Optimized native implementation of Lyra2 password hashing scheme in Java

BACHELOR THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software Engineering and Internet Computing

by

Aleksandr Lisianoi
Registration Number 01527346

elaborated at the
Institute of Computer Aided Automation
Research Group for Industrial Software
to the Faculty of Informatics
at TU Wien

Advisor: Clemens Hlauschek

Vienna, September 21, 2017

Abstract

This work describes the process of porting a password hashing scheme called Lyra2 from C99 to Java 1.8. A native Java implementation should ease integration with other Java projects and simplify its usage on other platforms, such as Android phones.

This paper follows a straightforward approach. First, a brief description of the basic Lyra2 configuration is given, followed by an overview of the reference implementation. Then follows a description of the ported Java project along with a number of notable challenges.

Section 5 presents the results. It opens with both manual and automated testing which shows that both projects generate the same hash values when given the same inputs. A performance comparison follows in section 5.2, showing several thousands of measurements. Finally, section 5.3 demonstrates a small proof-of-concept mobile application that uses Lyra2.

In summary, the ported Java project could be found on GitHub [3] and on Maven Central [4]. The comparison project and the source code for the mobile application are hosted on GitHub as well, projects [1] and [7] respectively. Generally, the ported implementation requires more processing time and memory. The main reasons for that could be found in comparison conclusion 5.2.5. However, it integrates easily into the Android ecosystem which may justify the performance drop.

Keywords

Password Hashing Scheme, Lyra2, Memory Hard Functions, Java, Android

Contents

List of Figures

List of Tables

1 Introduction

Passwords are currently the backbone of user authentication. Usually, passwords are stored in a hashed form in some kind of a database. Such databases are fairly often compromised and then the hashing mechanism is what stands between the attacker and the password cleartext.

1.1 Problem Description

Processing power increases with time while simultaneously getting cheaper. This works both for the legitimate users as well as the attackers. Password Hashing Schemes (PHSs) are therefore continuously adjusted to stay irreversible. However, recent advances in highly parallelized hardware (conventional multicore GPUs as well as the more specialized FPGAs and ASICs) present a new challenge for the commonly used cryptographic hash functions. An attacker can heavily parallelize the computation, trying several thousands of password and salt combination in the time it takes a legitimate user to compute just one.

In order to limit the throughtput a potential attacker could achieve, the Password Hashing Competition was announced in 2013 and concluded in 2015 [52]. Its evaluation criteria stress that proposed candidates should provide minimal speed-up for the highly parallelized hardware. The winner was declared to be Argon2 [18] and special recognition was also given to Catena [29], Lyra2 [10, 31], Makwa [40] and yescrypt [37].

1.2 Motivation

Even though the theoretical designs and their proof-of-concept implementations were presented back in 2015, the adoption of these new cryptographic algorithms could be better. There are many ways to do so: provide better documentation, detailed usage examples and success stories. Porting an existing implementation into another programming language can also result in an adoption boost.

1.3 Contribution

This work describes the porting process of the Lyra2 reference implementation into Java. The resulting implementation is hosted in the Maven Central repository [4] as well as on GitHub [3]. This makes it available for seamless inclusion as a dependency into any Java project. It is licensed under the MIT license which is a well-known permissive software license. Finally, the source code is publicly available and can be inspected or improved if necessary.

The primary goal of this porting effort is to provide a drop-in replacement for the reference implementation. Given the same input parameters, both implementations should produce the same hash values. Although this might sound like an automatic requirement, it is not in fact the case. The paper will highlight the challenges in details about the Java implementation section 3.3.

The secondary goal is to compare the ported implementation to the original. The comparison project is done in the spirit of reproducible research, is hosted publicly on GitHub [1] and can be used to verify the results presented in this paper.

The final goal is to use the ported implementation to write an Android application. Section 5.3 demonstrates the ease of deployment of Lyra2 on a platform where using the reference implementation is not as straightforward.

1.4 Outline of the Work

Chapter 2 is devoted to an overview of related work. It opens with the history of cryptographic competitions in section **??**. A description of password hashing fundamentals follows in section 2.2. Finally, sections 2.4, 2.3, 2.5 and 2.6 provide on overview of Password Hashing Competition finalists Makwa, Catena, yescrypt and Argon2.

Chapter 3 covers Lyra2. It begins with the necessary theoretical information about the sponge and the duplex constructions in 3.1. This section follows up with the different phases that Lyra2 goes through when performin a hash computation. Section 3.2 outlines the build system and the various possible configurations of Lyra2. The section concludes with an extension of the build system which allows to compile several configurations at once as well as compute a bulk of hash values. The ported Lyra2 project is discussed next in 3.3. It first deals with the transition from a function-based to an object-oriented project. The most important classes are shown on an UML diagram 3.3. Several classes which circumvent specific porting challenges are described. In particular, the endianness issue is shown in figure 3.4 and a rotation trick for the Blake2b-based sponges is shown in figure 3.5.

Chapter 4 discusses the choice of a unit testing framework. First, the practice of unit testing is described as a viable program verification approach. Section 4.1 follows up with the basic terms and definitions from the unit testing practice. Then section 4.2 motivates the choice of unit testing frameworks for different projects: the reference implementation, the Java port as well as the Python scripts that accompany it. In particular, 4.2.1 describes the effort it would take to set up `Boost.Test` for the reference projects Lyra2 and Argon2, showing possible reasons why those projects use their own solutions instead. Section 4.2.2 motivates the choice of `JUnit` as a unit testing framework for lyra2-java. That section highlights the usefulness of parametrized testing. Finally, section 4.2.3 provides reasons why the `harness` Python scripts rely on the `py.test` framework to tun parametrized tests as well.

Chpater 5 summarizes the results. It begins with the demonstration that both projects produce the same hash values given the same inputs 5.1. First, a manual testing session is recorded in 5.1.2. A section on continuous integration follows in 5.1.3. Then the performance comparison is presented in section 5.2. Due to a large number of configurable parameters in Lyra2, the first comparisons fix time- and memory costs (respectively, subsections 5.2.2 and 5.2.3). Section 5.2.4 then deals with both changing time- and memory costs, resulting in several thousands of measurements displayed. A comparison conclusion is made in 5.2.5. Section 5.3 presents the Android mobile application. It opens with the discussion of required Android API versions and tools. Then it shows several screenshots of the application which runs Lyra2.

Chapter 6 is the conclusion. It densely summarizes the results and suggests ideas for future research and development.

2 Related Work

2.1 Cryptographic Competitions

It is common practice to announce a competition in order to develop standard cryptographic primitives. For example, the symmetric block cipher Rijndael [23] was chosen to become the Advanced Encryption Standard (AES) [46] in a competitive selection process that lasted from 1997 to 2000. The competition was organized by the National Institute of Standards and Technology (NIST) and included 15 different designs which were narrowed down to 5 during the final phase: Rijndael [23], Serpent [9], Twofish [45], RC6 [42] and MARS [26]. Such an open standartization process was highly praised by the cryptographic community. So, another competition was held by NIST between 2007 and 2012 in order to select the next hash function standard, SHA-3. The final round included 5 designs: Blake [14], Grøstl [30], JH [58], Skein [28] and Keccak [16], the last of which ultimately became the winner of the competition.

Choosing cryptographic primitives through an open competitive process is an effective approach. Therefore, when the need for an updated, memory-hard password hashing scheme became apparent, a Password Hashing Competition was held. This time, however, it was not organized by NIST but directly by the cryptographic community. In particular, the password-hashing.net website lists two well-known cryptographers Dmitry Khovratovich and Jean-Philippe Aumasson as direct contacts. The Password Hashing Competition concluded in 2015, with Argon2 selected as its winner and Catena, Lyra2, Makwa and yescrypt receiving special recognition.

2.2 Password Hashing Fundamentals

Password hashing is the process of transforming a password into a hash value. Given the resulting hash value it should be computationally infeasible to restore the original password.

Password hashing is common practice when user authentication is required. The user provides a password which is then hashed using some password hashing algorithm. The resulting hash is then compared against a hash value which is stored in the database and is known to be correct. If both hashes match, the user is authenticated.

When a password database is leaked, the password hashing process is what prevents the attackers from gaining the original plaintext passwords. One of the early password hashing schemes, which is widely used today, is PBKDF2, described in section 5.2 of [32]. The fundamental idea behind it (which is shared among a few other early password hashing schemes) is to apply a *pseudo-random function* a number of times, treating it as a parameter for computational cost.

One of the early kinds of attacks against password hashes are *rainbow tables*. They are a classic example of a *Time/Memory Tradeoff Attack* (TMTO). Given a dictionary of possible passwords (up to a certain length and using a particular set of symbols) and a hashing algorithm an attacker precomputes hash values well in advance. When a password database of the defender is leaked, the hash values are compared to those from the rainbow table and in case of a match the password can be recovered because the computation has already been performed.

The currently well-known practice to combat the rainbow table attack is to use a sufficiently large *salt* when computing the password. This salt is a randomly generated value which is unique for

every password and is openly stored alongside it in the defender's database. This ensures that the attacker will have to perform the computation after the leak (possibly giving a chance to the defender to change their password). Another nice property is that the same password used by several users will be very likely to produce distinct hash values.

These days an attacker can often compute a large number of hashes in parallel. If so, the increased individual computational time cost of one hash does not prevent the attacker from trying many combinations at once. The throughtput of the attack (average attempted passwords per second) remains high. This is the type of attacks addressed by memory-hard password hashign schemes. The main idea is that parallel systems (such as general-purpose GPUs or specialized ASICs and FPGAs) usually have significantly less memory per one processing unit. Therefore, if a password hashing process offers a memory cost parameter, an attacker will face greater costs per one parallel computation.

Arguably the first password hashing scheme to implement this idea this was scrypt, which was recently published as RFC 7914 [36]. However, this scheme is sometimes criticized for being overly complicated and not offering a decoupled way to control time and memory costs. In other words, there is a single parameter that controls both those values.

In order to create new designs which would address this problem, the Password Hashing Competition was announced in 2013 and concluded in 2015 [52]. Below follows a quick summary of its winner (Argon2) and all of the finalists except Lyra2. The latter is the main focus of this work and will be described in more detail later in 3.

2.3 Main Features of CATENA

CATENA is a password-scrambling framework based on Bit Reversal Graphs [29]. It packs such features as *client-independent updates* which allows a hash value to be updated using larger time or memory cost parameters without the need to wait for a user to login. CATENA also offers a *server-relief* feature which allows to offload most of the hash computation to the client machine rather than the server, hence increasing the number of possible concurrent logins.

The CATENA-BUTTERFLY(-FULL) and CATENA-DRAGONFLY(-FULL) are the most notable configurations of CATENA. The former one is recommended when the memory-hard property is required. The -FULL versions utilize the complete set of rounds of the underlying hash function Blake2b.

An interesting related project is CATENA-AXUNGIA which can be found in [49]. It allows its user to specify the desired time and memory requirements in conventional (for a human) seconds and kilobytes. After that the program returns recommended values which will on average make CATENA-BUTTERFLY or CATENA-DRAGONFLY run for the set amount of time and consume the set amount of memory.

Finally, the CATENA-VARIANTS [48] project is a modular C++ implementation of CATENA. It is reported to be slower and consume more memory while at the same time providing more flexibility. There is an API that allows the developer to mix and match internal parts of the algorithm.

2.4 Main Features of Makwa

The Makwa PHS at its core relies on Blum integers [40]. A Blum integer is a natural integer n which can be represented as a pq where p and q are prime numbers with an additional property:

$$p = 3 \mod 4 \tag{2.1}$$

$$q = 3 \mod 4 \tag{2.2}$$

The squaring is primarily computation intensive and does not require a significant amount of memory. The p and q integers should be kept secret, if they are known then the computation can be accelerated considerably.

One of the distinguising features of Makwa highlighted on its website is *delegation* [39]. The author points out that the complexity of password hashing is essentially an arms race between defenders and attackers. Delegation allows to use untrusted systems to perform part of the computation of the hash value with the Makwa algorithm. The exact protocol can be found in Section 4 of the specification [41].

2.5 Main Features of yescrypt

The *yescrypt* PHS [37] improves upon its predecessor, *scrypt* [36]. However, yescrypt author Alexander Peslyak makes it clear that the author of scrypt is a different person, Colin Percival. The yescrypt PHS deals with some minor inconsistencies discovered in the specification of its predecessor.

The yescrypt PHS also introduces a novel configuration with a read-only memory (ROM) table. In that configuration random lookups are performed so as to ensure that this table remains in memory. Finally, the `YESCRYPT_RW` flag enables these lookups as well as a number of optimized instructions.

2.6 Main Features of Argon2

Argon2 has two distinct configurations: *Argon2i* and *Argon2d* [18]. The former revists the blocks of the in-memory matrix in the data-*independent* fashion while the latter does so in a data-*dependent* manner. This means that Argon2i is better suited for scenarios when *side-channel attacks* are a viable concern, such as during password hashing or key derivation. At the same time Argon2d is more resistant to *time-memory tradeoffs* which makes it more suitable for digital cryptocurrencies or other cases where proof of work is important.

Argon2 accepts the following set of parameters: password, salt, degree of paralelism, length of the produced hash (called a *tag* in [18]), memory cost, time cost, version number (for compatibility reasons, currently at `0x13`), secret value, associated data and type of configuration to use (Argon2i or Argon2d).

The more interesting parameters are the degree of parallelism as well as secret value together with associated data. The last of these three adds more flexibility to the scheme. The secret value parameter enables *keyed hashing* and improves security in case of a database leak. Keyed hashing is similar to salt but the key is the same for the entire database and is stored in Random Access Memory. This does not introduce theoretical security but instead complicates the technical job for an actual attacker. Finally, the degree of parallelism directly corresponds to the number of rows of the in-memory matrix.

Argon2 is arguably the most widespread and popular alogorithm among all of the finalists. Consequently, the `README.md` file in the GitHub repository [33] provides a long list of bindings for various languages. Finally, notable companies like the Django Software Foundation and Yandex are actively using Argon2 in production: [12, 13].

3 Lyra2: a sponge-based PHS

3.1 Theoretical Background

At the heart of Lyra2 is a *sponge function* or a *sponge construction* which is a class of algorithms with finite internal state that take an input bit stream of any length and produce an output bit stream of any length as well [55]. The internal state S consists of $w = b + c$ bits, where w is known as the *width* of the sponge, b is called its *bitrate* and c its *capacity*. At the heart of every sponge function lies a *fixed-width permutation* f which takes w bits as input and produces w bits as output. A common example of f is Blake2b [14], which in its 64-bit modification operates on 128-byte chunks.

As shown in figure 3.1, a sponge function consists of two stages: the absorbing and the squeezing stage. To start with, an incoming message is padded so that its length is a multiple of b bits. Secondly, the internal state of the sponge is initialized with zeros. Then the absorbing phase begins: the next b bits of the message are XORed with the first b bits of the state S and then the permutation f is applied to the whole S. This process repeats until the entire message has been absorbed. At this point the squeezing phase starts: the first b bits of S are emitted and then f is applied to the whole S. This keeps going until the desired output length is reached.

A similar cryptographic primitive is the *duplex construction*. The main difference with the sponge construction is that both the absorbtion of incoming bits and the squeezing of the outgoing bits happens simultaneously, in a tick-tock fashion. The name is based on an analogy between the half- and full-duplex modes of communication. A formal description can be found in [15] and a visual explanation is provided in figure 3.2. An important detail is that [15] proves that the sponge and the duplex construction are equally secure.

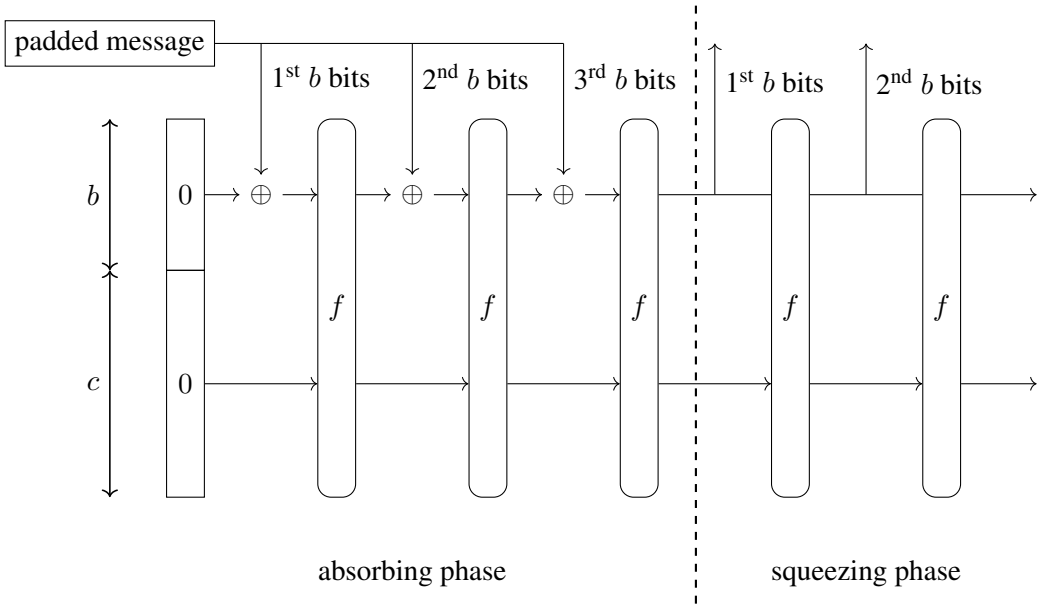


Figure 3.1: The sponge construction, adapted from [15, 31]

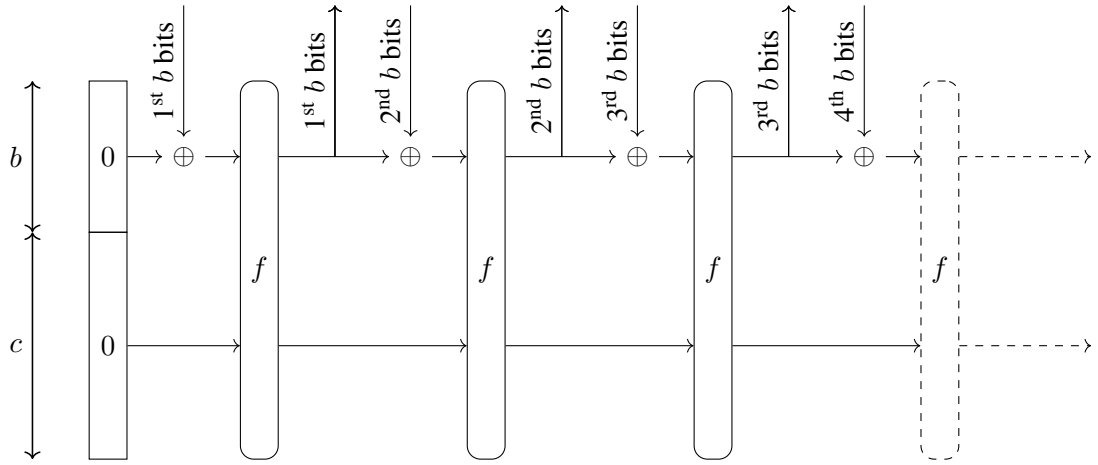


Figure 3.2: The duplex construction, adapted from [15, 31]

The next important component of Lyra2 design is the memory matrix. Its size is directly controlled by the memory cost parameter which corresponds to the number of rows of the matrix. An implementation detail are two other parameters: the number of columns and block length (switches `--blocks` and `--columns` in the Java executable of the ported project). Their product equals to the number of 64-bit words in a single row of the memory matrix. The reference C implementation sets both of these parameters at compile time while the Java implementation exposes them at runtime.

Conceptually, Lyra2 is a sequential application of a duplex construction which continuously reads and writes blocks to and from the memory matrix. The algorithm is structured into four main phases that take place one after the other: *Bootstrapping*, *Setup*, *Wandering*, and *Wrap-up*.

During the *Bootstrapping* phase the main event is the initialization of the sponge. It absorbs the password together with the salt and a few other parameters such as: lengths of output, password and salt, time cost, memory cost and number of columns. If required, this list could be expanded with application-specific information like user- or domain names. An implementation detail is that in case of Blake2b its IVs are used instead of zeroes.

The *Setup* phase deals primarily with initializing the memory matrix. The sponge here is used as a duplex construction: it continuously writes to the uninitialized rows of the matrix as well as updates some of the already visited ones. It takes special care so as not to let potential attackers discard parts of the memory matrix. It also speeds up the execution by using a reduced number of rounds of the permutation f. This phase ends once the last row of the matrix is visited.

The *Wandering* phase is the main operation phase of Lyra2. It runs for $time\_cost \times memory\_cost$ iterations and is by far the longest phase. The fact that both the $time\_cost$ and the $memory\_cost$ are decoupled allows legitimate users to fine-tune the parameter values to take full advantage of their platform. During the wandering phase, the rows of the memory matrix are continuously revisited und updated in a randomized fashion which is determined by the input parameters. As during the previous phase, special attention is paid to disallow time/memory tradeoffs that would either a) allow to parallelize computations or b) allow to discard parts of the memory matrix and recompute them later.

Finally, the *Wrap-up* phase concludes with a full-round absorbing operation by the sponge (which ensures that Lyra2 is at least as secure as the sponge) and a full-round squeeze operation that produces the requested number of output bits.

For a much more thorough introduction please consult the original paper [10, 31]

3.2 Reference Implementation

This section will present an overview of relevant parts of the reference implementation. It is hosted as a public GitHub repository [8]. There are several projects in the same repository (both Lyra2 and Lyra, as well as documentation). The Lyra2 directory of the master branch contains the latest version of the code and documentation. In particular, `Lyra2/src` is the root directory for code, `Lyra2/src/bench` contains benchmarking shell scripts, `Lyra2/src/cuda` contains code that looks how Lyra2 withstands GPU-based attacks, `Lyra2/src/sse` contains an sse-optimized implementation. Finally, the `Lyra2/src` contains the reference implementation in C99 and is the primary focus of this work.

The original public repository [8] was forked to [6] and the following modifications were made. The reference implementation uses `make` as its build system and the `Lyra2/src/makefile` provides clear compilation instructions. However, only one version of Lyra2 can be compiled at a time and the provided test vectors are hard-coded into the program. So, the following functionality was added for a more conventient comparison of the reference implementation to its ported Java version:

- Compile multiple versions of Lyra2 which could be used simultaneously.
- Perform sanity checks of the compiled executable with a set of quick tests.
- Compute and store hash values of some test inputs for each version of Lyra2.

This functionality can be found in the `harness` branch of the forked repository [6] and a pull request [5] to the reference repository. It is a Python 3 script `Lyra2/tests/harness.py` which can be configured both through the command line and the `Lyra2/tests/harness.yml` configuration file. The `Lyra2/tests/harness.py compile` compiles several Lyra2 reference implementations and the `Lyra2/tests/harness.py compute` runs those implementations on a series of test vectors and records the resulting hash values. Finally, if you install `py.test` then a call to `py.test` will run a few unit tests.

3.3 Java Implementation

It is fair to say that C is primarily a function-based language. At the same time Java is a lot more object-oriented. Therefore, the C functions from the reference implementation need to be translated into (abstract) classes and interfaces in Java. An additional challenge is the fact that the reference implementation uses conditional compilation, so a function with the same name actually contains different code depending on the instructions received by the compiler.

The architecture of the Java project underwent iterative improvements and a reasonably accurate UML diagram of the final version is presented in figure 3.3.

The `Main` class is the entry point of the project. An implementational detail that is not shown in figure 3.3 is that the `Main` class relies on a 3rd party command line library `picocli` [38]. This library parses input parameters (like the password, the salt, etc.) and constructs an instance of the `LyraParams` class. That class is also not shown in figure 3.3, it just stores the parameters and constants relevant to Lyra2's operation. Finally, the `echo` class is a collection of methods for pretty-printing different types of arrays as a table of bytes to the console. Lowercase name of a class is a convention that indicates that this class is essentially a collection of related static methods.

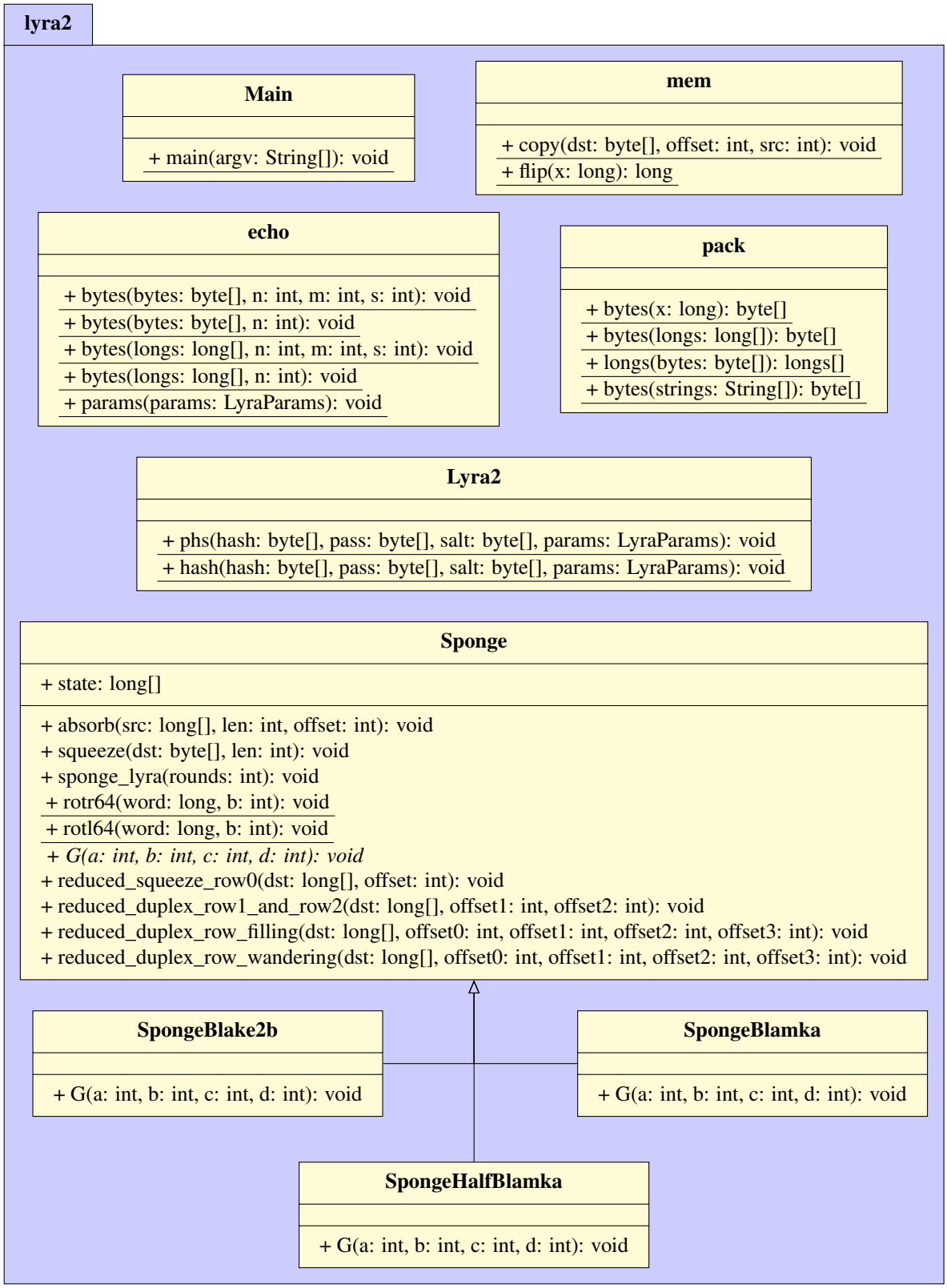


Figure 3.3: Approximate class diagram for `com.github.all3fox.lyra2` package

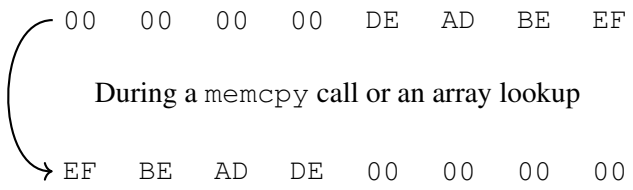


Figure 3.4: The byte swap on a little-endian platform (i.e. C99 on `x86_64`)

The `mem` and `pack` classes are responsible for memory manipulations. In particular, the `mem` class deals with the little- and big-endian discrepancies between the C and Java ecosystems on the `x86_64` architecture. The reference implementation is little-endian, which results in bytes being reversed when written to memory, as shown in figure 3.4. At the same time the Java virtual machine is big-endian. This is why the `long mem.flip(long x)` method is used constantly to reverse bytes back and forth.

The `pack` class provides methods that allow to emulate a particular use case when pointer casts are often used in `C`. It is often the case in `C` that a chunk of memory is allocated and treated as individual bytes or some other larger data type, like `uint64_t`. The `C` language allows to cast a `void*` pointer to `char*` or `uint64_t*`, depending on the situation. This also affects array indexing as the perceived size of the underlying element changes. In order words, depending on the type of the pointer, a `*(ptr1 + 5)` might be the beginning of the same element as `*(ptr2 + 1)`.

In order to circumvent this, the `pack` class provides a number of overloaded methods, where `pack.bytes(long[] xs)` takes `xs` and returns a copy of the same data in an array of type `byte` and `pack.longs(byte[] xs)` does the reverse for `long`. Of course, this introduces an expensive array copy. Luckily, this copy does not happen in the most expensive phases of *Setup* and *Wandering*.

The `Lyra2` class contains the main logic of the single-threaded Lyra2 instance. The functionality related to the sponge construction is captured by the `Sponge` *abstract* class. Its subclasses `SpongeBlake2b`, `SpongeBlamka` and `SpongeHalfBlamka` hold the specifics of the underlying fixed-width permutation f.

An insteresting detail is that the Sponge class in the Java project makes use of both left- and right bit rotations, done by the `rotl64` and `rotr64` methods respectively. These methods accept a 64 bit `long` and rotates it by the specified number of bits `b`. Those rotations are *not* arithmetic and do *not* preserve the sign of their argument.

The original Lyra2 reference implementation requires just one direction for rotations. Both types of rotations are helpful in the ported Java project because they allow to avoid a number of byte rotations related to endianness. For an example of this, refer to figure 3.5. It shows the `SpongeBlake2b` implementation, where lines $5, 8$, and 11 use a left rotation. The original Blake2b specification prescribes a right rotation for those steps. However, instead of calling `mem.flip` that would rotate its argument once and then actually rotate it with `Sponge.rotr64`, a single rotation in the other direction with `Sponge.rotl64` is performed.

The same trick cannot be performed on line 16 though. The rotation there is 63 bits to the right, which is not divisible by 8 bits (the size of one byte). Since `mem.flip` performs a byte-level rotation, it does not mix with the rotation and has to be performed explicitly. This rotation trick has accelerated the ported implementation by about 20%. The BlaMka family of sponges are based on Blake2b and benefit from the same trick as well.

```
1   public class SpongeBlake2b extends Sponge {
2       @Override
3       public void G(final int a, final int b, final int c, final int d) {
4           state[a] = mem.flip(mem.flip(state[a]) + mem.flip(state[b]));
5           state[d] = rotl64(state[d] ^ state[a], 32);
6
7           state[c] = mem.flip(mem.flip(state[c]) + mem.flip(state[d]));
8           state[b] = rotl64(state[b] ^ state[c], 24);
9
10          state[a] = mem.flip(mem.flip(state[a]) + mem.flip(state[b]));
11          state[d] = rotl64(state[d] ^ state[a], 16);
12
13          state[c] = mem.flip(mem.flip(state[c]) + mem.flip(state[d]));
14          // Cannot use the left rotation trick here: 63 % 8 != 0, so
15          // individual bytes do not stay the same, they change too.
16          state[b] = mem.flip(rotr64(mem.flip(state[b] ^ state[c]), 63));
17      }
18  }
```

Figure 3.5: Blake2b as instance of the Sponge class, demonstrating optimized rotations on lines 5, 8 and 11. Line 16 performs the complete set of rotations (no optimization).

```
1   public class SpongeBlamka extends Sponge {
2       private long fBlaMka(final long x, final long y) {
3           long lessX = 0x00000000FFFFFFFFL & x;
4           long lessY = 0x00000000FFFFFFFFL & y;
5
6           lessX *= lessY;
7
8           lessX <<= 1;
9
10          return lessX + x + y;
11      }
12
13      @Override
14      public void G(final int a, final int b, final int c, final int d) {
15          state[a] = mem.flip(fBlaMka(mem.flip(state[a]), mem.flip(state[b])));
16          state[d] = rotl64(state[d] ^ state[a], 32);
17
18          state[c] = mem.flip(fBlaMka(mem.flip(state[c]), mem.flip(state[d])));
19          state[b] = rotl64(state[b] ^ state[c], 24);
20
21          state[a] = mem.flip(fBlaMka(mem.flip(state[a]), mem.flip(state[b])));
22          state[d] = rotl64(state[d] ^ state[a], 16);
23
24          state[c] = mem.flip(fBlaMka(mem.flip(state[c]), mem.flip(state[d])));
25          // Cannot use the left rotation trick here: 63 % 8 != 0, so
26          // individual bytes do not stay the same, they change too.
27          state[b] = mem.flip(rotr64(mem.flip(state[b] ^ state[c]), 63));
28      }
29  }
```

Figure 3.6: BlaMka as instance of the Sponge class, the same rotation trick in action on lines 16, 19 and 22. Line 27 performs the complete set of rotations (no optimization).

4 Unit Testing Frameworks

For the lyra2-java project testing is important. It makes sure that the new implementation produces the same results as the reference implementation. Formal specification and verification methods (for example, as described in [34, 35]) would have been too complex in this situation.

A more practical verification approach is to provide reasonable unit test coverage for the new project, as suggested in [57]. However, this approach also provides its own classical questions: which tools to use [24] and when to stop writing tests [27].

Section 4.1 will introduce the basic practices and definitions. Section 4.2.1 will discuss how the two similar projects, the reference implementation of Lyra2 and of Argon2, address testing. Sections 4.2.2 and 4.2.3 motivate the choice of tools and techniques for the Java and Python projects.

4.1 Terms and Definitions

Unit testing is the process of verifying that the program produces expected results when given specific inputs. *Unit under test* is a common term that refers to the specific part of the code that is being tested by a specific *test case*. Unit testing is often powered by a *unit testing framework* which is a set of tools dedicated to simplify both writing and running unit tests. *Test Driven Development* is a software development practice which expects a piece of functionality to be produced together with the accompanying tests.

One notable property of a typical unit test is that it deals with the smallest logical piece of code possible: one particular function or a single method of a class. Consequently, distinct unit tests are ususally independent of each other and can be run in any order or in parallel. Parallelized unit test execution is therefore a desirable feature of a unit testing framework.

However, there are also cases when the complexity of a unit test is higher. For example, when a class heavily relies on data coming from a database, a unit test needs to *mock* the database connection and the data. *Mocking* is the process of simulating a real object with a simplified version of it.

It is often the case that many unit tests share the same logic. Specifically, when testing hash functions, this logic could be summarized with the following steps:

1. Select a configuration of the hash function.
2. Provide input data: a password, a salt, etc.
3. Compute the hash value and compare it to the correct one.

Writing a unit test for each combination of the hash function configuration and each set of input parameters is a daunting task. *Parametrized unit testing* allows to generate a template for a large number of unit tests and avoid the extra labour [50]. Therefore, unit testing frameworks that support this particular feature are of special interest in this work.

Code coverage indicates how well a program is tested. It is low when only a small portion of the program is tested, and high otherwise. There are many ways to measure code coverage, some of

Name	License	Parametrization	Parallelism
`Boost.Test`	Boost Software License	yes	needs 3rd party runner (`cmake`)
`JUnit4`	Eclipse Public License	yes	needs 3rd party runner (`mvn`)
`py.test`	MIT	yes	needs 3rd party plugin (`pytest.xdist`)

Table 4.1: Important features of several unit testing frameworks.

which are described in [27]. In practice, the most common code coverage metric is the number of lines of code (LOCs) covered by the test suite as a percentage of the total number of LOCs.

Other types of coverage include: *function* and *statement* coverage (i.e. the portion of functions or statements that has been executed), *branch* and *condition* coverage (i.e. the portion of conditions/branches executed in relation to the total number of *all possible* combinations) and many others. For the sake of simplicity, the projects developed in this work will not be using these metrics.

Continuous integration (CI) is the practice of running unit tests (and measuring code coverage changes) with every codebase change. This approach helps identify problems early and fix them quicker [57]. Continuous integration often occurs transparenly on a separate set of dedicated machines and its status is visible to the developers at all times. Its popularity and utility is undisputable, with such services like TravisCI [51], AppVeyor [11] and CircleCI [21] providing both commercial and free (for open source projects) continuous integration as a service.

4.2 Choice of the Unit Testing Framework

Although the reference Lyra2 project in [8] has automated tests, it does not use a dedicated unit testing framework. This is a reasonable choice which has its advantages. In particular, it allows to keep the complexity of the project lower and not rely on external libraries. The build process is somewhat simplified as well.

Furthermore, when it comes to the `C/C++` ecosystem, the choice of the framework is not trivial. The authors of Argon2 have also faced this choice and opted out to write their own test harness in [25]. Section 4.2.1 will provide a plausible explanation. Section 4.2.2 will provide motivation for my choice of the unit testing framework for the lyra2-java port and section 4.2.3 will describe the test suite for the Python script harness.

4.2.1 `Boost.Test`

There is a large choice of unit testing frameworks for the `C/C++` ecosystem. An extensive list can be found in [53, 54]. Given the number of available solutions, their comparison and an educated choice would be a large and daunting task for any developer. This could be one of the reasons why project authors sometimes tend to avoid actually using a unit testing framework. Instead, it is sometimes more practical to write a separate test file that would run a few sanity checks.

Consider the case of Argon2. As one can see in [33], the in-house `src/test.c` test program for Argon2 was introduced on 25th of January 2016. The commit hash starts with `7450df88` and it is number 317 out of (current) almost 600 in the version control history. So it is safe to say that this testing was introduced at the later stages of the project when the need for it was apparent [1].

A similar story could be observed for the reference Lyra2 project [8]. The rest of this section will explain why a rather popular unit testing library `Boost.Test` was not used in either of the projects. This particular library was tried because of my personal developer preference. A

[1] https://github.com/P-H-C/phc-winner-argon2/issues/85

```
#define BOOST_TEST_MODULE example_module_name
#include <boost/test/included/unit_test.hpp>

BOOST_AUTO_TEST_CASE(name_of_test_function) {
    BOOST_TEST(true);
}
```

Figure 4.1: Automatic unit test registration with the `Boost.Test` framework

different developer could attempt the same steps with another library, like `CTest` or `Google Test`.

`Boost` itself is a large collection of different libraries for `C++`. It was originally founded by Beman Dawes and David Abrahams but today the number of contributors is in the hundreds. The scope of the libraries spans from concurrent programming to regular expressions to linear algebra. There is a formal submission process [2] for any library that would like to be included into the collection as well as rigorous review in the mailing lists [3]. Many of the ideas that originate in `Boost` are later adopted by the `C++` language standard.

The `Boost.Test` is a unit testing framework that is part of `Boost`. It is licensed under the Boost Software License which is a free software license, OSI-approved and compatible with GPL. The library documnetation is located in [44].

The main features of the library are summarized in table 4.1. When it comes to password hashing, support for test parametrization becomes important because of the common test scenario described in Section 4.1. `Boost.Test` provides all the common features a unit testing library is expected to have. The tests can be completely decoupled from the implementation, subdivided into test suits and automatically registered with the test runner. The registration can be accomplished with a single `#include<...>` and a call to the `BOOST_AUTO_TEST_CASE` macro, as shown in 4.1.

Apart from that `Boost.Test` has a few convenience features. First of all, it was designed to be used as a header-only library. This means that the build process of a project needs only slight modification. Secondly, `Boost.Test` supports parametrized tests which are called "Data-Driven Test Cases" and can be found in [43]. First step is to declare such tests with the special `BOOST_DATA_TEST_CASE` macro. More details about its usage can be found in [19]. The data also has to be wrapped into a dataset, the procedure is described in detail in [20].

This section should have given the reader a hint that setting up `Boost.Test` for the real-world usage is not entirely painless. Even though the documentation provides an accurate description of the process, it still takes significant time to set everything up. Therefore it is understandable why Lyra2 and Argon2 projects chose to use their own testing solutions and not take advantage of test parametrization.

4.2.2 `JUnit` and `TestNG`

The Java ecosystem offers `JUnit`, which is a unit testing framework inspired by `xUnit`. The latter is a collective name for a specific architecture introduced by `NUnit`. This architecture is described below.

The main element of `JUnit` is a *test case* which is usually a class that contains testing logic. Such a class might have special methods called *fixtures* which are responsible for setting up and

[2] http://www.boost.org/development/submissions.html
[3] http://www.boost.org/community/groups.html#main

```java
@RunWith(Parameterized.class)
public class Lyra2Test {
    @Parameterized.Parameters
    public static Collection<Object[]> setupClass() {
        // Simplified initialization of a YAML data loader
        Yaml yaml = new Yaml();

        // A list of YAML file names with test vectors and resulting hashes
        String[] fnames = new String[] {"test-data-file_0.yml"};
        List<Object[]> entries = new ArrayList<>();

        for (String fname: fnames) {
            // Simplified loading of the YAML data from the file
            for (Object data : yaml.loadAll(reader)) {
                entries.add(new Object[]{data});
            }
        }

        return entries; // data provider has finished, returning result
    }

    private DataEntry entry;

    // Injection of test parameters
    public Lyra2Test(DataEntry entry) {
        this.entry = entry;
    }

    @Test
    public void simpleTest() {

        LyraParams params = new LyraParams(/* use entry to initialize */);

        byte[] hash = new byte[entry.klen];
        byte[] pass = entry.pass.getBytes();
        byte[] salt = entry.salt.getBytes();

        // Run the computation
        Lyra2.phs(hash, pass, salt, params);
        // Fetch the correct hash value
        byte[] correct_hash = pack.bytes(entry.hash);
        // Compare the computation result to the correct hash
        assertArrayEquals(correct_hash, hash);
    }
}
```

Figure 4.2: A simplified example of lyra2-java parametrized testing with `JUnit4`.

tearing down the context in which a test is executed. For example, this might include repeatedly creating a set of objects or opening a database connection. The methods marked with the `@Before` (respectively, `@After`) annotation are executed before (respectively, after) each individual test. The `@BeforeClass` (respectively, `@AfterClass`) annotation ensures that a method is run exactly once before (respectively, after) the test class is instantiated.

Several test cases can be collected into a single *test suite*. Finally, a program called a *test runner* is responsible for discovering the test cases, running them and reporting the results back to the user. The test runner can be instructed to run (or skip) specific test suits.

As well as `JUnit`, the Java ecosystem has a unit testing framework called `TestNG` [17]. There are several in-depth comparisons available online in [47, 56]. Arguably the most important differences follow.

Firstly, both frameworks allow to implement parametrization tests but the implementation details are significantly different. `JUnit4` requires the developer to mark the class with the `@RunWith` decorator as well as provide a class method marked with `@Parameters` and return a single `Collections<Object[]>` of data. On the other hand, `TestNG` has two distinct mechanisms, one of which uses `testng.xml` and the other is a `@DataProvider` method which returns either a single `Object[][]` or an `Iterator<Object[]>`. With the first mechanism, the test data is stored separately in `testng.xml` while the second approach allows to have a function that generates the data [4].

Formally, `TestNG` implements parametrized tests in a more flexible manner. However, in practice the `testng.xml` is not that widely used and the `Collections<Object[]>` `JUnit4` API is arguably cleaner than the generic-free way of `TestNG`. Finally, `JUnit4` is usually readily provided by Java IDEs and does not require additional setup. The feature parity with `TestNG` and general availability were the deciding factors in using this testing framework for lyra2-java.

Figure 4.2 provides a simplified example of the way unit tests are organized in lyra2-java. The testing logic is in the `simpleTest` method: configuration parameters are constructed using the values from the `entry` instance variable, then a call to `Lyra2.phs` is made and the resulting hash is compared to the correct answer by the `assertArrayEquals` call.

The `setupClass` method is marked with the `@Parametrized.Parameters` which makes this method the provider of data. The precomputed data comes from several `YAML` data files whose names are stored in the `fnames` variable. More information about the file structure and `YAML` in general is in appendix A. The **`for`** loop on line 12 sequentially opens the `YAML` data files, loads their contents and collects them in the `entries` variable. Once the `setupClass` method returns that variable, it is used by `JUnit` to initialize several instances of the class, providing each constructor with one element from the `entries` array.

This is the application of the parametrized tests approach which allows to run several hundreds of tests for different configurations and input values while writing the test code only once.

4.2.3 Unit Testing with `py.test`

Python is a very popular high-level general purpose scripting language created by Guido van Rossum. It has a built-in unit testing framework called `unittest` which comes together with the interpreter. It keeps a fairly compatible interface with the `xUnit` architecture described in 4.2.2. Other two notable unit testing frameworks in Python are `Node` and `py.test`. They both

[4] http://testng.org/doc/documentation-main.html#parameters

```python
import pytest
import subprocess

from pathlib import Path

bindir = Path(__file__).parent.parent.joinpath('bin42')
@pytest.mark.parametrize('path', list(bindir.glob("lyra2-*")))
@pytest.mark.parametrize('pwd', ['password', 'qwerty'])
@pytest.mark.parametrize('salt', ['salt', 'pepper'])
@pytest.mark.parametrize('k', [1, 2])
@pytest.mark.parametrize('t', [1, 2])
@pytest.mark.parametrize('m', [3, 4, 5, 6, 7, 8, 9, 10])
def test_sanity_0(path, pwd, salt, k, t, m):

    result = subprocess.run([path, pwd, salt, str(k), str(t), str(m)])

    assert result.returncode == 0
```

Figure 4.3: A simplified example of Lyra2 parametrized testing with `py.test`.

support an `xUnit` compatible layer and come as an external dependency. The features offered by these two external frameworks are superior to those provided by the built-in `unittest`.

Parametrized testing is tricky when it comes to a scripting language like Python. Even the built-in `unittest` framework could dynamically generate test classes, as demonstrated on stackoverflow [5]. However, there is no *designed* way to do so. The `Nose` library allows for parametrized testing through an elegant mechanism of Python generators (the `yield` keyword) that return functions to be run as test cases. The `py.test` framework goes the more `xUnit`-inspired route of using the `@mark.parametrized` decorator.

Unfortunately, the `Nose` unit testing framework has been in maintenance mode. This framework is no longer in active development and therefore is not recommended for new projects. This is why the Python build harness for the reference Lyra2 implementation uses `py.test`.

Figure 4.3 shows why parametrization testing is important when building different configurations of the reference Lyra2 implementation. Before running those configurations with a long list of parameters it is worth to make sure that the compiled executables actually work. However, writing a test for each individual possible configuration is not feasible. Instead, the first decorator `@pytest.mark.parametrize('path')` fetches all filenames that start with `lyra2-` (convention for Lyra2 executable names) from a specific directory. These names are later passed into a specific unit test in the `path` parameter.

This parametrized testing approach allows to write the testing logic once and reuse it for any possible Lyra2 configuration produced by the `Makefile` of the reference project.

[5] https://stackoverflow.com/a/20870875/1269892

5 Results

5.1 Algorithm-level Compatibility

The primary goal of this work was to port Lyra2 to Java so that it would produce the same hash values as the reference implementation when given the same inputs. Such algorithm-level compatibility was achieved and will be demonstrated in section 5.1.2 and section 5.1.3. The former deals with hand-picked test vectors while the latter demonstrates a reasonably large collection of randomly picked test vectors and a unit-testing framework which uses them to verify hash results.

5.1.1 Configuration Choice

By design Lyra2 has a large number of configurable parameters. This section will provide rationale for choosing particular values. The short summary can be found in table 5.1.

There are three sponges that could be tested: Blake2b, BlaMka and half-round BlaMka. Only the first two are in the manual testing shortlist because half-round BlaMka is similar to BlaMka. The sponge block size can be either 8, 10 or 12, so the extreme values made it into the shortlist. The columns of the memory matrix can be any positive number. The `Lyra2/src/runBenchCPU.sh` benchmarking script as well as other sources suggest that the values of 256 and 512 should be chosen.

Finally, time and memory costs are fixed at an arbitrary value of 100. The output length is chosen to be 10 so that the resulting hash value would fit easily on the page.

5.1.2 Manual Testing

Figure 5.1 shows a log of manual tests. New line delimits different Lyra2 configurations. The first line of each group represents a particular configuration group: `--outlen` is the output length, `--tcost` is the time cost, `--mcost` is memory cost. The second and the forth line in each group is the password and salt pair, and the third and fifth lines are the resulting hash values.

5.1.3 Automated Testing

The automated testing is possible because of the additional code that was added to the reference implementation. As mentioned in section 3.2, there is a `harness` branch which could be found

Parameter	Value
Sponge	Blake2b, BlaMka
Sponge blocks	8, 12
Sponge rounds	12
Columns in the memory matrix	256, 512
Time cost (number of iterations)	100
Memory cost (number of rows in the memory matrix)	100
Output length (bytes)	10

Table 5.1: Summary of parameter values for manually tested configurations

```
$ lyra2 --sponge blake2b --blocks 8 --columns 256 --outlen 10 --tcost 100 --mcost 100
> "password" "salt"
> 19 FD 3B 50 9A 03 0C DF 95 DA
> "The quick brown fox jumped over the lazy dog" "0123456789"
> 04 A0 BF 30 D1 E5 A5 05 53 E9

$ lyra2 --sponge blake2b --blocks 8 --columns 512 --outlen 10 --tcost 100 --mcost 100
> "password" "salt"
> 73 39 79 B6 C1 3C C1 F3 D7 17
> "The quick brown fox jumped over the lazy dog" "0123456789"
> A1 B0 18 F6 B6 79 5F E0 2A A4

$ lyra2 --sponge blake2b --blocks 12 --columns 256 --outlen 10 --tcost 100 --mcost 100
> "password" "salt"
> 9C 52 2A B9 18 30 F9 E7 09 55
> "The quick brown fox jumped over the lazy dog" "0123456789"
> 7D B2 9D C8 31 B4 E9 0E 10 22

$ lyra2 --sponge blake2b --blocks 12 --columns 512 --outlen 10 --tcost 100 --mcost 100
> "password" "salt"
> AC F2 B6 50 2D BC F0 62 DD 29
> "The quick brown fox jumped over the lazy dog" "0123456789"
> 4F 1B 03 6B C9 A2 09 C4 BC DA

$ lyra2 --sponge blamka --blocks 8 --columns 256 --outlen 10 --tcost 100 --mcost 100
> "password" "salt"
> 53 32 F3 D7 C4 9C 46 38 3C 1B
> "The quick brown fox jumped over the lazy dog" "0123456789"
> E7 6E 4B A0 81 B8 3C CF D6 64

$ lyra2 --sponge blamka --blocks 8 --columns 512 --outlen 10 --tcost 100 --mcost 100
> "password" "salt"
> D9 F9 F5 65 0D 05 88 D0 DF F6
> "The quick brown fox jumped over the lazy dog" "0123456789"
> 3A 3D 40 00 3E 33 44 45 B3 DD

$ lyra2 --sponge blamka --blocks 12 --columns 256 --outlen 10 --tcost 100 --mcost 100
> "password" "salt"
> C1 BC 48 80 99 1C E7 E6 52 18
> "The quick brown fox jumped over the lazy dog" "0123456789"
> 2E 4E 56 C7 5B 3D B7 F9 E0 30

$ lyra2 --sponge blamka --blocks 12 --columns 512 --outlen 10 --tcost 100 --mcost 100
> "password" "salt"
> 82 AF EB 03 5B E7 12 11 BE 63
> "The quick brown fox jumped over the lazy dog" "0123456789"
> F7 A8 56 D5 81 16 AA E5 C7 4D
```

Figure 5.1: Manual testing protocol, each configuration starts with `$`, each password and salt pair is followed by the resulting hash value. The hash value is printed in hexadecimal.

in the forked GitHub project [6] and the corresponding pull request [5]. This branch introduces the `Lyra2/tests/harness.py` script written in Python 3 which allows to compile several configurations of Lyra2 and then use those to compute and store hash values.

Additionally, the `Lyra2/tests/take.py` Python 3 script was used to choose a number of random hash values from those previously precomputed. These were then used in the Java project to unit-test the implementation. They were also included into the continuous integration service and their status can be verified by following [2]. In conclusion, both manual and automated testing indicate that the ported implementation does not contain immediately visible problems.

5.2 Performance Comparison

The single-threaded configuration of the reference C implementation was compared to its ported Java counterpart. For that, a separate GitHub repository was set up in [1]. This repository needs to control software written in both C and Java which is why an expressive, high-level scripting language was chosen to complete this task: Python 3.

5.2.1 Project Structure

The comparison project [1] is a combination of two Jupyter Notebooks [22] and an SQlite database. The two created notebooks follow the producer-consumer strategy. The `src/compare.ipynb` is the producer notebook which is responsible for compiling both C and Java implementations as well as dispatching computation tasks. The results are then collected into the `measurements.db` SQLite database (which also serves as a cache layer). The consumer is the `src/plot.ipynb` notebook which reads the data from the SQLite database and plots the figures shown below.

Lyra2 provides two main adjustable parameters: time and memory cost. Changing both of the parameters at the same time cannot be easily shown on a single graph which is why the comparison contains three main parts. Sections 5.2.2 and 5.2.3 show how several Lyra2 configurations behave when the time cost and the memory cost is fixed, respectively. Section 5.2.4 shows a grid of values for both time and memory costs. In all the sections, the two projects are compared using four configurations:

- Blake2b sponge with 256 columns, figures 5.2, 5.4 and 5.6
- Blake2b sponge with 2048 columns, figures 5.3, 5.5 and 5.7
- BlaMka sponge with 256 columns, figures 5.2, 5.4 and 5.8
- BlaMka sponge with 2048 columns, figures 5.3, 5.5 and 5.9

Configurations were run in parallel, creating one process per configuration. Running time was measured as the time between process creation and its termination. Memory consumption was recorded from the parent process and its peak value was stored. For more details, please consult the `src/compare.ipynb` and `src/plot.ipynb` notebooks from [1].

Finally, in section 5.2.4 the grid of time and memory cost parameters is 200×200 with a step of 10, which results in a total of $8 \times 21 \times 21 = 3528$ measurements used to build all the graphics.

5.2.2 Fixed Time Cost

There are two figures that show how running time and consumed memory depend on memory cost when the time cost is fixed. Figure 5.2 shows a 256-column and figure 5.3 shows a 2048-column

configuration of Lyra2. Both these figures show that the reference implementation works faster and consumes less memory than its Java counterpart. A nice property is that both the running time and memory consumption are changing roughly linearly as the memory cost parameter is changed. This is consistent with the fact that the memory cost parameter corresponds to the number of rows of the in-memory matrix. There are some outliers with respect to both running time and memory consumption which can be attributed to different system loads.

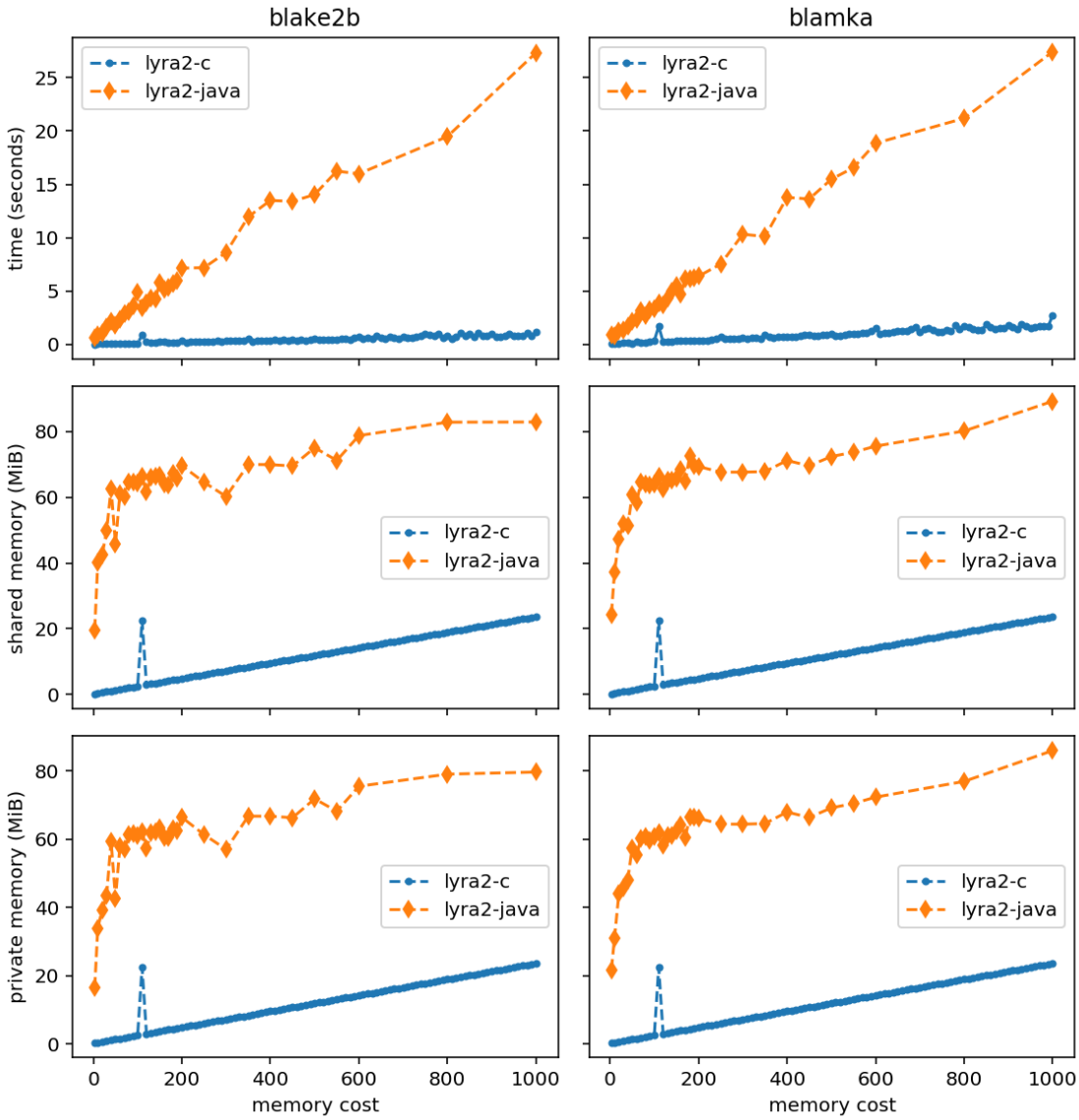


Figure 5.2: `lyra2-c` compared to `lyra2-java`: 256 columns, fixed time cost of 10. Blake2b sponge on the left, BlaMka sponge on the right.

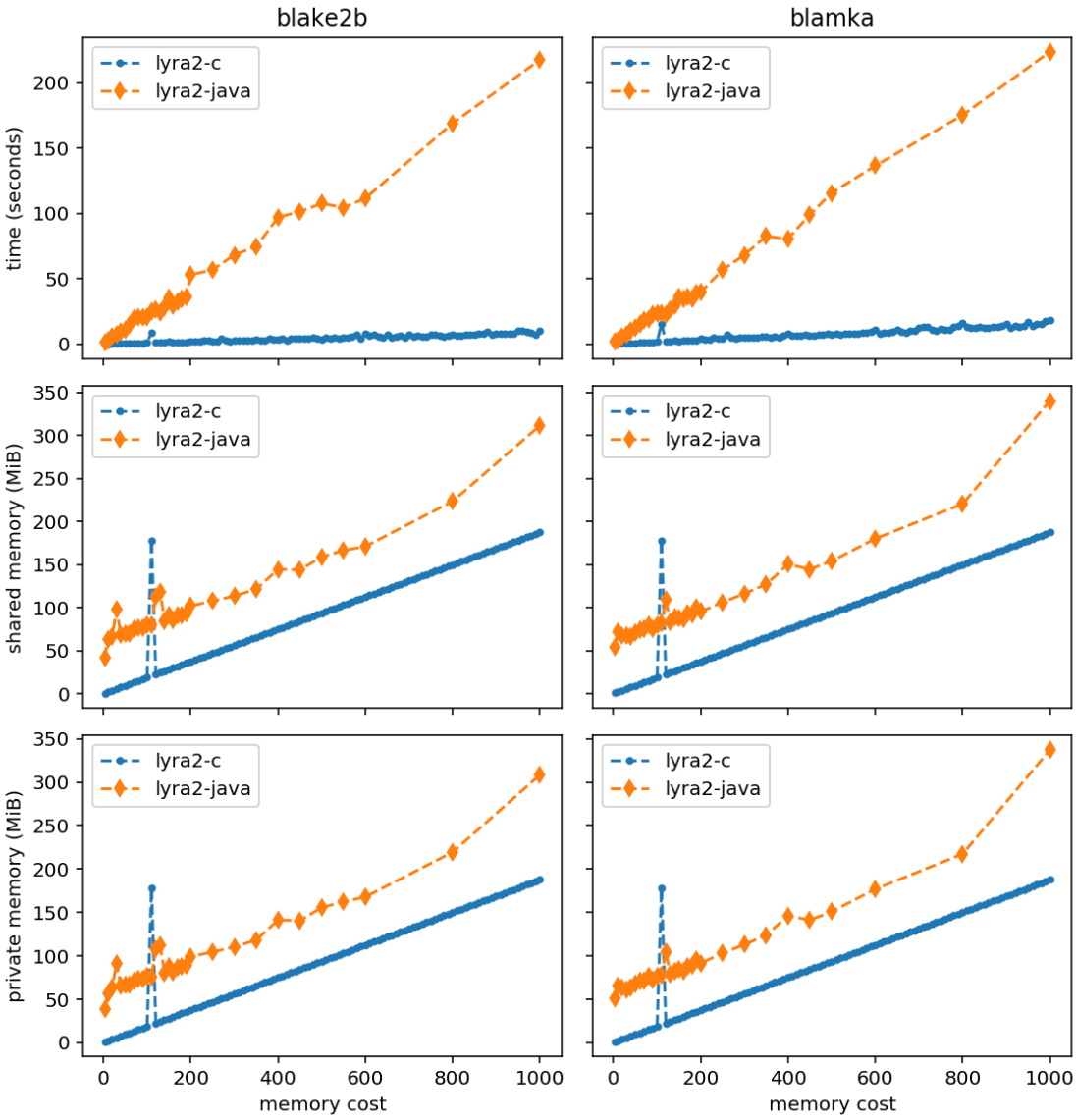


Figure 5.3: `lyra2-c` compared to `lyra2-java`: 2048 columns, fixed time cost of 10. Blake2b sponge on the left, BlaMka sponge on the right.

5.2.3 Fixed Memory Cost

There are two figures that show how running time and consumed memory depend on time cost when the memory cost is fixed. Figure 5.4 shows a 256-column and figure 5.5 shows a 2048-column configuration of Lyra2. Both these figures show that the original C implementation works faster and consumes less memory than its Java counterpart. The running time changes roughly linearly as the time cost parameter is changed which is consistent with the fact that time cost corresponds to the number of iterations done by Lyra2. Memory consumption stays roughly the same which is also consistent with the fact that the largest memory consumer is the in-memory matrix. This matrix has the same size for all of the configurations. Admittedly, there are some deviations in memory consumption of the Java implementation. The second set of graphs 5.5 even has a slight downward trend. The possible reasons for that include: the built-in garbage collector and the fact that the measurements run for several days, resulting in potentially different system loads.

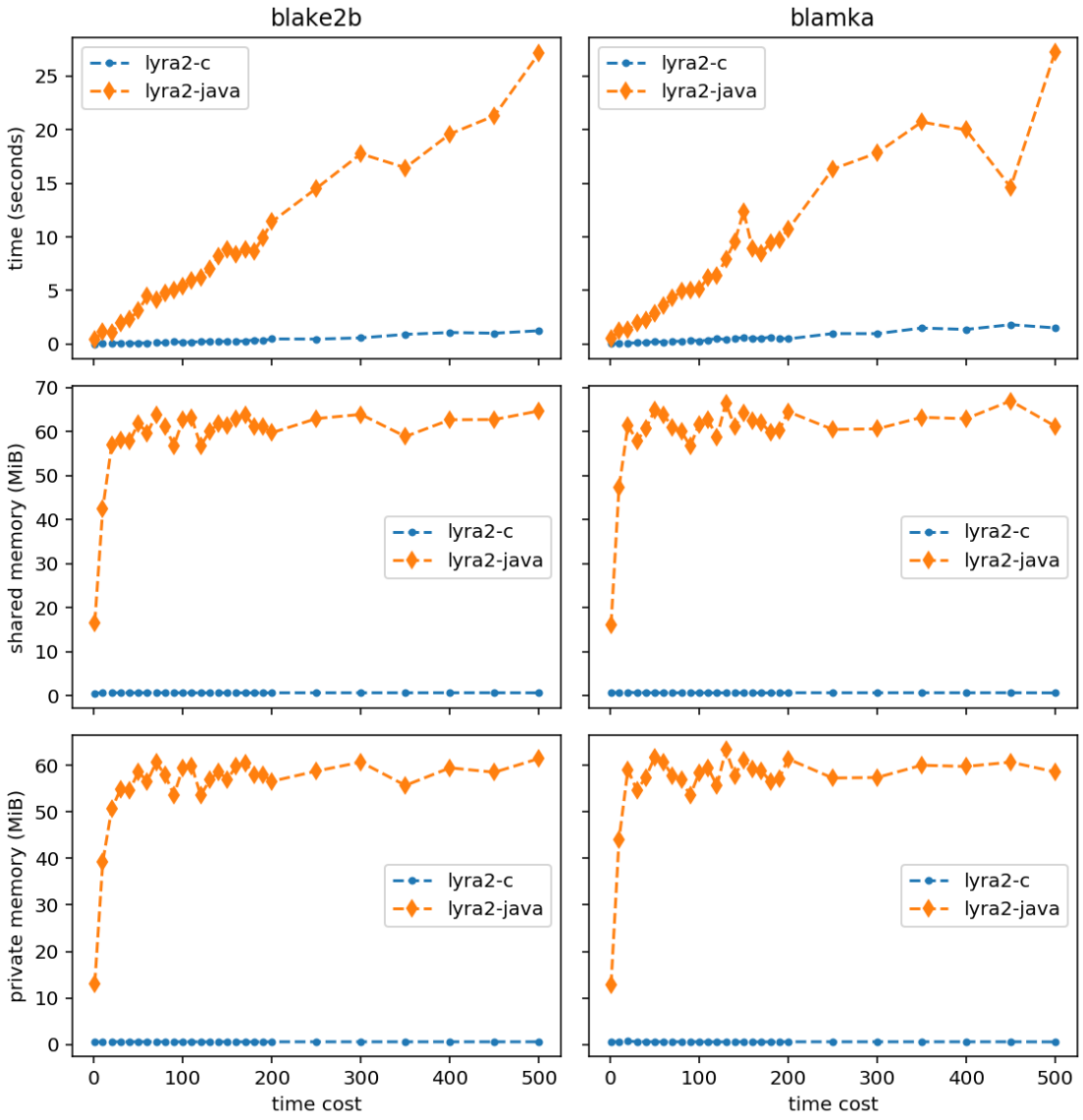


Figure 5.4: `lyra2-c` compared to `lyra2-java`: 256 columns, fixed memory cost of 20. Blake2b sponge on the left, BlaMka sponge on the right.

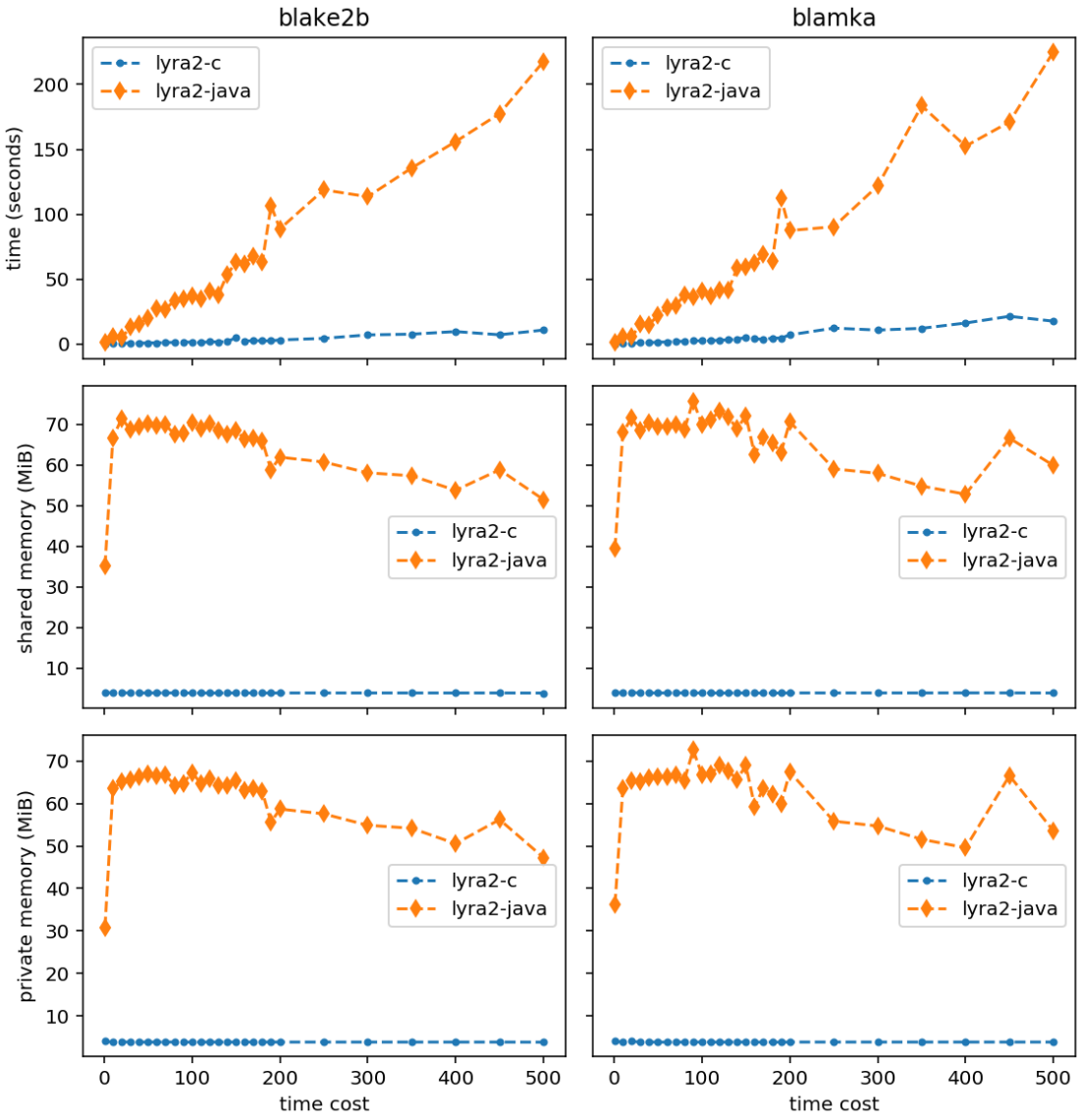


Figure 5.5: `lyra2-c` compared to `lyra2-java`: 2048 columns, fixed memory cost of 20. Blake2b sponge on the left, BlaMka sponge on the right.

5.2.4 Variable Time and Memory Costs

There are four figures that show running time and memory consumption of Lyra2 when both time and memory costs change. Figures 5.6 and 5.7 correspond to a 256- and 2048-column configurations of Lyra2 that both use the Blake2b sponge. Figures 5.8 and 5.9 are the 256- and 2048-column configurations of Lyra2 with the BlaMka sponge. All of these four figures share the same properties. First of all, the reference implementation is faster and consumes less memory than its Java counterpart. Secondly, the running time as well as required space depend roughly linearly on the time and memory cost parameters. However, together they create a quadratic time growth during the *Wandering phase*. This still allows for predictable and fine-tuned control of the time- and memory resources required by the algorithm.

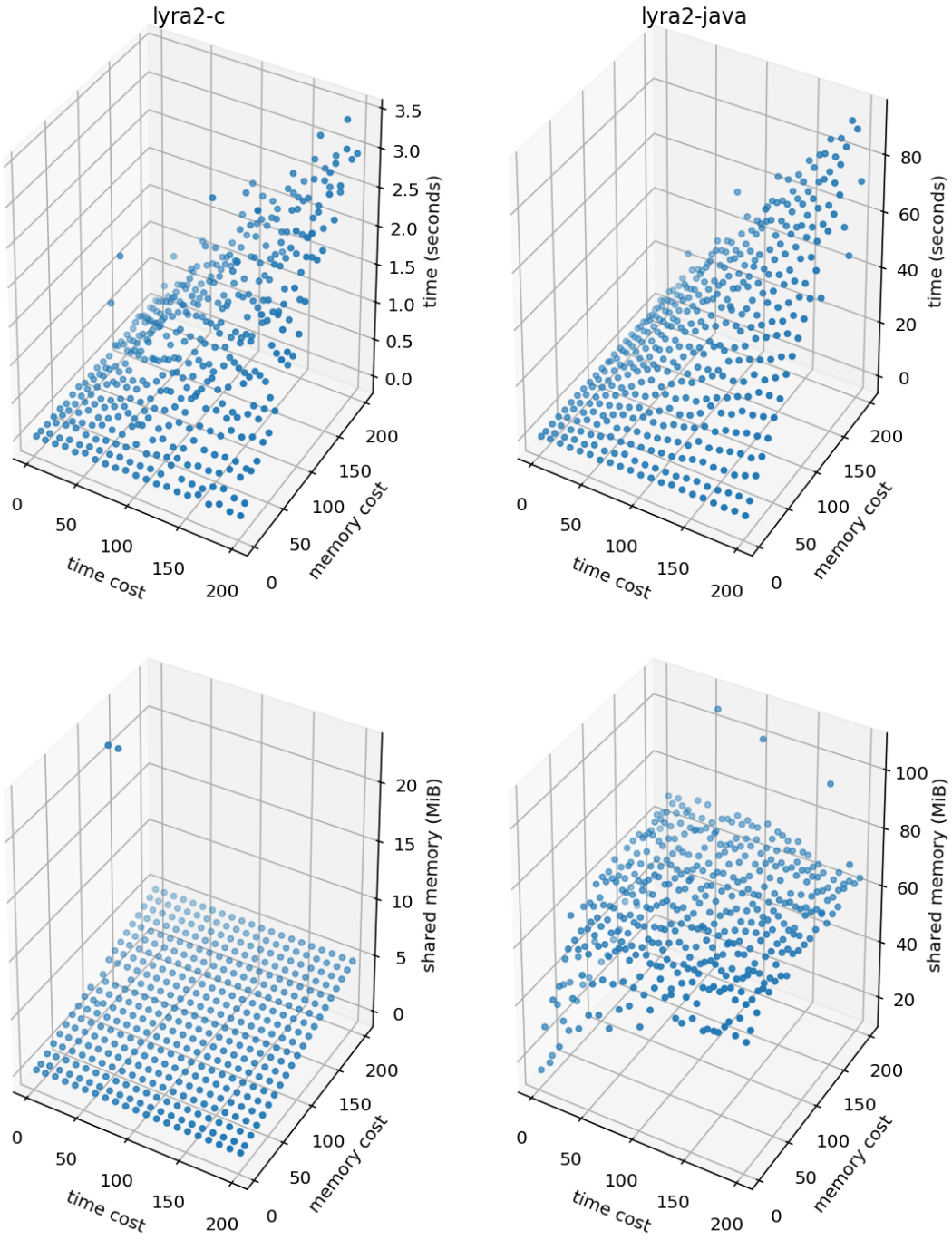


Figure 5.6: `lyra2-c` compared to `lyra2-java`: Blake2b sponge, 256 columns.

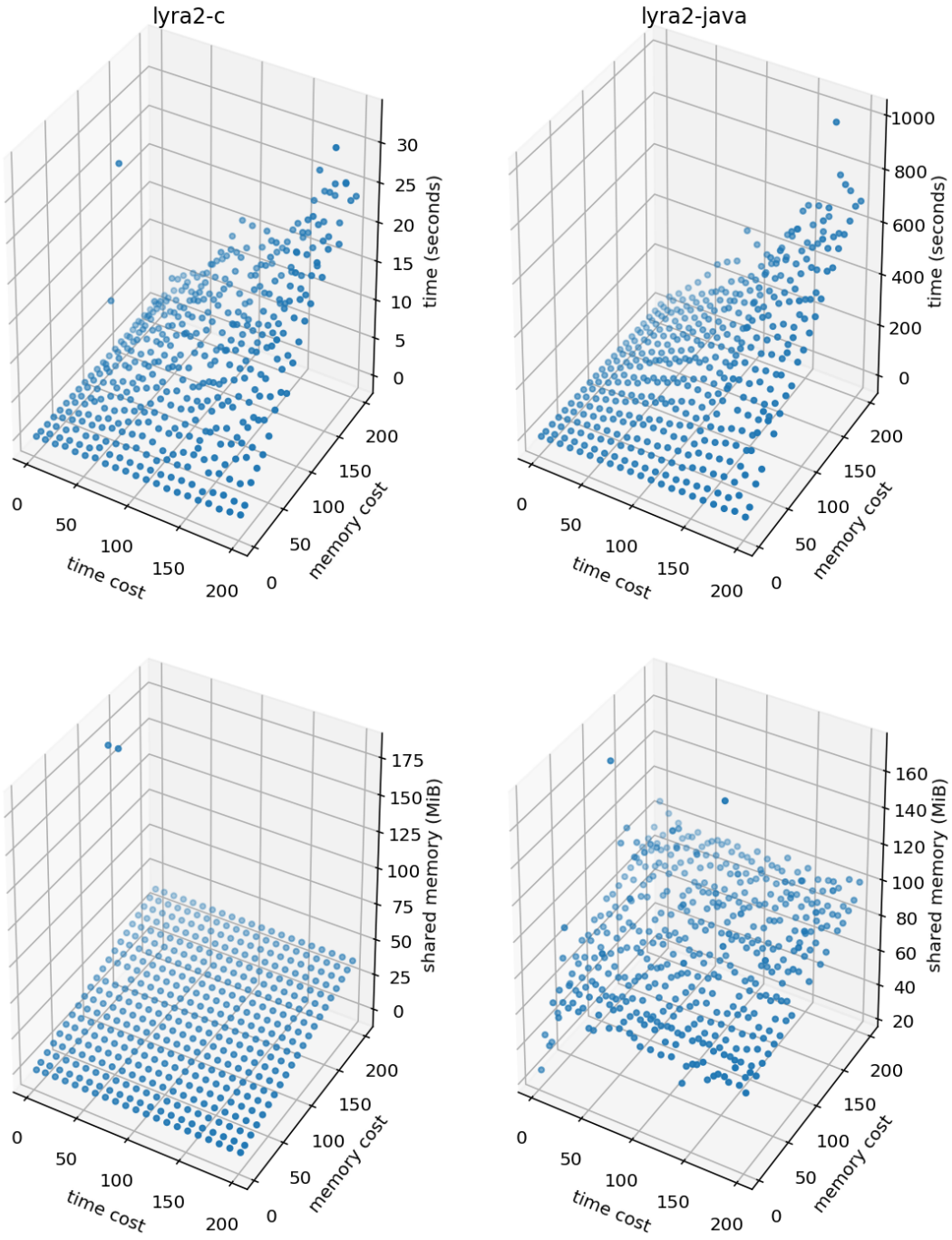


Figure 5.7: `lyra2-c` compared to `lyra2-java`: Blake2b sponge, 2048 columns.

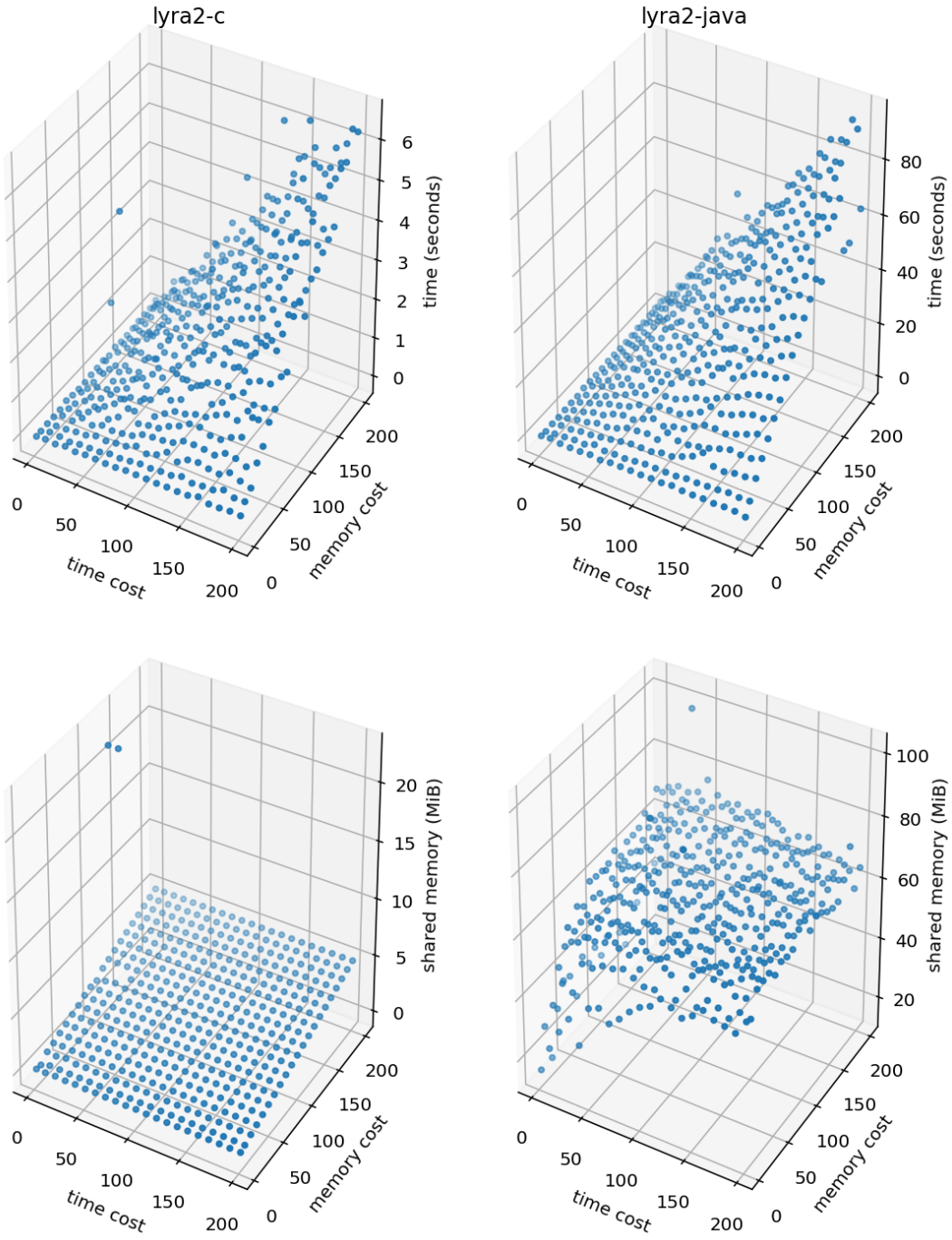


Figure 5.8: `lyra2-c` compared to `lyra2-java`: BlaMka sponge, 256 columns.

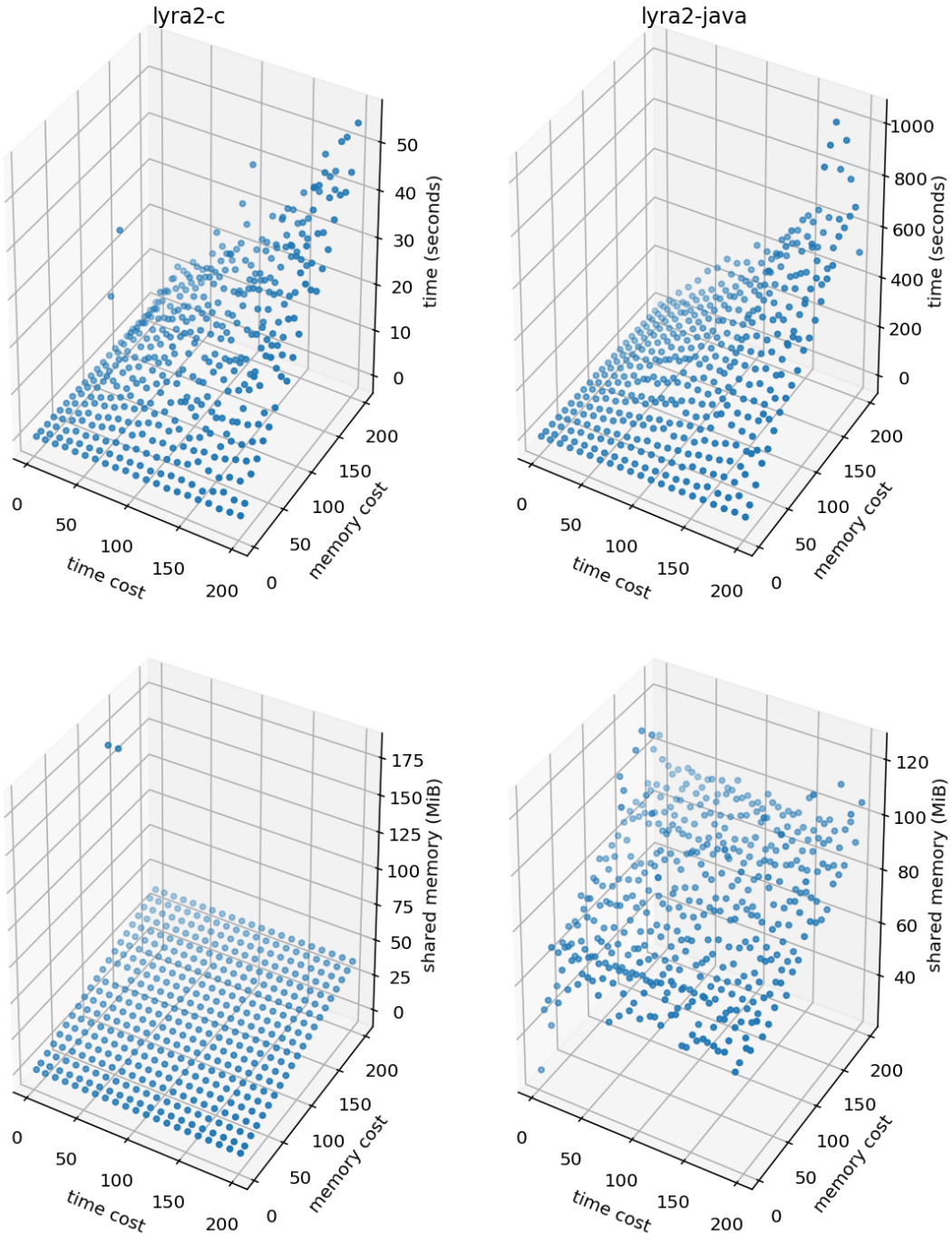


Figure 5.9: `lyra2-c` compared to `lyra2-java`: BlaMka sponge, 2048 columns.

5.2.5 Comparison Conclusion

The Java implementation was outperformed by the reference C implementation. This could in part be attributed to language ecosystems or programming skills but it also cannot be denied that the ported version takes a lot of extra steps in order to ensure algorithm-level compatibility. These include the extra rotations requried to simulate little-endian behaviour, as well as simulating pointer arithmetic or unsigned arithmetic for large 64-bit integers.

5.3 Android Application

Part of the porting effort is a small proof of concept mobile application. This application demonstrates that using a native Java library in an Android project is convenient.

One of the first steps is to determine the minimum versions of Android devices and API levels necessary. Since lyra2-java makes use of unsigned long number arithmetic, Java 1.8 support is required. This translates into a minimum Android versions of 7.0 and an API level of 24 [1].

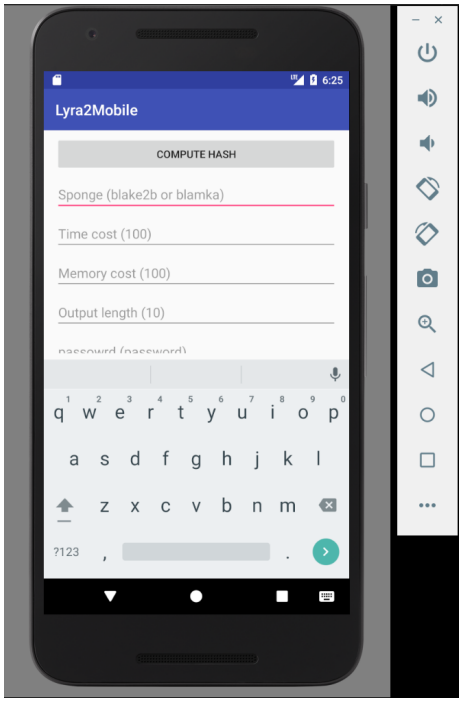
The most common development environment for Android applications is the Android Studio. Support for Java 1.8 features has not yet landed into the release version of this IDE. So, in order to be able to develop a mobile application with Lyra2 a development version `3.0` needs to be installed. If you face problems then you might find helpful advice in appendix B.

Once these preliminary steps are performed, the development of the application becomes straightforward. Android Studio is based on IntelliJ Studio and therefore picks up lyra2-java from the Maven Central repository in a matter of a couple clicks.

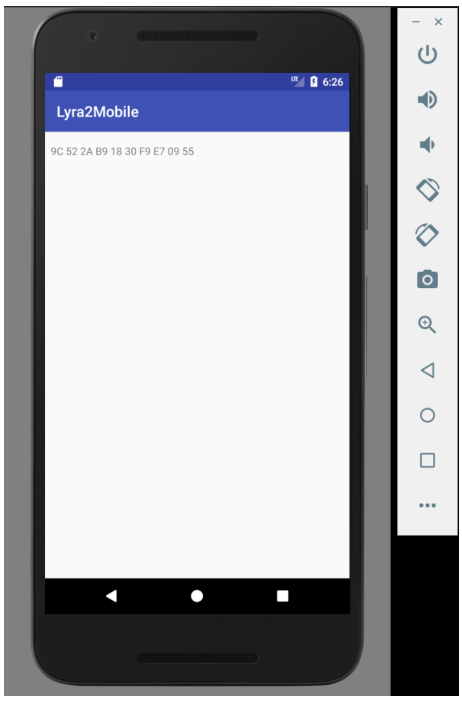
As a result, a simple Android application was developed and is available on GitHub [7]. It consists of a main screen and a results screen. The main screen allows the user to choose the particular configuration of Lyra2. By default, the Blake2b sponge with the time and memory cost of 100 will be used. The password and salt fields are set to the default values of "password" and "salt" respectively, the number of columns is 256 with each column being 12 blocks wide. The permutation inside the sponge makes a total 12 rounds of computations. All of the parameters above can be adjusted.

Figure 5.10 demonstrates the hash value after running with default parameters as well as the hash value when using BlaMka instead of Blake2b. The hash values match those from the manual testing section 5.1.2. In each case the computation lasted approximately 60 seconds which is consistent with the desktop version times. However, it should be noted that the application was tested in an emulator and not on an actual device.

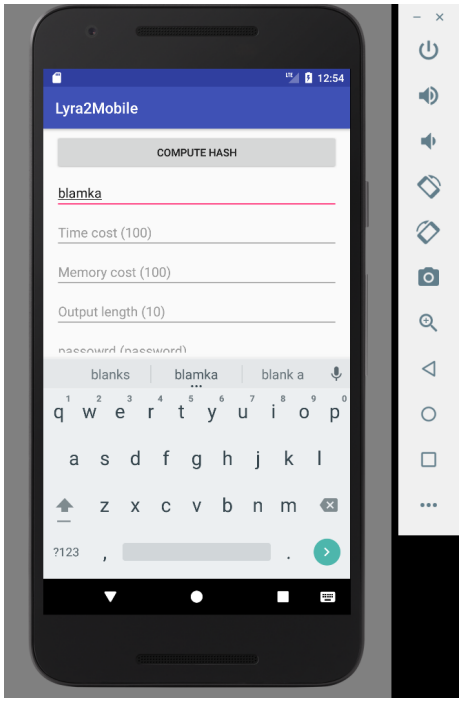
[1] https://developer.android.com/guide/platform/j8-jack.html



(a) Main screen (Blake2b sponge, default)



(b) Resulting hash (Blake2b sponge, default)



(c) Main screen (BlaMka sponge)

(d) Resulting hash (BlaMka sponge)

Figure 5.10: Android application running Lyra2. Each of the two configurations runs for $\approx$ 60 seconds. Parameter values on the left, resulting hash in hexadecimal on the right.

6 Conclusion

The reference implementation of Lyra2 was successfully ported from C99 to Java 1.8. The ported counterpart generally works slower and requires more memory. This can be explained by the numerous extra steps that must take place in order to ensure algorithm-level compatibility (i.e. producing the same hash values when given the same inputs). There are scenarios when performance can be traded for ease of integration and adoption, such as the case with smartphones. In the Android ecosystem the ported Lyra2 implementation can be integrated as a matter of a few clicks of the mouse button.

In my opinion, there are several viable options for further research. Firstly, algorithm-level compatibility can be sacrificed in favour of performance. However, personally I believe that producing the same hash values is key for successful adoption. Secondly, the parallel version of Lyra2 could be ported into Java. This would introduce another interesting layer for compatibility questions: concurrency. Finally, an exhaustive optimization effort of the Java implementation could be attempted as well. Although I have successfully found one significant optimization trick, there are likely to be more.

Bibliography

References

[9] Ross Anderson, Eli Biham, and Lars Knudsen. *Serpent: A Proposal for the Advanced Encryption Standard*. Tech. rep. 1998. URL: http://www.cl.cam.ac.uk/~rja14/serpent.html (visited on 02/09/2017).

[10] E. R. Andrade et al. „Lyra2: Efficient Password Hashing with High Security against Time-Memory Trade-Offs“. In: *IEEE Transactions on Computers* 65.10 (2016), pp. 3096–3108. ISSN: 0018-9340. DOI: 10.1109/TC.2016.2516011.

[14] Jean-Philippe Aumasson et al. *BLAKE2: simpler, smaller, fast as MD5*. Cryptology ePrint Archive, Report 2013/322. http://eprint.iacr.org/2013/322. 2013.

[15] Guido Bertoni et al. *Duplexing the sponge: single-pass authenticated encryption and other applications*. Cryptology ePrint Archive, Report 2011/499. http://eprint.iacr.org/2011/499. 2011.

[16] Guido Bertoni et al. *Keccak*. Cryptology ePrint Archive, Report 2015/389. http://eprint.iacr.org/2015/389. 2015.

[18] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. *Argon2*. Tech. rep. https://password-hashing.net/submissions/specs/Argon-v3.pdf. 2015.

[23] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002, p. 238. ISBN: 3-540-42580-2.

[24] E. Daka and G. Fraser. „A Survey on Unit Testing Practices and Problems“. In: *2014 IEEE 25th International Symposium on Software Reliability Engineering*. 2014, pp. 201–211. DOI: 10.1109/ISSRE.2014.11.

[26] Carolynn Burwick Don et al. *MARS — a candidate cipher for AES*. Tech. rep. 1998. URL: http://www.nic.funet.fi/~bande/docs/crypt/system/mars.pdf (visited on 02/09/2017).

[27] Frank Elberzhager et al. „Reducing test effort: A systematic mapping study on existing approaches“. In: *Information and Software Technology* 54.10 (2012), pp. 1092 –1106. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2012.04.007.

[28] Niels Ferguson et al. *The Skein Hask Function Family*. 2009. URL: http://www.skein-hash.info/sites/default/files/skein1.3.pdf (visited on 02/09/2017).

[29] Christian Forler, Stefan Lucks, and Jakob Wenzel. *Catena: A Memory-Consuming Password-Scrambling Framework*. Cryptology ePrint Archive, Report 2013/525. http://eprint.iacr.org/2013/525. 2013.

[30] Praveen Gauravaram et al. *Grøstl — a SHA-3 candidate*. 2011. URL: http://www.groestl.info/Groestl.pdf (visited on 02/09/2017).

[31] Marcos A. Simplicio Jr. et al. *Lyra2: Password Hashing Scheme with improved security against time-memory trade-offs*. Cryptology ePrint Archive, Report 2015/136. http://eprint.iacr.org/2015/136. 2015.

[32] Ed. K. Moriarty, B. Kaliski, and A. Rusch. *PKCS #5: Password-Based Cryptography Specification, Version 2.1*. Tech. rep. https://tools.ietf.org/html/rfc8018. 2017.

[34] Axel Van Lamsweerde. „Formal Specification: a Roadmap“. In: *Proceedings of the Conference on the Future of Software Engineering*. 2000, p. 147.

[35] Peter Müller and Arnd Poetzsch-Heffter. „Formal Specification Techniques for Object-Oriented Programs“. In: *Proceedings of 16th International Conference on Software Engineering*. 1994, pp. 223–223.

[36] C. Percival and S. Josefsson. *The scrypt Password-Based Key Derivation Function*. Tech. rep. https://tools.ietf.org/html/rfc7914. 2016.

[37] Alexander Peslyak. *yescrypt — a Password Hashing Competition submission*. Tech. rep. https://password-hashing.net/submissions/specs/yescrypt-v2.pdf. 2015.

[40] Thomas Pornin. *The Makwa password hashing function*. Tech. rep. https://password-hashing.net/submissions/specs/Makwa-v1.pdf. 2015. (Visited on 02/09/2017).

[41] Thomas Pornin. *The* MAKWA *Password Hashing Function, specification v1.1*. Tech. rep. http://www.bolet.org/makwa/makwa-spec-20150422.pdf. 2015. (Visited on 02/09/2017).

[42] Ronald L. Rivest et al. *The RC6 Block Cipher*. Tech. rep. 1998. URL: http://people.csail.mit.edu/rivest/pubs/RRSY98.pdf (visited on 02/09/2017).

[45] Bruce Schneier et al. „Twofish: A 128-Bit Block Cipher“. In: *First Advanced Encryption Standard (AES) Conference*. 1998. URL: https://www.schneier.com/academic/twofish/ (visited on 02/09/2017).

[46] *Specification for the Advanced Encryption Standard (AES)*. Federal Information Processing Standards Publication 197. 2001. URL: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[50] N. Tillmann, J. de Halleux, and T. Xie. „Parameterized unit testing: theory and practice“. In: *2010 ACM/IEEE 32nd International Conference on Software Engineering*. Vol. 2. 2010, pp. 483–484. DOI: 10.1145/1810295.1810441.

[52] Jos Wetzels. *Open Sesame: The Password Hashing Competition and Argon2*. Cryptology ePrint Archive, Report 2016/104. http://eprint.iacr.org/2016/104. 2016.

[57] Laurie Williams and Gunnar Kudrjavets. „On the Effectiveness of Unit Test Automation at Microsoft“. In: *International Symposium on Software Reliability Engineering (ISSRE)*. 2010. URL: https://collaboration.csc.ncsu.edu/laurie/Papers/Unit_testing_cameraReady.pdf (visited on 02/09/2017).

[58] Hongjun Wu. *The Hash Function JH*. 2011. URL: http://www3.ntu.edu.sg/home/wuhj/research/jh/jh_round3.pdf (visited on 02/09/2017).

Online References

[1] Lisianoi Aleksandr. *Lyra2 comparison project*. 2017. URL: https://github.com/all3fox/lyra2-compare (visited on 02/09/2017).

[2] Lisianoi Aleksandr. *Lyra2 continuous integration on Travis*. 2017. URL: https://travis-ci.org/all3fox/lyra2-java (visited on 02/09/2017).

[3] Lisianoi Aleksandr. *Lyra2, Java port*. 2017. URL: https://github.com/all3fox/lyra2-java (visited on 02/09/2017).

[4] Lisianoi Aleksandr. *Lyra2 package on Maven Central, version 1.3*. 2017. URL: http://search.maven.org/#artifactdetails%7Ccom.github.all3fox%7Clyra2%7C1.3%7Clyra2 (visited on 02/09/2017).

[5] Lisianoi Aleksandr. *Lyra2 pull request of the "harness" branch*. 2017. URL: https://github.com/leocalm/Lyra/pull/7 (visited on 02/09/2017).

[6] Lisianoi Aleksandr. *Lyra2 reference implementation (Lisianoi's fork)*. 2017. URL: https://github.com/all3fox/Lyra (visited on 02/09/2017).

[7] Lisianoi Aleksandr. *Lyra2Mobile, an Android application*. 2017. URL: https://github.com/all3fox/lyra2-mobile (visited on 02/09/2017).

[8] Leonardo C. Almeida et al. *Lyra2 reference implementation*. 2017. URL: https://github.com/leocalm/Lyra (visited on 02/09/2017).

[11] *AppVeyor Homepage (#1 Continuous Delivery service for Windows)*. URL: https://www.appveyor.com/ (visited on 02/09/2017).

[12] *Argon2 in Django Web Framework*. 2017. URL: https://docs.djangoproject.com/en/1.11/topics/auth/passwords/#using-argon2-with-django (visited on 02/09/2017).

[13] *Argonishche (Argon2 in Yandex)*. 2017. URL: https://github.com/yandex/argon2 (visited on 02/09/2017).

[17] Cédric Beust and contributors. *TestNG Homepage*. 2017. URL: http://testng.org/doc/index.html (visited on 02/09/2017).

[19] *Boost Data-Driven Unit Tests*. 2017. URL: http://www.boost.org/doc/libs/1_65_1/libs/test/doc/html/boost_test/tests_organization/test_cases/test_case_generation/datasets_auto_registration.html (visited on 02/09/2017).

[20] *Boost Datasets for Data-Driven Unit Tests*. 2017. URL: http://www.boost.org/doc/libs/1_65_1/libs/test/doc/html/boost_test/tests_organization/test_cases/test_case_generation/datasets.html (visited on 02/09/2017).

[21] *CircleCI Homepage (Build Faster. Test More. Fail Less.)* URL: https://circleci.com/ (visited on 02/09/2017).

[22] The Jupyter Community and Steering Council. *Project Jupyter*. 2017. URL: http://jupyter.org/ (visited on 02/09/2017).

[25] Daniel Dinu et al. *The Unit Testing Harness of Argon2*. 2017. URL: https://github.com/P-H-C/phc-winner-argon2/blob/68dd41f75c4c9df95f403154c0e19252fcf0f513/src/test.c (visited on 02/09/2017).

[33] Dmitry Khovratovich, Jean-Philippe Aumasson, and contributors. *The Argon2 GitHub repository*. 2017. URL: https://github.com/P-H-C/phc-winner-argon2/ (visited on 02/09/2017).

[38] Remko Popma. `picocli` — *a mighty tiny command line interface*. 2017. URL: http://picocli.info/ (visited on 02/09/2017).

[39] Thomas Pornin. *Makwa Homepage*. 2017. URL: http://www.bolet.org/makwa/ (visited on 02/09/2017).

[43] Gennadiy Rozental, Raffi Enficiaud, and BOOST contributors. *Data Driven Test Cases*. 2017. URL: http://www.boost.org/doc/libs/1_65_1/libs/test/doc/html/boost_test/tests_organization/test_cases/test_case_generation.html (visited on 02/09/2017).

[44] Gennadiy Rozental, Raffi Enficiaud, and BOOST contributors. *Documentation for the Boost.Test library*. 2017. URL: http://www.boost.org/doc/libs/1_65_1/libs/test/doc/html/index.html (visited on 02/09/2017).

[47] *TestNG vs JUnit4 comparison*. 2017. URL: https://www.mkyong.com/unittest/junit-4-vs-testng-comparison/ (visited on 02/09/2017).

[48] CATENA-VARIANTS *GitHub Repository*. 2017. URL: https://github.com/medsec/catena-variants (visited on 02/09/2017).

[49] CATENE-AXUNGIA *GitHub Repository*. 2017. URL: https://github.com/medsec/catena-axungia (visited on 02/09/2017).

[51] *TravisCI Homepage*. URL: https://travis-ci.org/ (visited on 02/09/2017).

[53] Wikipedia. *A List of Unit Testing Frameworks for C*. 2017. URL: https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks#C (visited on 02/09/2017).

[54] Wikipedia. *A List of Unit Testing Frameworks for C++*. 2017. URL: https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks#C.2B.2B (visited on 02/09/2017).

[55] Wikipedia. *Sponge Function*. 2017. URL: https://en.wikipedia.org/wiki/Sponge_function (visited on 02/09/2017).

[56] Wikipedia. *TestNG vs JUnit comparison*. 2017. URL: https://en.wikipedia.org/wiki/TestNG#Comparison_with_JUnit (visited on 02/09/2017).

A Configuration and Test File Format

The data for parametrized tests is stored in YAML files. YAML stands for *YAML Ain't Markup Language* and is a superset of JSON, another popular data format. YAML targets human readability and provides native support for custom datatypes: scalars, arrays, structures, etc. Most programming languages provide support for reading and writing of YAML files, inclusing both Java and Python.

A single file in YAML can hold several documents. This is leveraged by the Python test harness: each test data file corresponds to the Lyra2 executable, so the compile time parameters are fixed. They are stored together with all the runtime parameters and the resulting hash. See figure A.1 for reference. The `hash` field is stored as an array of strings which ensures that different YAML libraries deduce the content type of this array correctly. The `---` is a delimiter between the two YAML documents. This feature allows to store several sets of data in one file.

The YAML format is also used to store default configuration for the Python test harness. It can be found on the `harness` branch of the forked reference repository [6] in the `harness.yml` file. The primary parameters are `build_path` and `makefile_path`. The first one defines the location for the compiled Lyra2 executables and the second one points to the original `Makefile`.

The `matrix` group of parameters defines the build matrix. The `option` parameter configures the type of Lyra2 executable built by the `Makefile`, which is a generic version for the `x86_64` architecture by default. The `threads` parameter determines the parallelism degree and is set to 1. The `columns`, `sponge`, `rounds` and `blocks` correspond directly to compile-time Lyra2 parameters. Finally, the `bench` parameter determines if the test vectors should be included into the compiled executable. By default it is set to 0 which means those vectors are skipped.

The `data` group of parameters specifies the test vectors which will be used when generating hash values with `./harness.py compute`. That group includes `pass` — an array of passwords, `salt` — an array of salts, `klen` — an array of output lenghts (i.e. the length of the hash), `tcost` — an array of time costs and finally `mcost` — an array of memory costs. The `./harness.py` script generates a cartesian product of all of the array members and then runs every compiled executable using those values as test vectors. The results are stored in the `data_path` directory which can be configured as well. The `./harness.py` script does not overwrite old hash values, so you will have to remove them manually if regeneration is required.

Finally, compilation flags are also part of the configuration and their defaults can be seen in A.2.

```
blocks: 8
columns: 16
hash: [0f, ee, bd, 1f, '00', 2a, 5b, '87', '71', ee]
klen: 10
mcost: 3
pass: password
rounds: 1
salt: s
sponge: blake2b
tcost: 1
threads: 1
---
blocks: 8
columns: 16
hash: [0f, 7d, e3, 3c, e3, 9e, 0c, f9, 8e, '70']
klen: 10
mcost: 10
pass: password
rounds: 1
salt: s
sponge: blake2b
tcost: 1
threads: 1
```

Figure A.1: An example of a test data file.

```
CFLAGS:
  - -std=c99
  - -Wall
  - -pedantic
  - -O3
  - -msse2
  - -ftree-vectorizer-verbose=1
  - -fopenmp
  - -funroll-loops
  - -march=native
  - -Ofast
  - -mprefer-avx128
  - -flto
```

Figure A.2: The compilation flags used by the reference implementation.

B Android Studio Preview

At the moment using Lyra2 in an Android application requires a beta (i.e. preview) version of Android Studio. The instructions on the official webpage [1] warn the developer about the need to update plugins. However, this is not the only problem which you may face when running the preview version.

If you require an emulator to test the application, you may be treated with the following error traceback presented in B.1. Different kinds of advice is supposed to help with the problem: preloading the `libstdc++.so.6` library ahead of time, adding the user who runs Android Studio to the `video` group, etc. On my machine (a ThinkPad E570 notebook with a dedicated GTX 950M graphics card running ArchLinux), the solution that worked was to instruct Android Studio to rely on system libraries. This can be achieved by setting an environment variable:

```
ANDROID_EMULATOR_USE_SYSTEM_LIBS=1 ./studio.sh
```

```
Emulator: libGL error: unable to load driver: i965_dri.so
Emulator: libGL error: driver pointer missing
Emulator: libGL error: failed to load driver: i965
Emulator: libGL error: unable to load driver: i965_dri.so
Emulator: libGL error: driver pointer missing
Emulator: libGL error: failed to load driver: i965
Emulator: libGL error: unable to load driver: swrast_dri.so
Emulator: libGL error: failed to load driver: swrast
Emulator: X Error of failed request:  BadValue (integer parameter out of range for operation)
Emulator: Major opcode of failed request:  155 (GLX)
Emulator: Minor opcode of failed request:  24 (X_GLXCreateNewContext)
Emulator: Value in failed request:  0x0
Emulator: Serial number of failed request:  42
Emulator: Current serial number in output stream:  43
Emulator: Process finished with exit code 1
```

Figure B.1: An obscure error traceback connected to `libGL`.

[1] https://developer.android.com/studio/preview/index.html