and harder to read than non-optimized code. Often this can be alleviated by finding the right abstractions that allow hiding most of the optimizations deep down in the arithmetic routines.

Currently we regularly analyze our code with the following static analyzers.

- CheckStyle [6] is used to verify that the code adheres to our code standard.
- FindBugs [10] uses static analysis to identify bugs in the code.
- PMD [23] uses static analysis to identify bugs and badly organized code, e.g., blocks of code with high cyclomatic index.

We also actively invite, pay, and cooperate with consultants and researchers to analyze the code using all tools they have access to. Each issue raised in an analysis is carefully considered, and unless there is a valid reason to ignore it, the issue is resolved. The following people have analyzed the code.

- Mats Henricson at Crisp AB [8] was hired by Verificatum AB to write an independent code quality report [21].
- Prof. Joe Kiniry, `http://galois.com/team/joe-kiniry`, from Galois Incorporated [11] is an expert on code verification and code quality. He volunteered to analyze the software and concluded that its quality is exceptionally good. His statement is given in Appendix A.
- As part of a PhD course in code scanning given by Prof. Carsten Schürmann, PhD students Iago Abal Rivas and Peter Brottveit Bock at IT University of Copenhagen used several tools and manual inspection to write a report [24].

We stress that all the issues raised in the above reports that were not deemed irrelevant have been addressed. Some of the claims in the analyses have already been discussed in previous sections.

6 Testing

Testing is important to verify the functionality of a software package, and regression testing is key to avoid introducing new bugs that goes undetected. Unit testing is used to test as small components as possible to avoid an exponential blowup.

The most commonly used unit testing framework for Java is JUnit [20], but there are other options such as TestNG [25]. We considered these options, but we decided that it is more important to keep our software self-contained and avoid additional dependencies than having access to the features of, e.g., JUnit. Indeed, the functionality we need can readily be provided with a few hundred lines of code. Another advantage of our approach is that our testing code does not require the reader to know JUnit. This is particularly important for programmers that rarely use Java, but are able to audit Java code reasonably well.

7 Data Formats

There are two types of data formats: (1) those that are read both by computers and humans, and (2) those that are only read by computers. In the second case a format may be used both to read inputs and write outputs, and to store intermediate data during execution.

An obvious requirement is that the formats are language independent. This disqualifies Java Serialization and similar independently implemented packages.

There are a few natural formats that can be used in the former case. We use XML [9] without attributes, i.e., all values are stored between a begin tag and an end tag. We make sure that comments for all fields are printed automatically when data are written to file. It is very easy to implement a recursive descent parser from scratch for this format.

There are several possible choices for the second case as well, e.g., JSON [19] or one of the formats of ASN.1 [1]. The advantage of using a standard format is of course that it is standard and that there are libraries that can parse and write data using the format, but there are also several drawbacks.

A natural approach for an adversary is to attempt to modify either the definition of the format or the implementation for reading and writing data given in the format. This could change the behavior of the mix-net or independent verifiers, possibly circumventing the verification of the universally verifiable proof.

We can not expect that people who must implement parsers already master any standard format. Thus, if we can define a simpler special purpose format for which this is straightforward, then this simplifies implementing and maintaining both the mix-net and independent verifiers of the universally verifiable proof.

Internally the mix-net must store intermediate data on file, since some data requires too much space to be stored in memory when there are many ciphertexts. To avoid overhead problems when processing such data the format must be binary and reading and writing must be very fast. It must also allow simulating random access by seeking in a file, which is difficult in many standard formats. Employing multiple formats introduces additional complexity and should be avoided. Binary formats makes it harder to debug code that use it, but this problem is solved by implementing a tool that prints data in a properly formatted user readable form.

Recall that the Fiat-Shamir heuristic allows a mix-server to construct a non-interactive zero-knowledge proof that may later verified by an independent verifier. The Fiat-Shamir heuristic requires both parties to hash data stored on file. For this to work all parties in an application must use *exactly* the same format. It is not uncommon that external libraries claimed to meet a standard fails to do so or behaves differently for things that are not explicitly defined in the standard. In our setting the consequences of such a discrepancy could be particularly serious, since it may influence which mix-servers are considered to be corrupted.

Another aspect to consider is that whereas externally generated data must be treated as malformed, internally generated data is expected to be correctly formed. In the latter case we may drop certain consistency tests that have an impact of the running time and simply keep some basic sanity checks. Thus, having a two tiered solution where the basic format stores structured binary data and the complete format assumes such a structure makes sense.

To summarize, the best approach is to define a single special purpose two-tiered format that:

is easy to describe, is near trivial to parse, allows fast parsers, writers, and seeking. We let our arithmetic objects be stored as the integers from which they are formed, and integers are simply stored in two's complement format. Then such data is stored as an ordered tree of byte arrays. A piece of data in our format is called a *byte tree*. This format is described in [26, 27].

8 User Interface

To avoid accidental use with wrong parameters, we expect users to provide their own wrappers of the functionality provided by VMN with some fixed parameters and a subset of parameters. Thus, it is natural to provide a command-line interface to VMN. This interface is described in [27]. To help the user, each command has conservative defaults for most values and a help flag that prints detailed usage information.

References

[1] Abstract syntax notation one (ASN.1). `http://www.itu.int/en/ITU-T/asn1`, October 2014.

[2] Apache ANT project. `http://maven.apache.org`, October 2014.

[3] Apache Maven project. `http://maven.apache.org`, October 2014.

[4] Autoconf. `http://www.gnu.org/software/autoconf`, October 2014.

[5] Automake. `http://www.gnu.org/software/automake`, October 2014.

[6] Checkstyle. `http://checkstyle.sourceforge.net`, October 2014.

[7] Code conventions for the Java programming language. `http://web.archive.org/web/20090911103851/http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html`, October 2014.

[8] Crisp AB. `http://crisp.se`, October 2014.

[9] Extensible markup language (XML). `http://www.w3.org/XML`, October 2014.

[10] FindBugs. `http://findbugs.sourceforge.net`, October 2014.

[11] Galois Incorporated. `http://galois.com`, October 2014.

[12] GNU C library. `http://www.gnu.org/software/glibc`, October 2014.

[13] GNU compiler collection (GCC). `http://www.gnu.org/software/gcc`, October 2014.

[14] GNU libtool. `http://www.gnu.org/software/libtool`, October 2014.

[15] GNU M4. `http://www.gnu.org/software/m4`, October 2014.

[16] GNU make. `http://www.gnu.org/software/make`, October 2014.

[17] Gnu multiple precision arithmetic library (GMP). `http://www.gmplib.org`, October 2014.

[18] Google coding standard. http://google-styleguide.googlecode.com/svn/trunk/javaguide.html, October 2014.

[19] Javascript object notation (JSON). http://www.json.org, October 2014.

[20] JUnit. http://junit.org, October 2014.

[21] Mats Henricson. Results from reviewing the Verificatum code base. http://www.verificatum.com/documents/crisp_henricson_analysis.pdf, October 2014.

[22] OpenSSL cryptography and SSL/TLS toolkit. http://www.openssl.org, October 2014.

[23] PMD. http://pmd.sourceforge.net, October 2014.

[24] I. A. Rivas and P. B. Bock. Analysis of the Verificatum mix-net project report – PhD course in code scanning. http://www.verificatum.com/document/demtek_rivas_brottveit_analysis.pdf, October 2014.

[25] TestNG. http://testng.org, October 2014.

[26] D. Wikström. How to implement a stand-alone verifier for the Verificatum mix-net. http://www.verificatum.org, October 2011.

[27] D. Wikström. User manual for the Verificatum mix-net. http://www.verificatum.org, October 2012.

A Comments About the Source by Other Researchers

“Verificatum library is superior in quality – both in the big-picture design and the implementation details – than any other crypto library I have ever audited. Typical crypto code has no architecture, little design, few docs, and a handful of tests. Doug’s library has a clean architecture and design that mirrors one’s expectations as an applied cryptographer and a software engineer, has excellent documentation coverage, decent tests, and excellent performance. He has shown that it is not just possible, but it is wise, to build quality crypto via quality engineering principles.

With regards to how many libraries I have audited/looked at, our current hacrypto GitHub repo contains about 35 libraries. The only stuff that comes close to yours is NaCl and PolarSSL.”

— Prof. Joseph Kiniry, October 28, 2014 (quote reproduced with the permission)