# Porting Lyra2 to Java

## Optimizing and Unit Testing a Memory-hard Password Hashing Scheme

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:** Thomas Grechenig
**Assistance:** Clemens Hlauschek
Florian Fankhauser

Vienna, January 19, 2018

# Abstract

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

The question of secure password storage is what motivated the Password Hashing Competition which took place between 2013 and 2015. The competition resulted in the development of several memory-hard password hashing algorithms. These algorithms are designed to consume a significant amount of memory as well as processing time. Requiring more memory is the novel feature which should discourage highly parallelized password recovery attacks that use graphics processing units (GPUs) (or other hardware like field-programmable gate arrays (FPGAs) and application-specific integrated circuits (ASICs)). The cornerstone assumption is that such hardware has a lot less memory per processing unit than the central processing units (CPUs) of desktop computers and mobile phones.

Lyra2 is one of the finalists of the Password Hashing Competition. It was originally implemented in C and this work describes its porting to Java. An implementation in a different language should provide an adoption boost and simplify the algorithm's integration into other projects.

The porting effort poses several challenges. A number of discrepancies between C and Java are addressed: endianness, unsigned arithmetic for 64-bit integers and simulation of pointer arithmetic for array accesses. Both implementations also produce the same hash values when provided with identical inputs. This presents a difficulty for verification and unit testing. A cross-project continuous integration and unit testing setup that overcomes this difficulty is presented and discussed.

This work also provides a performance comparison. In short, the reference C implementation is faster and consumes exactly as much memory as instructed. Its Java counterpart is somewhat slower and more memory demanding. This is in part explained by the significant compatibility effort, as well as programming skill and ecosystem. One important scenario where the Java implementation might be preferred is the Android platform, where it could be integrated into any application like a simple external library.

The ported Java project of Lyra2, the comparison project and the proof of concept mobile application are available under the Massachusetts Institute of Technology (MIT) License on GitHub:

https://github.com/all3fox/lyra2-java/tree/2c05da3f6739859f3c3abe5116754666689028a2
(visited on 11/07/2017)
https://github.com/all3fox/lyra2-compare/tree/206851cd2f322e3dabae7439c7f9f75bef81d3a1
(visited on 11/07/2017)
https://github.com/all3fox/lyra2-mobile/tree/88904b25fc5f637f8c3ec14eec4bce88db176b41
(visited on 11/06/2017)

## Keywords

Password Hashing Scheme, Lyra2, Memory-hard Functions, Reproducible Research, Java, Jupyter Notebook, Unit Testing, Android

# Contents

# List of Figures

# List of Tables

# List of Listings

# 1 Introduction

Passwords have been the backbone of user authentication for over 50 years, as stated by Bonneau et al. [27]. Using passwords securely has been shown to be difficult for both the end-users as well as developers. This fact has been demonstrated several times, see for instance Adams and Sasse [2] or Green and Smith [78]. This paper describes three stages during which password usage takes place in some form, namely *authentication*, *communication* and *storage*. The description includes common practices as well as attack vectors.

Password security is crucial throughout all of these stages but the main focus of this work shifts specifically towards password storage. It is common practice to use some kind of a password database during this stage. For example, end-users these days have a choice of several password managers, some of which have been analyzed by Arias-Cabarcos et al. [13]. ~~Of course, they are very different to enterprise level databases but the core principles and goals remain the same: sensitive information should be encrypted with some cryptographic primitive and the secret keys to that primitive should not be stored~~ Most of the time these databases use a password hashing scheme (PHS) so as to avoid storing the master password in cleartext. In order to compute a cryptographically secure hash of the keys, a password hashing scheme (PHS) is often used. If the password database is leaked then the strength of the password hashing algorithm is the last line of defense between the attacker and the the secret keys.

## 1.1 Problem Description

Processing power increases with time while simultaneously getting cheaper, as can be seen for instance in Mack [94]. This works both for the legitimate users as well as the attackers, so PHSs are 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. For a prominent example, consult e.g. Wiemer and Zimmermann [136].

In order to limit the throughput achievable by a potential attacker, new algorithm designs are required. This is the reason why the Password Hashing Competition was announced in 2013 and concluded in 2015. An overview of the proposed algorithms can be found e.g. in Wetzels [135]. The evaluation criteria of the competition stress that proposed candidates should provide minimal speed-up for the highly parallelized hardware. The winner was declared to be Argon2 by Biryukov et al. [25] and special recognition was also given to Catena by Forler et al. [53], Lyra2 by Andrade et al. [6, 87], Makwa by Pornin et al. [113] and yescrypt by Peslyak [110].

## 1.2 Motivation

The theoretical designs and applied implementations for Argon [25], Catena [53], Lyra2 [6, 87], Makwa [113] and yescrypt [110] exist since 2015. However, the adoption of these new cryptographic algorithms could be better. There are many ways to improve current situation. For example, providing better documentation and detailed usage examples would be a solid first step.

As well as that, porting the existing implementations into other programming languages can also lead to an adoption boost.

## 1.3 Contribution

This work describes the porting process of the Lyra2 reference implementation into Java. In order to better distinguish between the two projects, this work refers to the reference implementation as the `lyra2-c` project and the new implementation as the `lyra2-java` project. The new implementation is hosted in the Maven Central repository [93] as well as on GitHub [65]. It is licensed under the MIT License which is a well-known permissive software license. This makes it available for seamless inclusion as a dependency into any other Java project. 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. This paper will highlight the challenges and the necessary implementation details of the `lyra2-java` project in 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. This means that the comparison procedure is also publicly available on GitHub [64]. This is done in order to allow anyone to be able to verify the results presented in this paper.

The final goal is to use the `lyra2-java` project to write an Android application. This should demonstrate that having a pure Java implementation allows for easier adoption of the Lyra2 algorithm on a new platform. Section 5.3 demonstrates a working proof of concept application which produces the same results as the implementation from the `lyra2-c` project.

## 1.4 Outline of the Work

*Chapter 2* is devoted to an overview of related work. It opens with the discussion of secure authentication methods in Section 2.1 and secure communication in Section 2.2. Next, Section 2.3 covers the issue of secure password storage and password hashing. It opens with the history of cryptographic competitions in Section 2.3.1. A description of password hashing fundamentals follows in Section 2.3.2. Finally, Section 2.3.3, Section 2.3.4, Section 2.3.5 and Section 2.3.6 provide an 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 Section 3.1. The different phases that Lyra2 iterates through when performing a hash computation are described there as well. 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 `lyra2-java` project is discussed next in Section 3.3. It first deals with the transition from a function-based to an object-oriented project. The most important classes are shown in the Unified Modeling Language (UML) diagram, see Figure 3.4. Several classes which circumvent specific porting challenges are described. In particular, the endianness issue is shown in Figure 3.5 and a rotation trick for the BLAKE2b and BlaMka sponge classes is shown in Listing 3.1 and Listing 3.2.

*Chapter 4* discusses the choice of a unit testing framework. In particular, Section 4.1 describes the effort it would take to set up Boost.Test for the `lyra2-c` and Argon2 projects, showing possible reasons why those projects use their own solutions instead. Section 4.2 motivates the choice of JUnit as a unit testing framework for the `lyra2-java` project. That section highlights the usefulness of parametrized testing. Finally, Section 4.3 provides reasons why the Python build harness relies on the py.test framework to run parametrized tests as well.

*Chpater 5* summarizes the results. It begins with the demonstration that both the `lyra2-c` and `lyra2-java` projects produce the same hash values given the same inputs, see Section 5.1. First, a manual testing session is recorded in Section 5.1.2. Automated unit testing and continuous integration follow in Section 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, Section 5.2.2 and Section 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 follows in Section 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 relies on the `lyra2-java` project.

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

# 2 Related Work

As mentioned in Chapter 1, this paper describes three stages during which a user's credentials must be protected. In particular, Section 2.1 describes secure authentication, Section 2.2 covers secure communication and Section 2.3 discusses secure password storage. Each section provides an overview of common threats as well as defensive techniques. The chapter culminates with an introduction to memory-hard password hashing algorithms.

## 2.1 Secure Authentication

This section presents a brief overview of different methods for secure authentication and some of the accompanying types of attacks. The presented taxonomy was in part borrowed from Raza et al. [119] and then expanded and filled with details from other sources.

### 2.1.1 Authentication Approaches

Several authentication approaches have emerged over the years. Some of them are quite common and rather old while at the same time others are novel and aim to overcome known issues. Below is a summary of common authentication schemes.

*Conventional Passwords* are by far the most common method of user authentication [27]. The service challenges the user to produce a pair of a (public) username and a (private) password. If the provided pair is correct, the user is allowed to proceed, otherwise the request is rejected. Such an intuitive approach works reasonably well but there are a number of fundamental issues.

According to sources from Ives, Walsh and Schneider [83], a typical user daily accesses roughly 15 services that require a password. At the same time, the user remembers on average 5 unique passwords. This suggests that some of those passwords are reused for several services. Even if attempts are made to slightly tailor the password to the service, this reuse still has a devastating effect on security, as shown by Gaw and Felten [56] or Shay et al. [123]. The most protected account becomes as secure as the least protected one and the possibility of a breach grows with the number of services that share the same (or a similar) password.

Conventional passwords have been shown to have many flaws. To begin with, strong and secure passwords are difficult for humans to remember which is why users often resort to simple and predictable passwords instead. Moreover, an attacker who knows the password is indistinguishable from a legitimate user unless other authentication features are being used. For more conventional password weaknesses please see e.g. Yan et al. [141], Braz and Robert [36] or Jobusch and Oldehoeft [85, 86]. It is not surprising that a number of viable alternatives has been developed over the years in order to address some of the flaws of conventional passwords.

*Graphical Passwords* are one such alternative. They can generally be subdivided into two categories: recognition and recall based, as done by Suo, Zhu and Owen [127]. The *recognition based* graphical passwords vary widely, the main idea being that the user has to recognize correct images and perform an action. For example, clicking on previously chosen pictures or clicking inside a convex hull created by them. Pictures of human faces are sometimes suggested because they are arguably the easiest to remember but other types of imagery are used as well. The *recall based* schemes, on the other hand, expect the user to reproduce some type of a drawing. These include the well-known graphical passwords on smartphones, where the user is typically required to connect points of a $3 \times 3$ grid with straight lines without lifting the finger off the screen. More complex approaches either let the user initiate several touch gestures (thus producing an even more detailed drawing) or write something that resembles a signature.

Some applications of graphical passwords claim better resistance to shoulder surfing attacks [127]. Unfortunately, these schemes usually take more time on average to perform the login when compared to conventional passwords. This adversely affects their practical application.

*Keystroke Dynamics* is an elegant authentication mechanism, described e.g. by Monrose and Rubin [98], which tries to capture the typing pattern of a legitimate user. Several statistics about the speed and rhythm, with which the user presses the keys, are captured: the time between consecutive keys, the time between pressing and releasing the key, the time it takes to type certain words etc. A more extensive list of metrics can be found in Tillmann, de Halleux and Xie [128]. Those statistics are then used in a machine learning algorithm which ultimately decides if the user passes authentication or gets rejected. These algorithms include fine-tuned $k$ Nearest Neighbors, as shown e.g. by Ivannikova, David and Hämäläinen [82], naïve Bayes, as in Ho and Kang [80], gaussian mixture models, as in Deng and Zhong [48], and others.

An immediate challenge is that in this scenario authentication becomes *probabilistic*. Instead of asking "what the user knows", methods based on keystroke dynamics evaluate "what the user is" by judging their ability to type. This is particularly difficult in edge cases when the user is under heavy mental stress or is somehow physically restricted. One possible solution is to perform authentication as a continuous process, as shown in Borus [35] or Serwadda et al. [122]. In such a scenario the user is monitored over a prolonged period of time. Similar challenges also appear with a few other biometric authentication methods, of which keystroke dynamics is a prominent example.

*Biometric Authentication* is a collective term for a technique which focuses on physical traits unique to every user, as explained e.g. by Matyas and Riha [97]. These include fingerprints, facial recognition, voice recognition, iris scanning and similar. Other types of biometric data as well as certain challenges connected with false recognition and false non-recognition can be found in Jain, Ross and Prabhakar [84]. That paper also highlights the notion of *negative recognition* which means that biometric authentication allows to establish if the person is who they *deny* to be. This property distinguishes this class of methods from all the others described in this section.

*One-time Passwords* is another elegant solution for user authentication. As defined by Rayes [118], it is a password that is used exactly once and then immediately discarded. Such a password is usually used in a multi-factor fashion, specifically two-factor authentication (2FA). This means that the user first provides the conventional username/password pair ("what the user knows") and then follows it up with a one-time password. This second password is expected to come from a dedicated smart card, a separate keyfile or a smartphone application ("what the user has"). Of major interest are two one-time password generation algorithms: HMAC-based One-time Password Algorithm (HOTP), specified by M'Raihi et al. [101], and Time-based One-time Password Algorithm (TOTP), also by M'Raihi et al. [102]. They are both designed as part of the Initiative for Open Authentication (OATH). The demand for such algorithms as well as a system that uses conventional smartphones for 2FA was demonstrated e.g. by Aloul, Zahidi and El-Hajj [4].

### 2.1.2 Conventional Password Strength

Previous Section 2.1.1 describes several approaches to user authentication. It mentions that conventional passwords are the most widespread means of user authentication [27]. That section also shows that conventional passwords have their set of problems [36, 85, 86, 141]. Therefore, being able to distinguish between strong and weak passwords is important. Several methods to do so are presented below.

*Information Entropy* is one of the usual ways to assess password strength, consult e.g. Bonneau [26]. It is a base-two logarithm of the number of guesses which are required to recover a password with complete certainty. Information entropy is measured in bits. For example, if the password is completely random and has $x$ bits of information entropy, then an attacker must compute $2^x$ password candidates to be guaranteed a successful recovery.

Human generated passwords, however, are notorious for not being truly random. Which is why computing information entropy often produces inaccurate results [26].

*NIST Special Publication 800-63-2* by Burr et al. [37] tried to address this problem and suggested a refined way of computing information entropy for passwords:

- The entropy of the $1^{\text{st}}$ character is 4 bits.
- The entropy of characters 2 to 7 is 2 bits per character.
- The entropy of characters 9 to 21 is 1.5 bits per character.
- The entropy of characters 21 and above is 1 bit per character.
- If *both* uppercase letters and non-alphabetic characters are used, the password gains 6 more bits of entropy.
- If the password is 1 to 19 letters long and passes an extensive dictionary check, then an additional 6 bits of entropy should be added.

The last bonus is not assigned to passwords of length 20 and above because it is assumed that those are *pass phrases*, i.e. passwords which consist of several dictionary words. This refined approach was still criticized for being inaccurate and was removed from the revised version ~~of the publication~~ 800-63-3 of the NIST Special Publication by Grassi, Garcia and Fenton [77], which deprecated NIST Special Publication 800-63-2.

*Guess Number Calculation* is a different approach to estimating password strength suggested by Kelley et al. [90]. The idea behind this method is to first choose a particular algorithm for password cracking which is known to work well. The number of password candidates which this algorithm needs to try before recovering the target password becomes the indicator of this password's strength. Such an approach was shown to perform reasonably well but it also comes with its own challenges, as pointed out by Weir et al. [134] or Zhang, Monrose and Reiter [142]. First of all, the choice of the algorithm is arbitrary. Secondly, many state of the art password cracking algorithms require a training set to fine tune their parameters. Changing the training set essentially produces a different configuration of the algorithm. This in turn leads to a recalculation of guess numbers. So, this approach in general is mainly empirical and often volatile.

### 2.1.3 Authentication Attacks

Previous sections, namely Section 2.1.1 and Section 2.1.2, describe authentication approaches and password strength estimation. Section 2.1.3 provides a summary of some of the common types of authentication attacks.

*Brute Force* is one of the most common and least sophisticated types of a password attack, see e.g. Narayanan and Shmatikov [105]. The attacker begins by choosing a specific password domain and then launches the trial and error guessing process. For example, one could try all passwords

| Hashtype | Millions of hashes per second |
|---|---:|
| MD5 | 2912.4 |
| SHA1 | 1004.6 |
| SHA-512 | 117.6 |
| SHA-3 (Keccak) | 104.6 |

**Table 2.1:** Hashcat v3.6.0 Running on a Single Nvidia GTX 950M Graphics Card

that are up to 8 characters in length and consist of lowercase Latin letters. This domain contains 217180147158 unique passwords. Assuming that the attacker can try 1000 passwords per second, it will take them $\approx 6.8$ years to complete the search. This might sound like a lot of time but the 1000 passwords per second is a conservative speed estimate. Depending on the available hardware, the hash algorithm and other factors, this speed could be a *a lot* higher. Table 2.1 shows some real world speeds for a few password hashing algorithms as measured by running a common hash computation tool on ordinary consumer hardware available to the author of this paper.

*Dictionary Attacks* are an improvement upon brute force [105]. A dictionary is a collection of possible passwords, often ordered by popularity or likelihood of occurrence. Such dictionaries are compiled from various sources: conventional language dictionaries, website pages, previously leaked passwords and so on. Advanced dictionaries or software that uses them often include *mutations*. A mutation is a modification rule applied to a dictionary entry. An example of a popular modification rule might be a substitution of the letter *o* with the digit 0 or an addition of 42 to the end of the password. Dictionary attacks with modification rules are the most common academic approach to password cracking [90, 123, 134, 142].

Even though dictionary attacks are the standard go-to method for password recovery research, there are several problems with this approach [26]. Rather often it is difficult or outright impossible to accurately compare the results of different studies. For example, the exact password dictionaries may not be available to every researcher because they must be purchased or have changed since the date of the study. The popular tools used for the analysis are also being constantly developed, so keeping track of the exact versions is cumbersome [26].

*Markov Model Based Attacks* are an example of machine learning techniques applied to password recovery, consult e.g. Dell'Amico, Michiardi and Roudier [47]. When constructing the next password candidate, these algorithms use the beginning of the string to determine its end. For example, if the first part of the password candidate is "deter", then the Markov model will try "determine" and "detergent" earlier than "deteraaa" or "deter000". The algorithm uses training data to fit a probability distribution over the domain of possible passwords. Once the distribution is constructed, the most probable password candidates are tried first, which in practice noticeably increases the speed of password recovery. An interesting observation made by the authors of [47] is that the second-best training data is the collection of usernames. The best training set is the passwords themselves. In other words, training a Markov model on the usernames and then using it to recover the passwords works very well. This is an intriguing observation because the password and the username are supposed to serve different purposes. The provided explanation is that the user creates both the password and the username during registration in quick succession. This is why many users probably reuse the same thinking patterns and ideas to come up with these two strings.

*Shoulder Surfing* is an entirely different kind of a password attack [119, 127] . As the name suggests, its simplest form is a person standing next to the user who is performing authentication. However, this is by far not the only scenario. The user might be authenticating from a café, a bank, an airport or any other location with sufficiently advanced video surveillance system. If a high resolution recording of the user's authentication process is captured, then this video becomes a viable attack vector.

*Phishing Attacks* are yet another formidable type of an attack, as summarized by Hong [81] or Wu, Miller and Garfinkel [139]. A legitimate user is usually tricked into visiting a webpage which impersonates a typical webservice, like an e-mail provider or a social network. The user is then prompted to enter their authentication details. If they do not recognize the attack in time then their credentials end up in the hands of the attackers. Phishing attacks usually target a large group of individuals and do not use personalized information. On the other hand, *spear phishing attacks* is a term reserved for targeting specific people (journalists, politicians, high-profile executives, etc.) [81]. These attacks are a lot harder to defend from because the information is tailored specifically to the target.

## 2.2 Secure Communication

Previous Section 2.1 is devoted to the authentication stage. Current Section 2.2 discusses the next stage, namely communication. In particular, the question of establishing a secure communication channel over an insecure connection is addressed.

### 2.2.1 Secure Sockets Layer and Transport Layer Security

*Secure Sockets Layer (SSL)*, as ~~standardized~~ described by Freier, Karlton and Kocher [54], and *Transport Layer Security (TLS)*, as formalized by Dierks and Rescorla [49], are a ~~pair~~ stack of protocols widely used to ensure secure communication. TLS first appeared in 1999 and is the successor to SSL. The TLS protocol guarantees that the connection between communicating parties is *secure* (i.e. a passive eavesdropper will not obtain information) and *reliable* (i.e. an active attempt to tamper with the connection will be detected). The authentication of communicating parties can occur with the help of *public key cryptography*. With a bit of additional configuration *forward secrecy* can be achieved as well, as shown by Adrian et al. [3]. Forward secrecy means that if at some later point the service is compromised and its private keys are leaked, the contents of the past communication sessions will not be affected.

TLS is a complex algorithm which allows *a lot* of flexibility when it comes to parameter choice. It begins with a TLS *handshake*:

1. The client presents the list of supported ciphers and hash functions to the server.
2. The server chooses a cipher and hash function and informs the client.
3. Usually, the server also provides identity information by means of a digital certificate.
4. The client decides if they want to proceed based on identity information (or lack thereof).
5. The client and the server generate a unique session key: if perfect forward secrecy is required, then a variation of a Diffie-Hellman key exchange takes place. Otherwise, the client generates a random number and encrypts it with the server's public key. Once passed to the server, it will be decrypted with the server's private key.

During the handshake different algorithms can be negotiated: Rivest-Shamir-Adleman (RSA), Diffie-Hellman, ephemeral Diffie-Hellman, Elliptic Curve Diffie-Hellman, ephemeral Elliptic Curve Diffie-Hellman, and a few others. The digital certificate relies on the *public key infrastructure (PKI)* which consists of users as well as registration and certificate authorities (RAs and CAs respectively). PKI in general receives a lot of criticism:

- Obtaining a digital certificate costs both time and money.
- CAs are being constantly breached which enables attackers to issue fake certificates: the 2011 Comodo [42] and DigiNotar [71] incidents, the 2013 ANSSI [70] and TURKTRUST [73] incidents, the 2014 National Informatics Center of India [76] incident, the 2015 & 2016 loss of trust in Symantec [72, 100], 2016 & 2017 loss of trust in WoSign and StartCom [74, 75].
- Protocol versions are known to have been deprecated due to design flaws and security vulnerabilities. SSL 2.0 was formally deprecated, as described by Polk and Turner [112]~~,~~; SSL 3.0 ~~—~~ – by Barnes et al. [18]. TLS 1.2 is the current latest version of the protocol.
- Libraries that implement the protocol have had major security issues. One of the most well-known examples is *Heartbleed*, as described by Durumeric et al. [50]. It was a major vulnerability that existed between 2012 and 2014 in a widely used OpenSSL library. The bug allowed anyone access to arbitrary data as a result of a buffer overflow.

The list of incidents and issues is by no means complete. However, SSL and TLS are an ubiquitous pair of protocols which have been in use for a long time. Therefore, it should not come as a surprise that a few design and usage problems have surfaced over the years. The lack of trust in certificate authorities remains a cornerstone design issue though. This lead to the development of a number of alternative secure communication methods that do not rely on the current PKI. One such family of methods is described next in Section 2.2.2.

### 2.2.2 Password Authenticated Key Exchange

*Password Authenticated Key Exchange (PAKE)* is a family of methods that establish secure communications over an insecure channel without the need for a PKI. Insight into a select couple of protocols is provided below. In each case the described protocols provide the following guarantees:

1. Resistance to offline dictionary attacks.
2. Resistance to online dictionary attacks: only one password attempt per session.
3. Forward secrecy: past and current session keys remain secure if the password is compromised at some later point in time.
4. Known session key security: past and future sessions remain secure even if the current session key is compromised.

*Encrypted Key Exchange (EKE)*, pioneered by Bellovin and Merritt [20, 21], was the first method from the PAKE family. Its early version consists of the following steps (where Alice and Bob are the names of participating parties):

1. Alice and Bob both know a (weak) conventional password $P$.
2. Alice begins by generating a random public/private pair of keys $E_A$ and $D_A$. She uses $P$ to encrypt the public key: $M_0 = P(E_A)$. She sends $M_0$ to Bob.
3. Bob uses $P$ to decrypt the public key of Alice: $P^{-1}(P(E_A)) = E_A$. He then generates a random secret key $R$ and encrypts it first with the asymmetric and then with the symmetric keys: $M_1 = P(E_A(R))$. Then he sends $M_1$ to Alice.

4. Alice uses both $P$ and $D_A$ to recover $R$: $D_A(P^{-1}(P(E_A(R))) = R$.

Once this initial exchange is over, both Alice and Bob know $R$ and $E_A$, which are presumed to be stronger than the initial (weak) conventional password $P$. Better strength is presumed because both $R$ and $E_A$ are supposed to have been generated randomly over a large possible keyspace. If the attacker controls the communication channel, then they know $M_0 = P(E_A)$, $M_1 = P(E_A(R))$ and $R(\texttt{Known message})$. To try some candidate password $P'$, they must produce a candidate $E'_A = P'^{-1}(P(E_A))$ and then decide if there exists a key $R'$ such that $E'_A(R') = E_A(R)$ and $R'^{-1}(R(\texttt{Known message}))$ recovers the known message. This is expected to be far more expensive than a plain attack on $P$.

The straightforward implementation of EKE was shown to be insecure, you could find a summary of theoretical and practical weaknesses in Hao and Ryan [79]. In addition to that it is patent-encumbered. Therefore, alternative protocols have been developed.

*Password Authenticated Key Exchange by Juggling (J-PAKE)* is a more recent and advanced method from the PAKE family [79]. It has a formal security proof provided by Abdalla, Benhamouda and MacKenzie [1]. Its computation cost is similar when compared to EKE. Moreover, it is not patented and has been included into such cryptographic libraries as OpenSSL and Bouncy Castle.

## 2.3 Secure Password Storage

Section 2.1 and Section 2.2 discussed methods for secure authentication and communication. The rest of this work focuses on the problem of secure password storage. This issue is as important as the ones discussed above because password databases are routinely compromised [26, 47, 83]. As a result, protection against offline dictionary attacks is of utmost importance.

### 2.3.1 Cryptographic Competitions

It is common practice to announce a competition in order to develop standard cryptographic primitives. For example, the symmetric block cipher Rijndael by Daemen and Rijmen [45] was chosen to become the Advanced Encryption Standard (AES) [124] in a competitive selection process [9, 125] 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 [44], Serpent by Biham, Anderson and Knudsen [5], Twofish by Schneier et al. [121], RC6 by Rivest et al. [120] and MARS by Burwick et al. [38]. Another competition was held by NIST between 2007 and 2012 [126] in order to select the next hash function standard, Secure Hash Algorithm 3 (SHA-3). The final round included 5 designs: BLAKE by Aumasson et al. [16], Grøstl by Gauravaram et al. [55], JH by Wu [138], Skein by Ferguson et al. [52] and Keccak by Bertoni et al. [24], the last of which ultimately became the winner of the competition.

Choosing cryptographic primitives through an open competitive process is therefore a common approach. So, when the need for an updated, memory-hard PHS 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 (visited on 10/28/2017) website lists two well-known cryptographers Khovratovich and Aumasson as direct contacts. The Password Hashing Competition concluded in 2015, with Argon2 [25] selected as its winner and Catena [53], Lyra2 [6], Makwa [113] and yescrypt [110] receiving special recognition.

### 2.3.2 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, consult e.g. Simplício et al. [87].

Password hashing is common practice when user authentication is required [87]. 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 the Request for Comments (RFC) specification by Moriarty, Kaliski and Rusch [99]. 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 the number of repetitions as a parameter for computational cost.

One of the early kinds of attacks against password hashes were *rainbow tables*, as discussed by Avoine, Junod and Oechslin [17]. They are a classic example of a *time-memory trade-off (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. If two hashes match, the recovery of the original password becomes a matter of a simple lookup.

The currently well-known defense against the rainbow table attack is to use a sufficiently large *salt* when computing the password [87]. This salt is a randomly generated value which is unique for every password and is openly stored alongside it in the defender's database. The salt 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, as shown by Murakami, Kasahara and Saito [103]. If so, the increased individual computational time cost of one hash does not prevent the attacker from trying many candidates at once. The throughput of the attack (average attempted passwords per second) remains high. This type of attacks is addressed by memory-hard password hashing 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 than CPUs of a personal computer. Therefore, if a password hashing process offers a memory cost parameter, an attacker will need (much) more memory per each specialized processing unit in order to have the same throughput. This is supposed to make the attack much more expensive and slow it down considerably.

Arguably the first PHS to implement this idea was scrypt by Percival [108], which was recently published as RFC 7914 by Percival and Josefsson [109]. However, this scheme is sometimes criticized for being overly complicated and not offering a decoupled way to control time and memory costs [87]. In other words, there is a single parameter that controls both those values.

In order to create new designs which would address the need for a new memory-hard password hashing function, the Password Hashing Competition was announced in 2013 and concluded in 2015 [135]. Below follows a quick summary of its winner (Argon2 [25]) and all of the finalists except Lyra2 [6]. The latter will be described in more detail later in Chapter 3.

### 2.3.3 Main Features of Catena

Catena [53] is a password-scrambling framework based on bit reversal graphs. One of its prominent features is *client-independent updates*. It allows a hash value to be updated with larger time or memory cost values 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 by Aumasson et al. [15].

An interesting related project is Catena-Axungia [61]. 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 parameter values for the current machine. These values 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 [62] project is a modular C++ implementation of Catena. It is reported to be somewhat 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.

### 2.3.4 Main Features of Makwa

The Makwa PHS [113] at its core relies on Blum integers. A Blum integer is a natural integer $n$ which can be represented as a product $pq$ (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 core idea of Makwa is to square the (derivative hash of) the password (together with the salt and other parameters) many times modulo a Blum integer. This 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 distinguishing features of Makwa highlighted on its website is *delegation* [96]. 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 [113].

### 2.3.5 Main Features of yescrypt

The yescrypt PHS [110] improves upon its predecessor, scrypt [108]. 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 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.

### 2.3.6 Main Features of Argon2

Argon2 [25] has two distinct configurations: *Argon2i* and *Argon2d*. The former revisits 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. A side-channel attack is described by Caddy [39] as a type of attack which exploits information relevant to the implementation of a cryptographic algorithm as well as its mathematical properties. At the same time Argon2d is more resistant to *time-memory trade-offs* 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 parallelism, length of the produced hash (called a *tag* [25]), 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 the secret value together with associated data. The last of these three adds more flexibility to the scheme. The secret value parameter enables *keyed hashing* [69] 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 algorithm among all of the finalists. Consequently, the `README.md` file in the GitHub repository [58] 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 [11, 59].

## 2.4 Software Compatibility

This paper deals with porting an algorithm from one programming language to another. Therefore the question of compatibility of the two resulting implementations is going to be addressed next.

One possible approach to ensure that the two programs work the same way is to utilize formal specification and verification methods, as shown by van Lamsweerde [91] or Müller and Poetzsch-Heffter [104]. In particular, this requires building (or reusing) some kind of a theoretical model and performing rigorous analysis of the source code. In case the source code ever changes, the analysis usually has to be repeated as well. This is a manual and labor intensive process which the author of this paper would like to avoid.

Instead, the `lyra2-java` project relies on a more practical verification approach, namely unit testing. Of course, unit testing is not a strict and formal compatibility and correctness guarantee. Nonetheless it has been shown useful for building confidence in the written code by Williams and Kudrjavets [137]. Unit testing comes with its own classical questions: which tools and frameworks to use, as discussed by Daka and Fraser [46], and how to know that enough effort has been spent on writing tests, as demonstrated by Elberzhager et al. [51]. This section discusses the basic terms, definitions and common practices connected with unit testing. Chapter 4 is devoted to the choice of particular unit testing tools for the `lyra2-java` project and the Python build harness.

*Unit testing* is a testing process during which it is verified that parts of the program function as expected when provided with specific data, consult e.g. Huizinga and Kolawa [43] or Cheon and Leavens [40]. *Unit test* can be understood as the testing logic together with the associated data that test some small part of the program [43]. 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 (TDD)* is a software development practice which expects a piece of functionality to be produced together with the accompanying tests, as described e.g. by Astels [14] or Beck [19].

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 [19]. It is also expected that distinct unit tests are usually independent of each other and can be run in any order or in parallel [19]. 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 high. For example, when there is a strong dependency on the data coming from a database, a unit test needs to *mock* the database connection and the data [19]. *Mocking* is the process of simulating a real object with a simplified version of it, see e.g. Mackinnon, Freeman and Craig [95].

It is often the case that many unit tests share the same logic. Specifically in the context of 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 labor, as shown by Tillmann, de Halleux and Xie [131]. 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 [51] but 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 [14]. For simplicity, the `lyra2-java` project uses just the LOC coverage metric.

*Continuous integration (CI)* is the practice of running unit tests and measuring the changes in code coverage with every update of the source code. This approach helps identify problems early and fix them quicker [137]. Continuous integration often occurs transparently on a separate set of dedicated machines and its status is visible to the developers at all times. Its popularity and utility is indisputable, with such companies like TravisCI [132], AppVeyor [8] and CircleCI [41] providing both commercial and free (for open source projects) continuous integration as a service. Continuous integration for the `lyra2-java` project is set up on TravisCI [92].

# 3 Lyra2: a Sponge-based PHS

This chapter focuses on Lyra2 and its implementation. The cryptographic foundations are outlined in Section 3.1. Section 3.2 describes the architecture of the `lyra2-c` reference implementation project. Finally, Section 3.3 covers the architecture of the ported `lyra2-java` project as well as a number of challenges that had to be solved.

## 3.1 Theoretical Background

At the heart of Lyra2 is a *sponge function* or a *sponge construction*. This is an algorithm with finite internal state which takes an input bit stream of arbitrary length and produces an output bit stream of arbitrary length as well, see e.g. Bertoni et al. [22]. 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*. There is a *fixed-width permutation* $f$ inside of every sponge function. It takes $w$ bits as input and produces $w$ bits as output. A common example of $f$ is BLAKE2b [15], 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. Next, 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

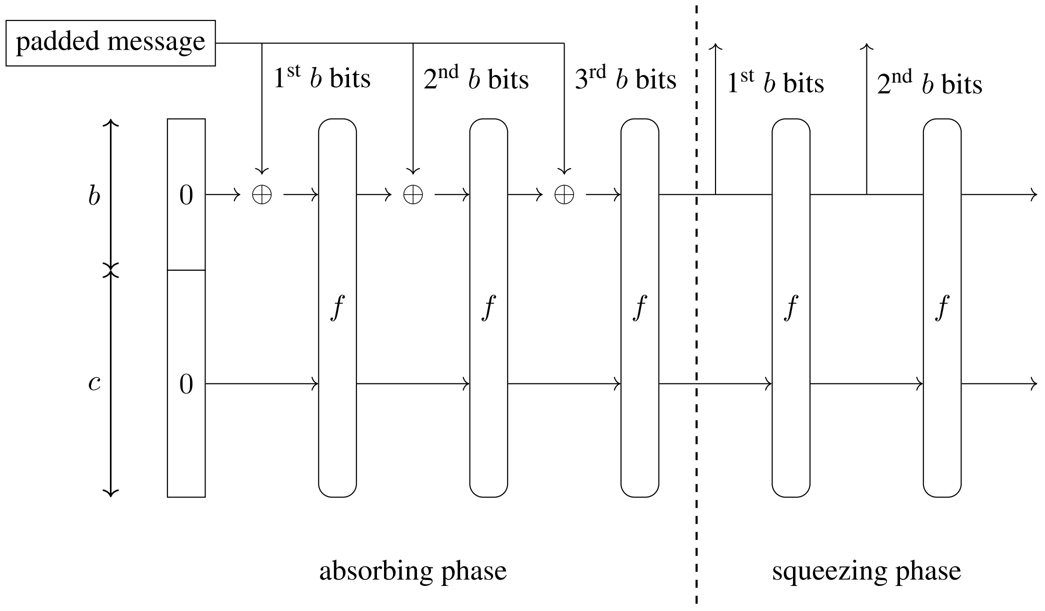


**Figure 3.1:** Sponge Construction: adapted from [23, 87].

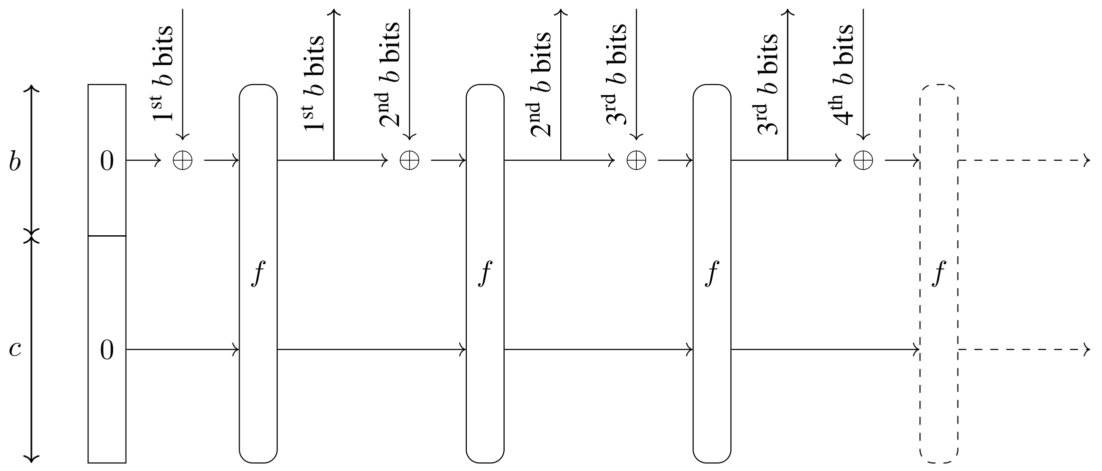


**Figure 3.2:** Duplex Construction: adapted from [23, 87].

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 the work by Bertoni et al. [23] and a visual explanation is provided in Figure 3.2. An important detail is that [23] proves that the sponge and the duplex construction are equally secure.

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 `--columns` and `--blocks` in the `lyra2-java` 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 an application of a duplex construction which continuously reads and writes blocks to and from the memory matrix. The algorithm is structured into four sequential phases: *Bootstrapping*, *Setup*, *Wandering*, and *Wrap-up*, see Figure 3.3 for reference.

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 initialization vectors are used instead of zeroes to initialize part of the sponge state.

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 and 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 trade-offs that would either allow to parallelize computations or 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 [6, 87].

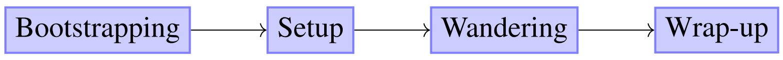


**Figure 3.3:** Strictly Sequential Design of Lyra2

## 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 [67]. 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 examines how Lyra2 withstands GPU-based attacks and `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 [67] was forked to the public repository of the author of this paper [68] 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 convenient comparison between the `lyra2-c` and the `lyra2-java` projects:

- 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 [68] and a pull request [57] 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 the py.test framework then a call to the `py.test` script 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 of 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 `lyra2-java` project underwent iterative improvements. A simplified UML diagram of the final version is presented in Figure 3.4.

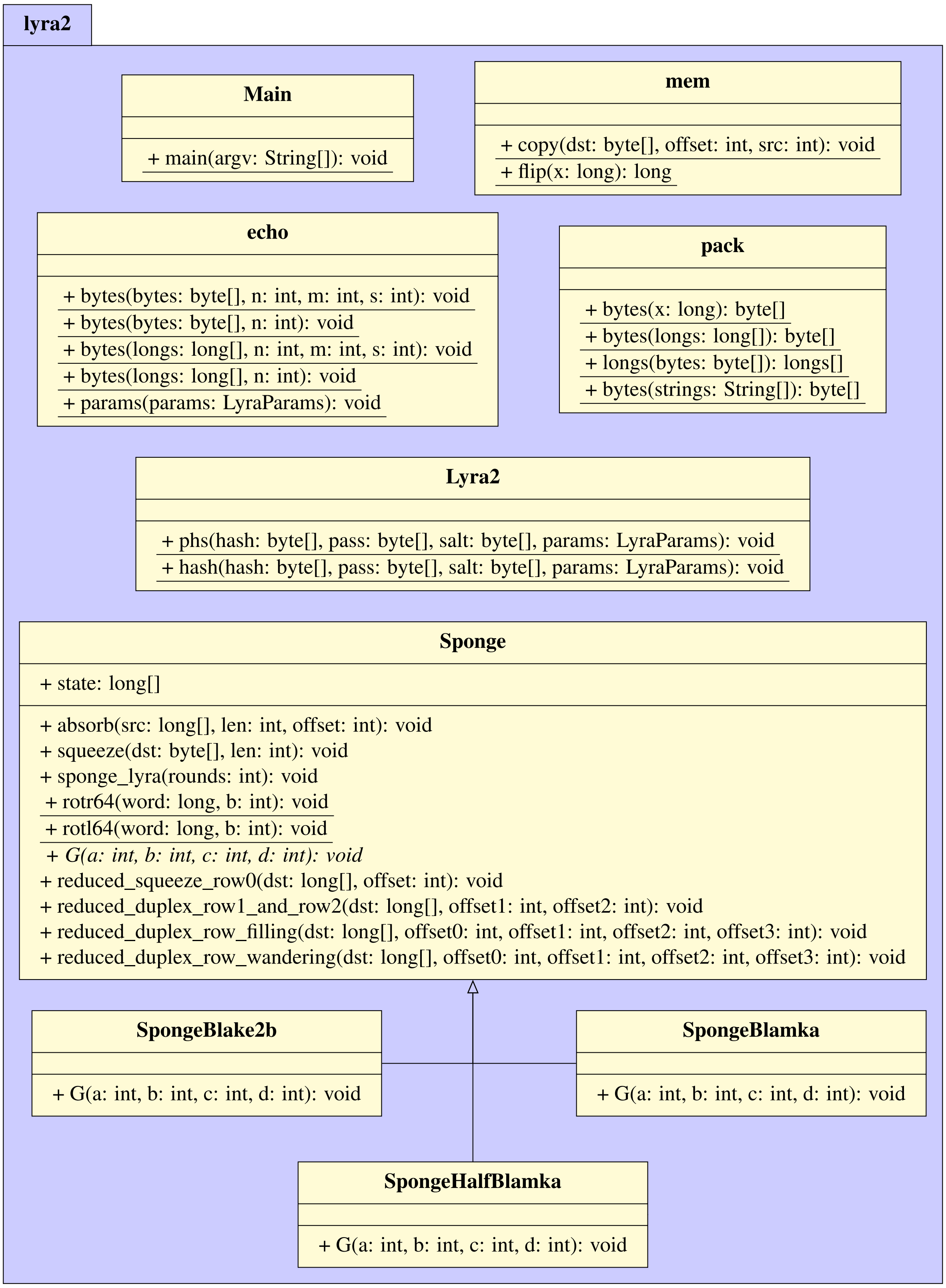

lyra2
Main
+ main(argv: String[]): void
mem
+ copy(dst: byte[], offset: int, src: int): void
+ flip(x: long): long
echo
+ bytes(bytes: byte[], n: int, m: int, s: int): void
+ bytes(bytes: byte[], n: int): void
+ bytes(longs: long[], n: int, m: int, s: int): void
+ bytes(longs: long[], n: int): void
+ params(params: LyraParams): void
pack
+ bytes(x: long): byte[]
+ bytes(longs: long[]): byte[]
+ longs(bytes: byte[]): longs[]
+ bytes(strings: String[]): byte[]
Lyra2
+ phs(hash: byte[], pass: byte[], salt: byte[], params: LyraParams): void
+ hash(hash: byte[], pass: byte[], salt: byte[], params: LyraParams): void
Sponge
+ state: long[]
+ absorb(src: long[], len: int, offset: int): void
+ squeeze(dst: byte[], len: int): void
+ sponge_lyra(rounds: int): void
+ rotr64(word: long, b: int): void
+ rotl64(word: long, b: int): void
+ G(a: int, b: int, c: int, d: int): void
+ reduced_squeeze_row0(dst: long[], offset: int): void
+ reduced_duplex_row1_and_row2(dst: long[], offset1: int, offset2: int): void
+ reduced_duplex_row_filling(dst: long[], offset0: int, offset1: int, offset2: int, offset3: int): void
+ reduced_duplex_row_wandering(dst: long[], offset0: int, offset1: int, offset2: int, offset3: int): void
SpongeBlake2b
+ G(a: int, b: int, c: int, d: int): void
SpongeBlamka
+ G(a: int, b: int, c: int, d: int): void
SpongeHalfBlamka
+ G(a: int, b: int, c: int, d: int): void

**Figure 3.4:** Simplified Class Diagram for the `lyra2-java` Project

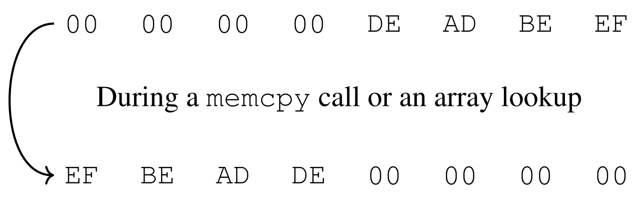


**Figure 3.5:** Byte Swap in the Little-endian Case

The `Main` class is the entry point of the project. An implementation detail that is not shown in Figure 3.4 is that the `Main` class relies on a 3rd party command line library `Picocli` [111]. 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 3.4, it just stores the parameters and constants relevant to the operation of Lyra2. 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 a collection of related static methods.

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 languages 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 3.5. 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 in the `lyra2-java` project.

The `pack` class provides methods that allow to partially emulate the pointer casting which happens in C. It is often the case in the `lyra2-c` project that a chunk of memory is allocated and treated as individual bytes or some 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 because the perceived size of the underlying element changes event though the bytes themselves remain the same. In other 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. This copy does not happen during *Setup* and *Wandering* phases.

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 interesting detail is that the `Sponge` class in the `lyra2-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 rotate it by the specified number of bits `b`. Those rotations are *not* arithmetic and do *not* preserve the sign of their argument.

The `lyra2-c` project requires just one type of rotations. On the other hand, both types of rotations are helpful in the `lyra2-java` project. Using both left and right directions of rotation allows to avoid a number of byte rotations related to endianness. For an example of this, refer to Listing 3.1. 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. The `lyra2-java` project would then have to first call a `mem.flip` and then follow it up with a call

```
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
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 }
```

**Listing 3.1:** BLAKE2b as Instance of the `Sponge` Class

to `Sponge.rotr64`, essentially performing rotations in opposite directions. Instead, a single call to `Sponge.rotl64` takes place which performs a right rotation.

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, see Listing 3.2.

```java
public class SpongeBlamka extends Sponge {
    private long fBlaMka(final long x, final long y) {
        long lessX = 0x00000000FFFFFFFFL & x;
        long lessY = 0x00000000FFFFFFFFL & y;

        lessX *= lessY;

        lessX <<= 1;

        return lessX + x + y;
    }

    @Override
    public void G(final int a, final int b, final int c, final int d) {
        state[a] = mem.flip(fBlaMka(mem.flip(state[a]), mem.flip(state[b])));
        state[d] = rotl64(state[d] ^ state[a], 32);

        state[c] = mem.flip(fBlaMka(mem.flip(state[c]), mem.flip(state[d])));
        state[b] = rotl64(state[b] ^ state[c], 24);

        state[a] = mem.flip(fBlaMka(mem.flip(state[a]), mem.flip(state[b])));
        state[d] = rotl64(state[d] ^ state[a], 16);

        state[c] = mem.flip(fBlaMka(mem.flip(state[c]), mem.flip(state[d])));
        // Cannot use the left rotation trick here: 63
        // individual bytes do not stay the same, they change too.
        state[b] = mem.flip(rotr64(mem.flip(state[b] ^ state[c]), 63));
    }
}
```

**Listing 3.2:** BlaMka as Instance of the `Sponge` Class

# 4 Unit Testing Framework Choice

Chapter 4 discusses the choice of particular unit testing frameworks for the `lyra2-java` project. For an introduction to unit testing terms and basic practices please consult Section 2.4. In this chapter, however, Section 4.1 discusses how two similar projects, the `lyra2-c` project and Argon2, address testing. Section 4.2 and Section 4.3 motivate the choice of tools and techniques for the `lyra2-java` project and the Python build harness.

Although the `lyra2-c` project has automated tests, it does not use a dedicated unit testing framework [67]. This is a justifiable choice which has its advantages. In particular, it allows to keep the complexity of the project under control 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 [12]. Section 4.1 will provide a plausible explanation for this decision.

## 4.1 Boost.Test

There is a large choice of unit testing frameworks for the C/C++ ecosystem [33, 60, 63]. 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 avoid using a unit testing framework altogether. Instead, they often write a separate test file that runs a few sanity checks.

For example, Argon2 uses its own `src/test.c` test program which was introduced on 25th of January 2016 [58]. The commit hash starts with `7450df88` and it is number 317 out of (current) almost 600 in the version control history. This shows that this testing was introduced at the later stages of the project when the need for it was apparent [10].

A similar story could be observed for the reference Lyra2 project [67]. The rest of this section will attempt to demonstrate why a rather popular unit testing library Boost.Test was not used in either of the projects. This particular library was tried because of the personal developer preference of the author of this paper. A different developer could attempt the same steps with another library, like Catch [60] or Google Test [63].

Boost itself is a large collection of different libraries for C++ [28]. It was originally founded by Dawes and 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 for any library that would like to be included into the collection [29] as well as rigorous review in the mailing lists [30].

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

**Table 4.1:** Important Features of Several Unit Testing Frameworks

```
#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);
}
```

**Listing 4.1:** Automatic Unit Test Registration With Boost.Test

The Boost.Test is a unit testing framework which is part of Boost [33]. It is licensed under the Boost Software License which is a free software license, approved by the Open Source Initiative and compatible with the General Public License (GPL). The library documentation can be found online [33].

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 2.4. Boost.Test provides parametrization as well as other more common features of a unit testing library. In particular, 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 Listing 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 the documentation [31]. First step is to declare such tests with the special `BOOST_DATA_TEST_CASE` macro. This macro informs the test runner that the corresponding test case requires test data in order to function. The data must be wrapped into a dataset in order to be passed into the test case, as described in the documentation [32].

Password hashing algorithms often require several parameters in order to run, i.e. at least a password and a salt. In the case of Boost.Test it means that datasets must be somehow combined and manipulated together. For that reason operations such as *joins*, *zips* and *grids* are supported by the dataset wrapper [34].

In conclusion, Boost.Test requires considerable setup in order to provide parametrized testing for password hashing algorithms. Although the documentation offers an accurate description of the process, it still takes significant effort and time to configure the test harness. The `lyra2-c` project and Argon2 understandably avoid these development costs by using their own unit testing programs.

## 4.2 JUnit and TestNG

The Java ecosystem offers a unit testing framework called JUnit [88]. The basic components of this framework are 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 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.

In addition to JUnit, the Java ecosystem has a unit testing framework called TestNG [129]. Below is a short comparison of the functionality relevant to testing password hashing algorithms.

Firstly, both frameworks offer parametrization tests but the implementation details are different. In order for a class to implement parametrized testing with JUnit4, it must be marked with the `@RunWith` annotation. That annotation must specify `Parameterized.class` as the test runner for the class. As well as that, the class must provide a method marked with the `@Parameters` annotation. This method should return a single `Collections<Object[]>` collection of test data [89]. On the other hand, TestNG has two distinct mechanisms, one of which uses the `testng.xml` file and the other relies on a method with the `@DataProvider` annotation. In the second case the method should return either a single `Object[][]` array of test data or an `Iterator<Object[]>` iterator over such an array. With the first mechanism, the test data is stored directly in the `testng.xml` file. So, only the second approach allows to generate test data while the tests are being run [130].

Based on the comparison above, the `lyra2-java` project was developed with JUnit4. There is nothing that would prohibit the usage of the TestNG framework for the same task. However, JUnit4 offers a single approach to writing parametrized unit tests which is flexible enough and does not make the developer choose between the different methods and different return values, as TestNG does. Finally, JUnit4 is often provided by default in modern Java development environments and does not require additional setup. Such availability has also played a major part in the choice of the unit testing framework for the `lyra2-java` project.

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

The `setupClass` method is marked with the `@Parametrized.Parameters` annotation which makes this method the provider of data. The precomputed data comes from several YAML Ain't Markup Language (YAML) data files whose names are stored in the `fnames` variable. More information about the YAML format in general and the test data file structure in particular can be found in Section 4.4. The `for` loop on line 12 sequentially opens the YAML data files, loads their contents and collects them in the `entries` collection 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 set of test data from the `entries` collection.

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

## 4.3 Unit Testing With py.test

Python is a popular high-level general purpose scripting language created by van Rossum [117]. It has a built-in unit testing framework called unittest [133] which comes together with the interpreter. Similar to JUnit4, it expects the developer to provide a hierachy of test classes grouped into test modules. Other two notable unit testing frameworks in the Python ecosystem are Nose [106] and py.test [115]. They are both similar to the built-in unittest framework but at the same time of-

```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);
    }
}
```

**Listing 4.2:** Example of JUnit4 Parametrized Testing for the `lyra2-java` Project

```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
```

**Listing 4.3:** Simplified Example of py.test Parametrized Testing for Python Build Harness

fer more flexibility: the developer does not *have to* organize their tests into classes, parametrized testing is readily available and parallelized test execution can be taken advantage of as well.

The design of the unittest framework does not offer a dedicated mechanism to write parametrized tests. At the same time the Nose library implements parametrized testing by making use of Python generators (the `yield` keyword). They generate separate functions at runtime which are then executed as standalone test cases [107]. On the other hand, the py.test framework does not use Python generators but instead relies on the `@mark.parametrized` decorator [116]. A usage example can be found in Listing 4.3.

Unfortunately, the Nose unit testing framework is currently 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 `lyra2-c` project uses the py.test framework.

Listing 4.3 shows why parametrized testing is important when building different configurations of the reference Lyra2 implementation. Before running those configurations it is reasonable to make sure that the compiled executable files actually work. However, since there are many parameters that can be configured at compile time (such as the number of columns or the size of the block of the memory matrix), writing a test for each individual possible configuration is not feasible. Instead, each generated executable file is prefixed with `lyra2-` and has other compile time parameters listed in the filename, like the aforementioned number of columns or size of the block. Then the `@pytest.mark.parametrize('path')` decorator is used to fetch all the matching filenames from the build directory. Each of these names is later used as the `path` parameter to create the corresponding unit test case dynamically.

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.

## 4.4 Configuration and Test File Format

The data for parametrized tests is stored in YAML files. Those files are produced by the `lyra2-c` project with the help of the Python build harness. Then they are consumed by the JUnit testing framework of the `lyra2-java` project. This mechanism allows to make sure that both projects compute the same hash values.

YAML stands for *YAML Ain't Markup Language* and is a superset of JavaScript Object Notation (JSON), another popular data format. YAML targets human readability and provides native support for custom datatypes: scalars, arrays, structures, etc. [140]. Most programming languages provide support for reading and writing of YAML files, including both Java and Python.

A single file in YAML can hold several documents. This feature is leveraged by the Python build harness. In particular, each YAML file corresponds to a single Lyra2 executable. This guarantees that exactly one Lyra2 executable is used to compute the test data hashes from that file. So, the compile time parameters are the same for the contents from that file. These parameters are stored together with all the runtime parameters and the resulting hash, see Listing 4.4 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.

The YAML format is also used to store default configuration for the Python build harness of the reference Lyra2 project. It can be found in the `harness` branch of the forked reference repository in the `harness.yml` file [68]. The primary parameters are `build_path` and `makefile_path`. The first one defines the location for the compiled Lyra2 executable files 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 lengths (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 grid of all of the possible value combinations and then runs every compiled executable using those values as test data. The resulting hashes 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 must remove them manually if regeneration is required.

Finally, compilation flags are also part of the configuration and their defaults can be seen in Listing 4.5.

```
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
```

**Listing 4.4:** Example of a YAML Test Data File: distinct documents are separated with `---`.

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

**Listing 4.5:** Compilation Flags Used by the `lyra2-c` Project

# 5 Results

This chapter presents the main results of the porting effort. In particular, Section 5.1 covers algorithm-level compatibility and explains the choice of compared configurations. Section 5.2 compares performance of the `lyra2-c` and `lyra2-java` single-threaded implementations on a number of different configurations of the Lyra2 password hashing scheme. Finally, Section 5.3 demonstrates the ease of integration of the ported project into an Android mobile application.

## 5.1 Algorithm-level Compatibility

The primary goal of this work was to port Lyra2 to Java in such a way that the `lyra2-java` project would produce the same hash values as the reference `lyra2-c` project when provided 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 the reasoning behind the choice of 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-c` project contains a `Lyra2/src/runBenchCPU.sh` benchmarking script which suggests 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.

| 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:** 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
```

**Listing 5.1:** Manual Testing Protocol of the `lyra2-java` Project

### 5.1.2 Manual Testing

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

### 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 in the forked GitHub project [68] and the corresponding pull request [57]. 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 test the implementation. Some of the hash values were also included into the continuous integration service and their status can be verified by visiting TravisCI [92]. However, continuous integration runs only a subset of the vectors that were run locally due to the limited amount of memory available on the remote machines. 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 [64]. 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 [64] is a combination of two Jupyter Notebooks [114] and an SQlite database. The two created notebooks follow the producer-consumer strategy. The producer notebook is `src/compare.ipynb` and it is responsible for compiling both C and Java implementations. It is then used to asynchronously dispatch computation tasks. The results are 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. Section 5.2.2 and Section 5.2.3 show how several Lyra2 configurations behave when time cost and 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, Figure 5.1, Figure 5.3 and Figure 5.5
- BLAKE2b sponge with 2048 columns, Figure 5.2, Figure 5.4 and Figure 5.6
- BlaMka sponge with 256 columns, Figure 5.1, Figure 5.3 and Figure 5.7
- BlaMka sponge with 2048 columns, Figure 5.2, Figure 5.4 and Figure 5.8

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 [64].

### 5.2.2 Fixed Time Cost

There are two figures that show how running time and consumed memory depend on memory cost when time cost is fixed. Figure 5.1 shows a 256-column and Figure 5.2 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.

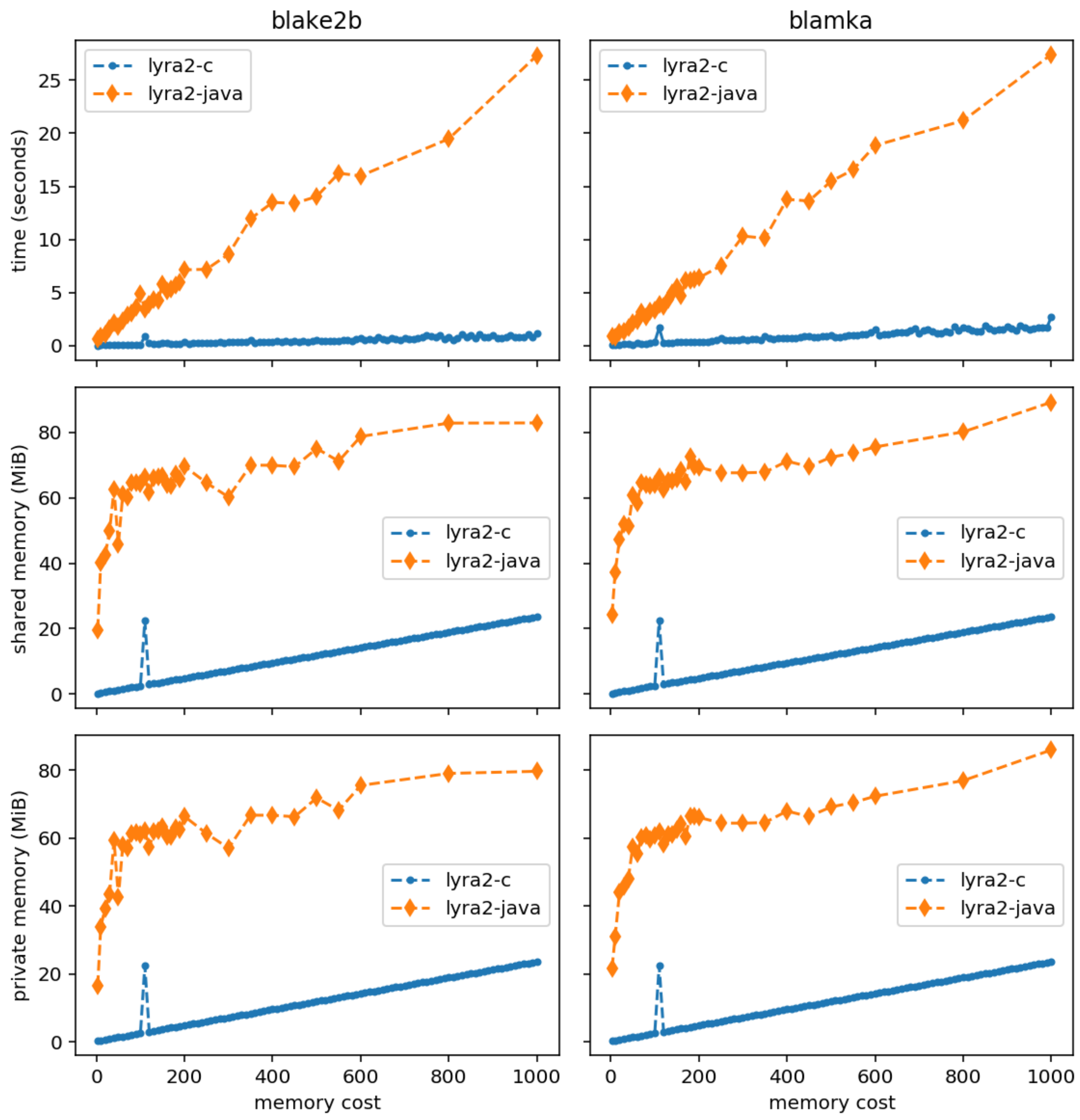


**Figure 5.1:** Comparison of `lyra2-c` and `lyra2-java`: 256 columns, fixed time cost of 10. BLAKE2b sponge on the left, BlaMka sponge on the right.

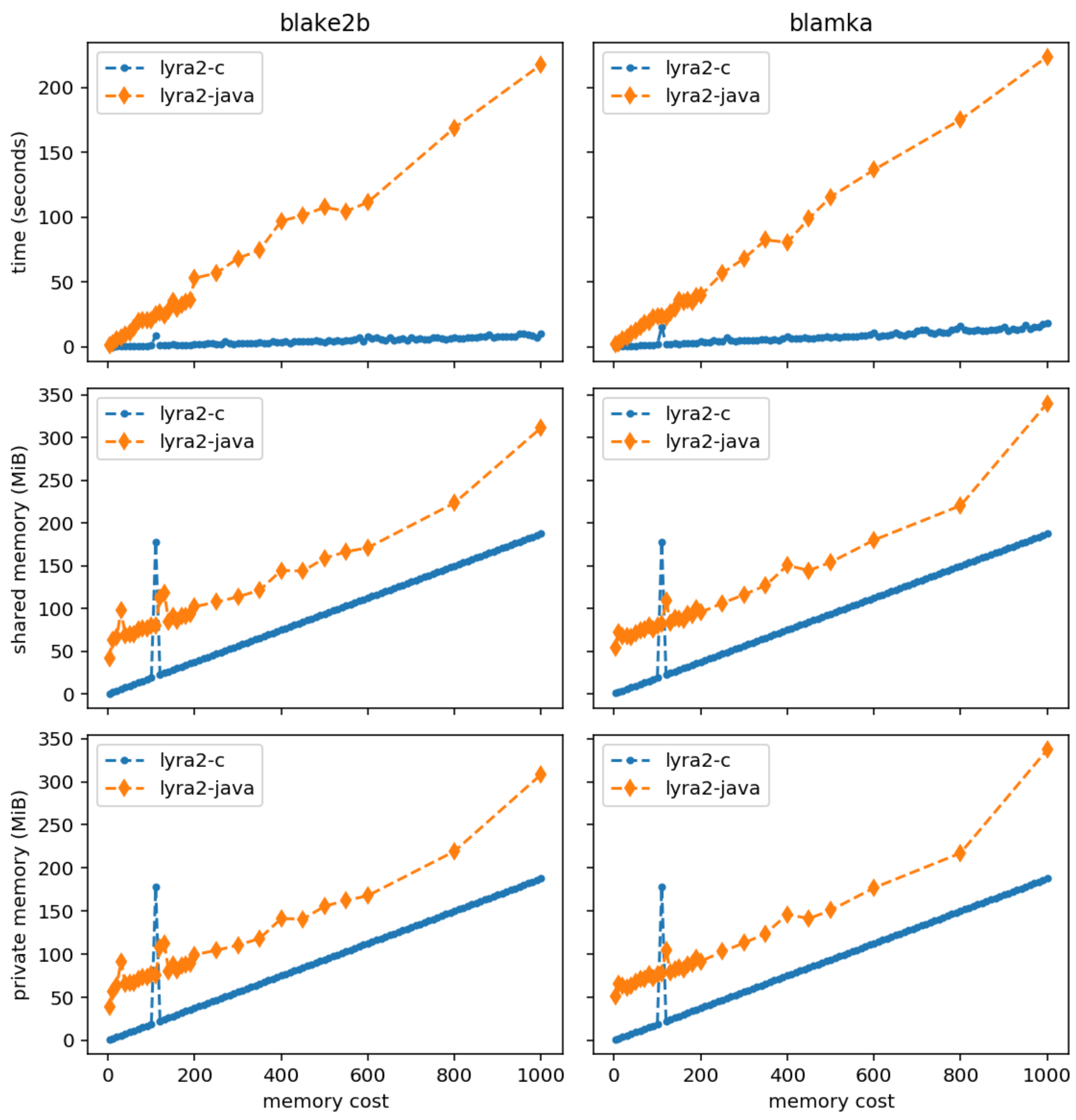


**Figure 5.2:** Comparison of `lyra2-c` and `lyra2-java`: 2048 columns, fixed time cost of 10. BLAKE2b sponge on the left, BlaMka sponge on the right.

### 5.2.3 Fixed Memory Cost

There are two figures that show how running time and consumed memory depend on time cost when memory cost is fixed. Figure 5.3 shows a 256-column and Figure 5.4 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. Figure 5.4 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.

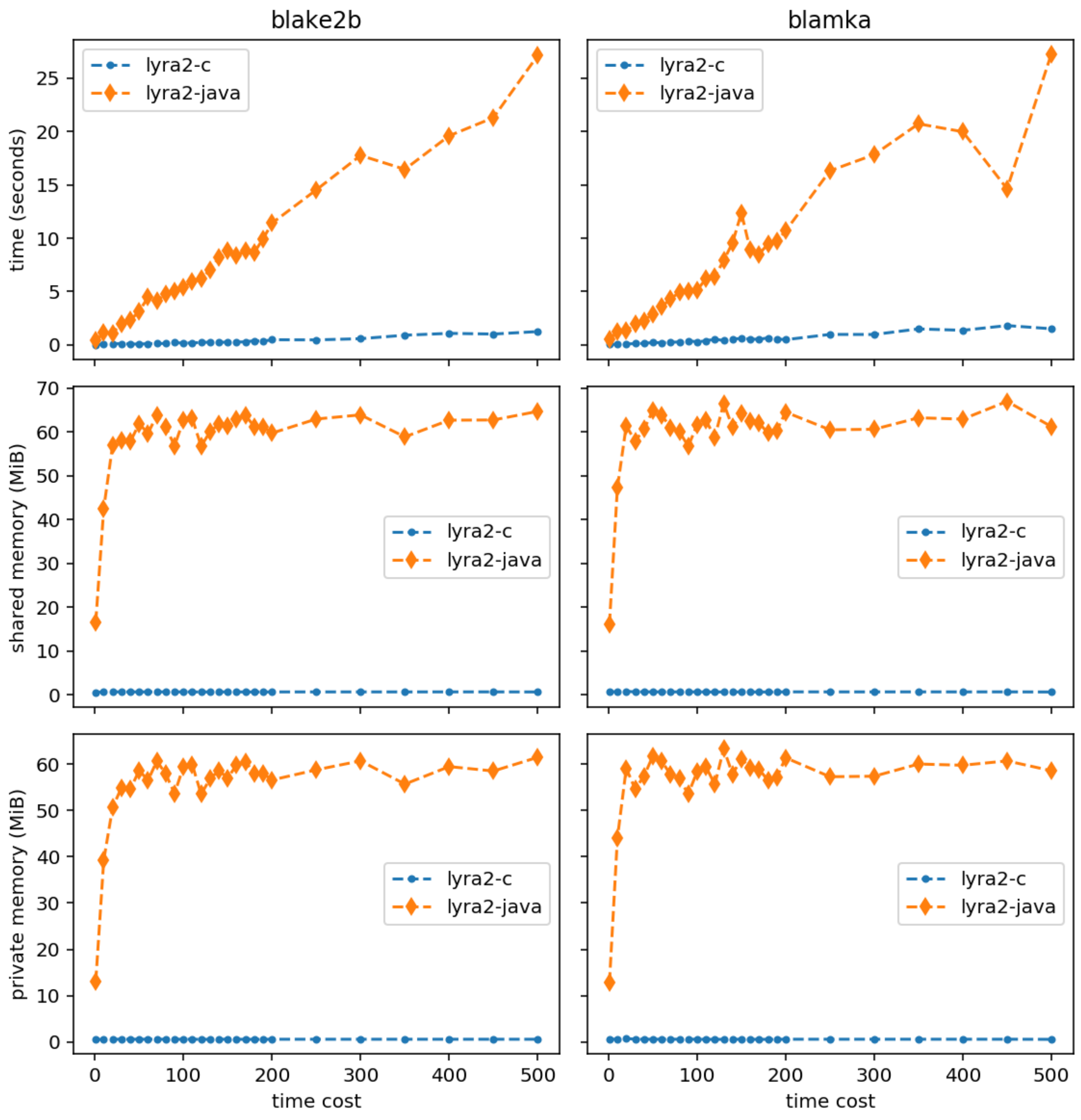


**Figure 5.3:** Comparison of `lyra2-c` and `lyra2-java`: 256 columns, fixed memory cost of 20. BLAKE2b sponge on the left, BlaMka sponge on the right.

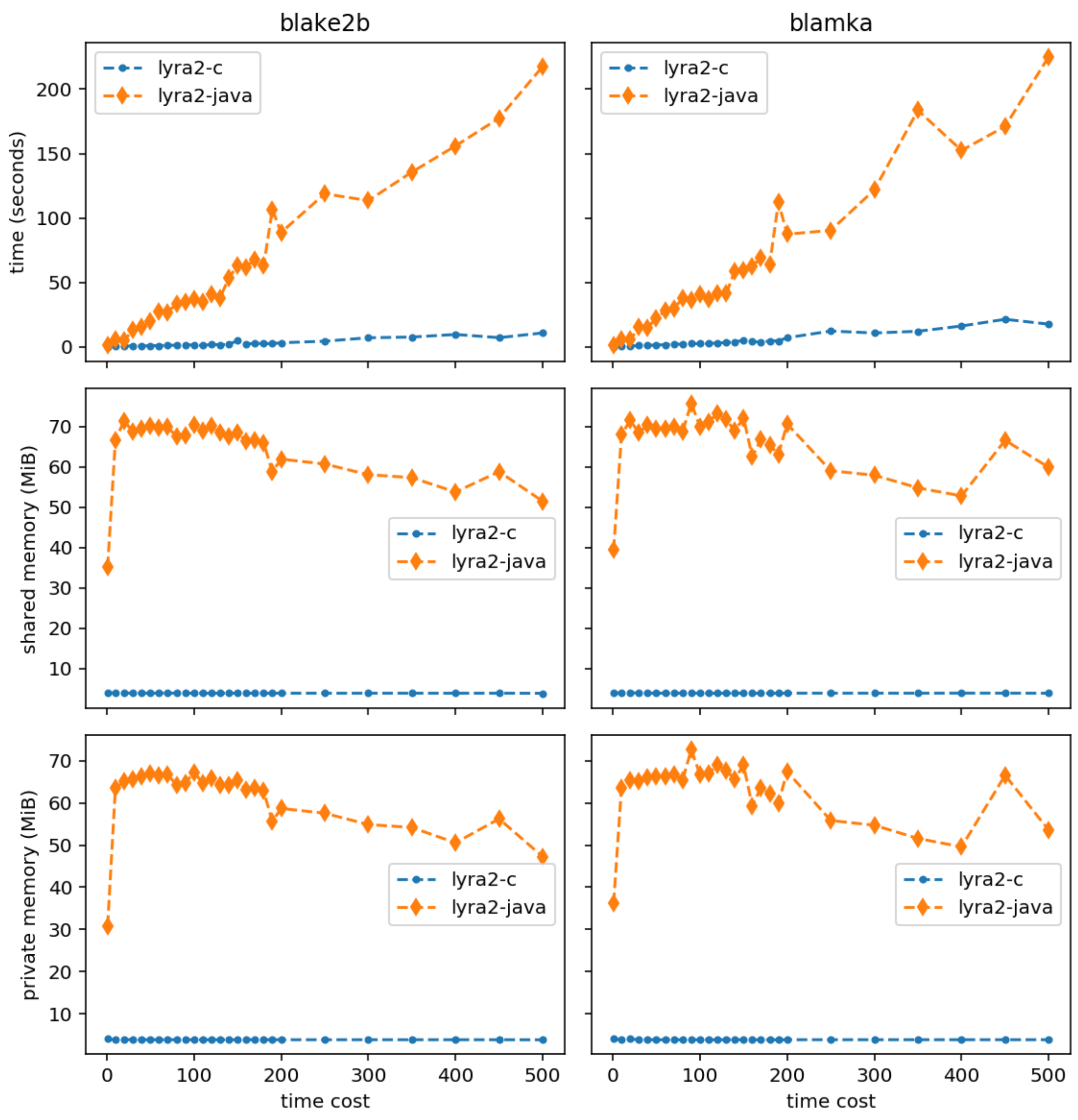


**Figure 5.4:** Comparison of `lyra2-c` and `lyra2-java`: 2048 columns, fixed memory cost of 20. BLAKE2b sponge on the left, BlaMka sponge on the right.

### 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. Figure 5.5 and Figure 5.6 correspond to a 256- and 2048-column configurations of Lyra2 that both use the BLAKE2b sponge. Figure 5.7 and Figure 5.8 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*. Arguably, this still allows for predictable and fine-tuned control of the time and memory resources required by the algorithm.

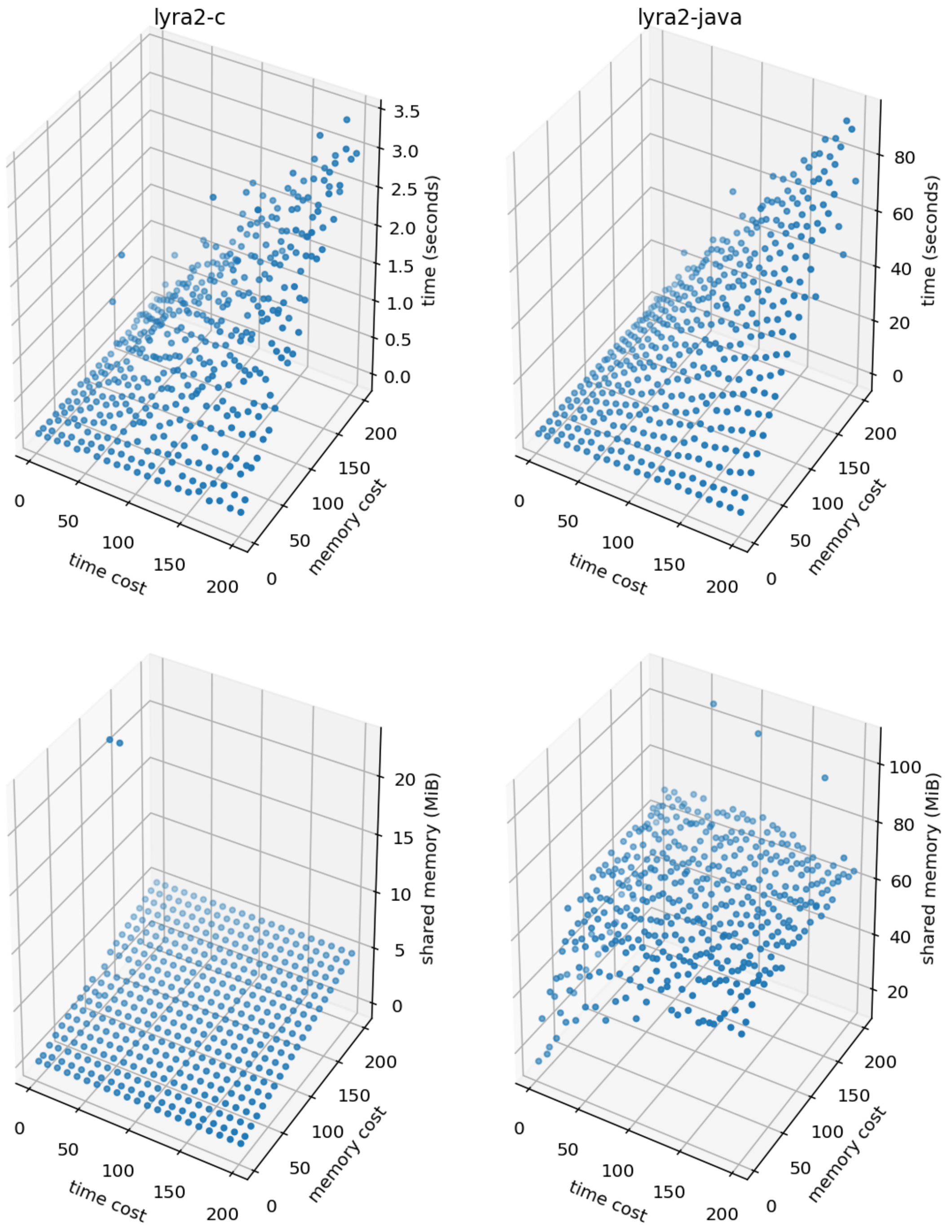


**Figure 5.5:** Comparison of `lyra2-c` and `lyra2-java`: BLAKE2b sponge, 256 columns.

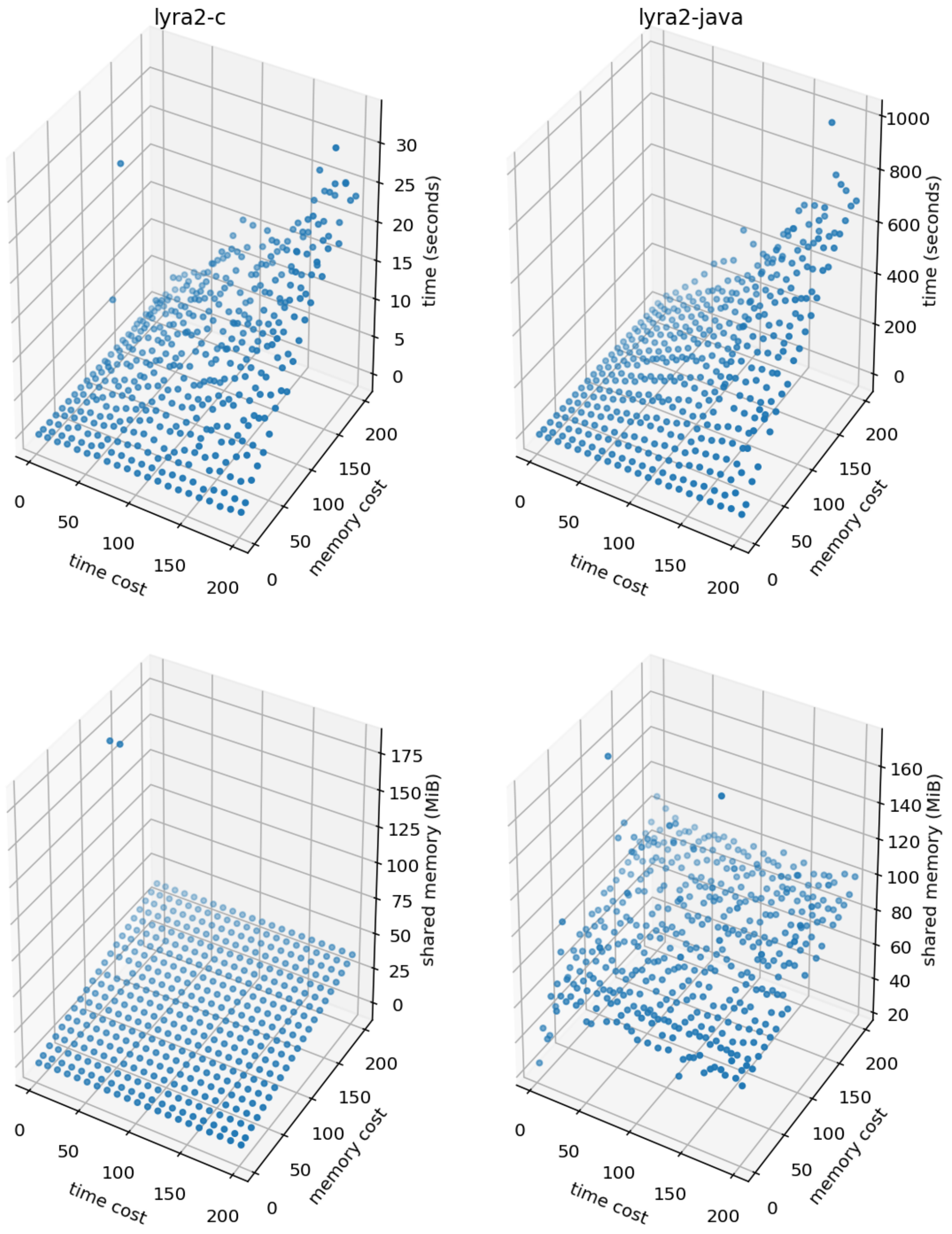


**Figure 5.6:** Comparison of `lyra2-c` and `lyra2-java`: BLAKE2b sponge, 2048 columns.

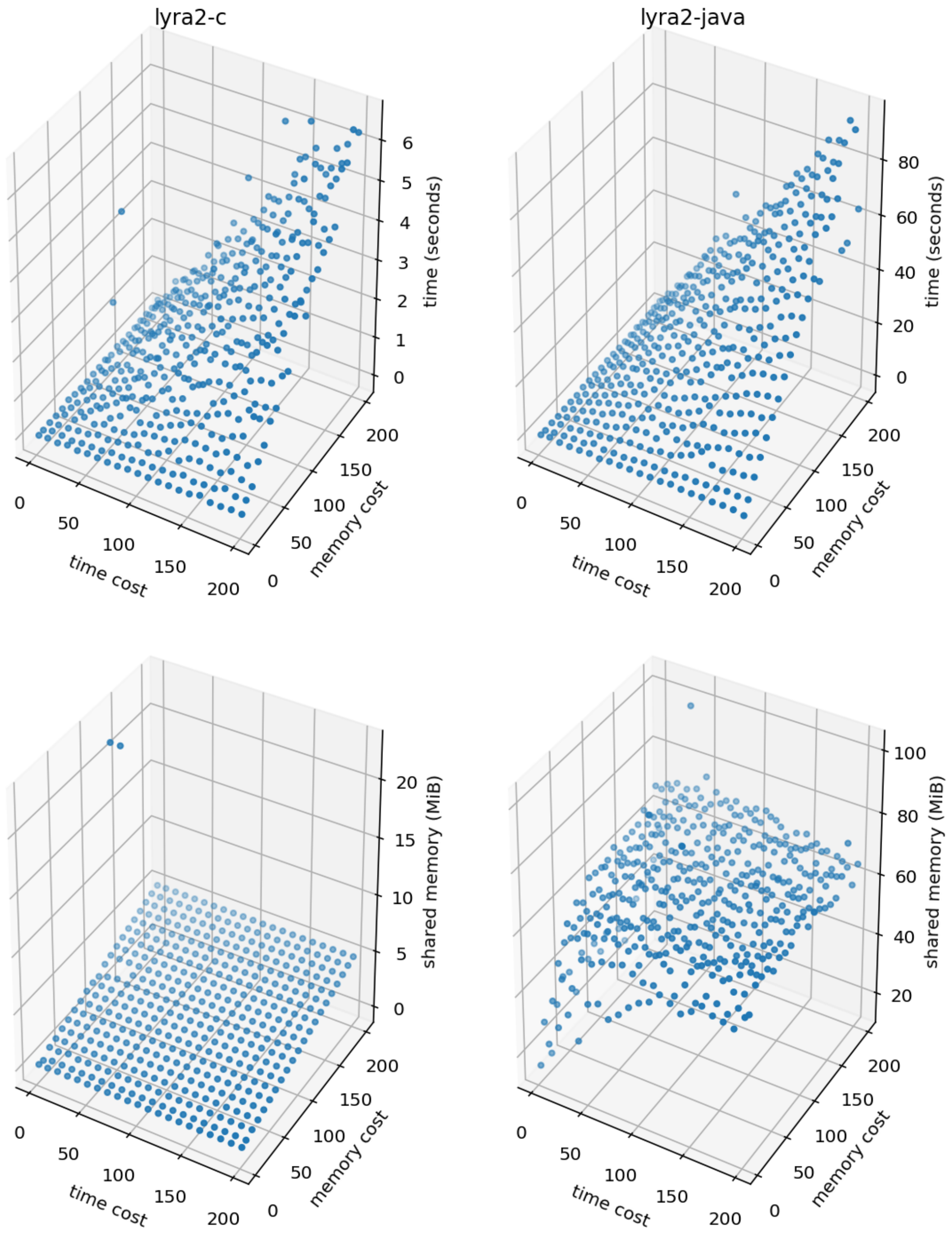


**Figure 5.7:** Comparison of `lyra2-c` and `lyra2-java`: BlaMka sponge, 256 columns.

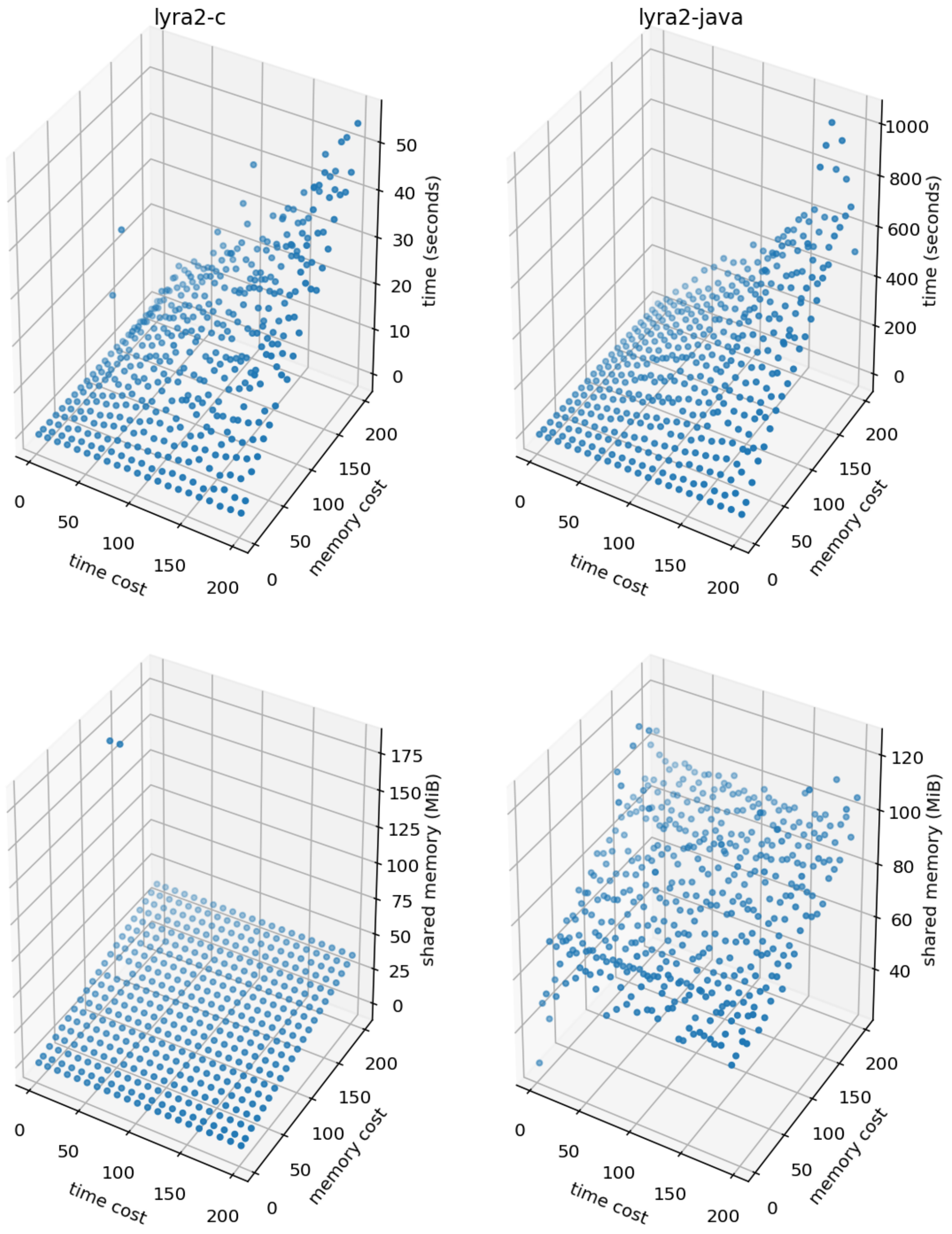


**Figure 5.8:** Comparison of `lyra2-c` and `lyra2-java`: BlaMka sponge, 2048 columns.

```
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
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
```

**Listing 5.2:** Android Studio Emulator: an error traceback connected to libGL.

### 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. It also cannot be denied that the `lyra2-java` project performs extra computations in order to ensure algorithm-level compatibility. These include the extra rotations required to simulate little-endian behavior, as well as simulating pointer arithmetic and 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 version of the supported Android device and the corresponding necessary API level. Since the `lyra2-java` project makes use of unsigned 64-bit number arithmetic, Java 1.8 support is required. This translates into the minimum Android version of 7.0 and the API level of 24 [7].

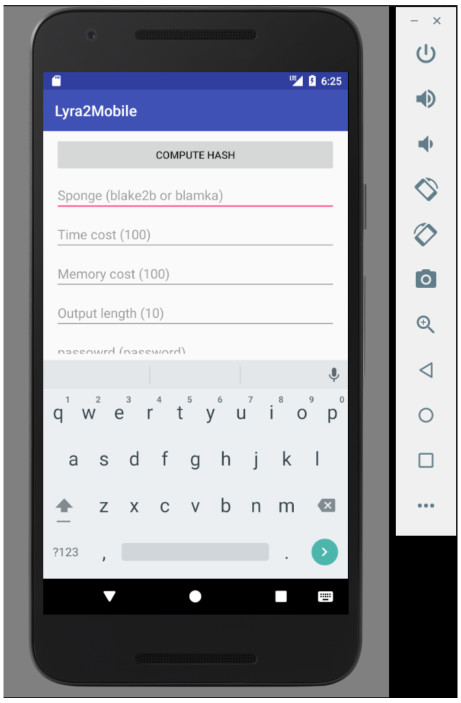
One of the most common development environments for Android applications is the Android Studio. Support for Java 1.8 features has not yet landed into the release version of this IDE. So, in order to be able to develop a mobile application with Lyra2, an Android Studio development version `3.0` and above needs to be installed. If you encounter a (similar) traceback as shown in Listing 5.2, one possible workaround is to instruct Android Studio to use system libraries:

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

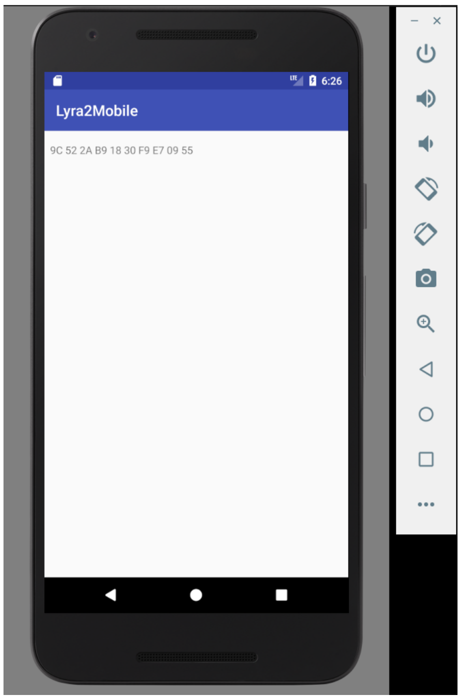
Once these preliminary steps are performed, the usual application development process can proceed. Since Android Studio is based on IntelliJ Studio, the `lyra2-java` project can be included as a dependency of the mobile application in largely the same manner as for any other project. Once the build system is configured, the `lyra2-java` project can be downloaded from Maven Central (or any other suitable repository) and made available to the developer.

To demonstrate the entire process, a simple Android application was developed and made available on GitHub [66]. It consists of a main screen and a results screen. The main screen allows the user to choose some 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 5.9 summarizes the result. In particular, Figure 5.9a and Figure 5.9b show the main screen and the results screen when running Lyra2 in the mobile emulator with the default parameters. At the same time Figure 5.9c and Figure 5.9d show the settings and the result for the BlaMka sponge. In both cases the hash values match those from the manual testing Section 5.1.2. In each case the computation lasted for approximately 60 seconds which is consistent with the desktop version times. However, it should once again be noted that the application was run in an emulator and not on an actual mobile device.

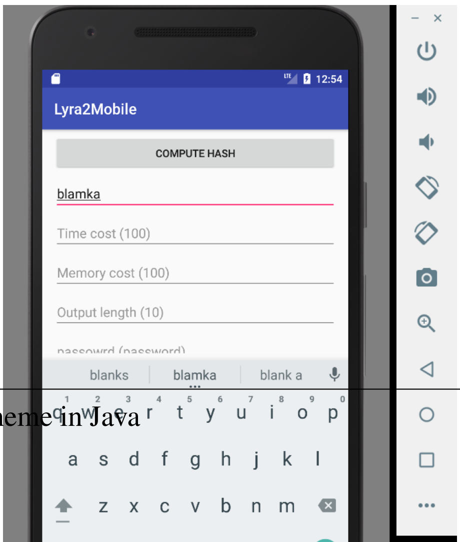


**(a)** Main Screen With the BLAKE2b Sponge



```
9C 52 2A B9 18 30 F9 E7 09 55
```

**(b)** Resulting Hash With the BLAKE2b Sponge



# 6 Conclusion

The reference `lyra2-c` implementation of Lyra2 was successfully ported from C99 to Java 1.8. The resulting `lyra2-java` implementation produces identical hash values. In order to achieve that, several compatibility issues had to be addressed. In particular, the mismatching endianness had to be constantly accounted for throughout all stages of the algorithm.

The extra byte rotations introduced by that require additional processing time. They cannot be avoided by design but can sometimes be performed simultaneously during other operations. In particular, the BLAKE2b and BlaMka sponge classes were adjusted to use both left and right rotations. This allowed to cut three quarters of the extra operations. Other compatibility issues included the need for pointer arithmetic simulation as well as 64-bit unsigned division.

The performance comparison of the two projects showed that the reference C implementation in general executes faster and requires less memory than its Java counterpart. This could be explained by the extra steps which the `lyra2-java` project has to take in order to produce compatible hash values. As well as that programming skill and language features could have played a part.

The mobile application was presented as proof of concept for Lyra2 integration. It showed that the `lyra2-java` project can be included into an Android application in a similar fashion as any other Java library. Compared to the C reference implementation this might be easier for the developer. In turn, the saved developer time may be worth the likely performance sacrifice.

Finally, there are several viable options for further research. Firstly, algorithm-level compatibility can be sacrificed in favor of performance. However, the author of this paper personally believes 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 some optimization opportunities were pointed out in this work, there are likely to be more.

# Bibliography

## References

[1] M. Abdalla, F. Benhamouda, and P. MacKenzie. „Security of the J-PAKE Password-authenticated Key Exchange Protocol“. In: *Symposium on Security and Privacy*. IEEE, 2015, pp. 571–587.

[2] A. Adams and M. A. Sasse. „Users are Not the Enemy“. In: *Communications of the ACM* 42.12 (1999), pp. 40–46.

[3] D. Adrian et al. „Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice“. In: *Conference on Computer and Communications Security*. ACM, 2015, pp. 5–17.

[4] F. Aloul, S. Zahidi, and W. El-Hajj. „Two Factor Authentication Using Mobile Phones“. In: *International Conference on Computer Systems and Applications*. IEEE, 2009, pp. 641–644.

[5] R. J. Anderson, E. Biham, and L. R. Knudsen. *Serpent: a Proposal for the Advanced Encryption Standard*. AES Algorithm Submission. 1998. URL: https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/archived-crypto-projects/aes-development (visited on 11/14/2017).

[6] 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.

[13] P. Arias-Cabarcos et al. „Comparing Password Management Software: Towards New Directions in Usable and Secure Enterprise Authentication“. In: *IT Professional* 18.5 (2016), pp. 34–40.

[14] D. Astels. *Test-driven Development: a Practical Guide*. Upper Saddle River, New Jersey: Prentice Hall, 2003, p. 562.

[15] J. P. Aumasson et al. *BLAKE2: Simpler, Smaller, Fast as MD5*. Cryptology ePrint Archive, Report 2013/322. 2013. URL: http://eprint.iacr.org/2013/322 (visited on 10/23/2017).

[16] J. P. Aumasson et al. *SHA-3 Proposal BLAKE*. Submission to NIST (Round 3). 2011. URL: https://csrc.nist.gov/projects/hash-functions/sha-3-project (visited on 11/14/2017).

[17] G. Avoine, P. Junod, and P. Oechslin. „Characterization and Improvement of Time-memory Trade-off Based on Perfect Tables“. In: *ACM Transactions on Information and System Security* 11.4 (2008), pp. 1–22.

[18] R. Barnes et al. *Deprecating Secure Sockets Layer Version 3.0*. RFC 7568. Internet Engineering Task Force, 2015. URL: https://tools.ietf.org/html/rfc7568 (visited on 11/06/2017).

[19] K. Beck. *Test-driven Development by Example*. Boston, Massachusetts: Addison-Wesley, 2003, p. 224.

[20] S. M. Bellovin and M. Merritt. „Augmented Encrypted Key Exchange: a Password-based Protocol Secure Against Dictionary Attacks and Password File Compromise“. In: *Conference on Computer and Communications Security*. ACM, 1993, pp. 244–250.

[21] S. M. Bellovin and M. Merritt. „Encrypted Key Exchange: Password-based Protocols Secure Against Dictionary Attacks“. In: *Symposium on Security and Privacy*. IEEE, 1992, pp. 72–84.

[22] G. Bertoni et al. *Cryptographic Sponge Functions*. Tech. rep. 2007. URL: https://keccak.team/files/CSF-0.1.pdf (visited on 10/23/2017).

[23] G. Bertoni et al. *Duplexing the Sponge: Single-pass Authenticated Encryption and Other Applications*. Cryptology ePrint Archive, Report 2011/499. 2011. URL: http://eprint.iacr.org/2011/499 (visited on 10/23/2017).

[24] G. Bertoni et al. *Keccak Specifications*. Submission to NIST (Round 3). 2011. URL: https://csrc.nist.gov/projects/hash-functions/sha-3-project (visited on 11/14/2017).

[25] A. Biryukov, D. Dinu, and D. Khovratovich. *Argon2*. Tech. rep. 2015. URL: https://password-hashing.net/submissions/specs/Argon-v3.pdf (visited on 10/23/2017).

[26] J. Bonneau. „The Science of Guessing: Analyzing an Anonymized Corpus of 70 Million Passwords“. In: *Symposium on Security and Privacy*. IEEE, 2012, pp. 538–552.

[27] J. Bonneau et al. „Passwords and the Evolution of Imperfect Authentication“. In: *Communications of the ACM* 58.7 (2015), pp. 78–87.

[35] P. Bours. „Continuous Keystroke Dynamics: a Different Perspective Towards Biometric Evaluation“. In: *Information Security Technical Report* 17.1 (2012), pp. 36–43.

[36] C. Braz and J. M. Robert. „Security and Usability: the Case of the User Authentication Methods“. In: *Conference on l'Interaction Homme-machine*. ACM, 2006, pp. 199–203.

[37] W. E. Burr et al. *NIST Special Publication 800-63-2 Electronic Authentication Guideline*. 2013.

[38] C. Burwick et al. *MARS — a Candidate Cipher for AES*. AES Algorithm Submission. 1998. URL: https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/archived-crypto-projects/aes-development (visited on 11/14/2017).

[39] T. Caddy. „Side-channel Attacks“. In: *Encyclopedia of Cryptography and Security*. Boston, Massachusetts: Springer, 2005, pp. 576–577.

[40] Y. Cheon and G. T. Leavens. „A Simple and Practical Approach to Unit Testing: the JML and JUnit way“. In: *European Conference on Object-oriented Programming*. Springer, 2002, pp. 231–255.

[43] A. Kolawa D. Huizinga. *Automated Defect Prevention: Best Practices in Software Management*. Hoboken, New Jersey: Wiley, 2007, p. 426.

[44] J. Daemen and V. Rijmen. *AES Proposal: Rijndael*. AES Algorithm Submission. 1998. URL: https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/archived-crypto-projects/aes-development (visited on 11/14/2017).

[45] J. Daemen and V. Rijmen. *The Design of Rijndael: AES — the Advanced Encryption Standard*. Berlin, Germany: Springer, 2002, p. 238.

[46] E. Daka and G. Fraser. „A Survey on Unit Testing Practices and Problems“. In: *International Symposium on Software Reliability Engineering*. IEEE, 2014, pp. 201–211.

[47] M. Dell'Amico, P. Michiardi, and Y. Roudier. „Password Strength: an Empirical Analysis“. In: *International Conference on Computer Communications*. IEEE, 2010, pp. 1–9.

[48] Y. Deng and Y. Zhong. „Keystroke Dynamics User Authentication Based on Gaussian Mixture Model and Deep Belief Nets“. In: *ISRN Signal Processing* 2013.1 (2013), pp. 1–7.

[49] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. Network Working Group, 2008. URL: https://tools.ietf.org/html/rfc5246 (visited on 10/23/2017).

[50] Z. Durumeric et al. „The Matter of Heartbleed“. In: *Internet Measurement Conference*. ACM, 2014, pp. 475–488.

[51] F. Elberzhager et al. „Reducing Test Effort: a Systematic Mapping Study on Existing Approaches“. In: *Information and Software Technology* 54.10 (2012), pp. 1092–1106.

[52] N. Ferguson et al. *The Skein Hash Function Family*. Submission to NIST (Round 3). 2011. URL: https://csrc.nist.gov/projects/hash-functions/sha-3-project (visited on 11/14/2017).

[53] C. Forler, S. Lucks, and J. Wenzel. *Catena: a Memory-consuming Password-scrambling Framework*. Cryptology ePrint Archive, Report 2013/525. 2013. URL: http://eprint.iacr.org/2013/525 (visited on 10/23/2017).

[54] A. O. Freier, P. Karlton, and P. C. Kocher. *The Secure Sockets Layer (SSL) Protocol Version 3.0*. RFC 6101. Internet Engineering Task Force, 2011. URL: https://tools.ietf.org/html/rfc6101 (visited on 10/23/2017).

[55] P. Gauravaram et al. *Grøstl — a SHA-3 Candidate*. Submission to NIST (Round 3). 2011. URL: https://csrc.nist.gov/projects/hash-functions/sha-3-project (visited on 11/14/2017).

[56] S. Gaw and E. W. Felten. „Password Management Strategies for Online Accounts“. In: *Symposium on Usable Privacy and Security*. ACM, 2006, pp. 44–55.

[69] L. Gong et al. „Protecting Poorly Chosen Secrets From Guessing Attacks“. In: *IEEE Journal on Selected Areas in Communications* 11.5 (1993), pp. 648–656.

[77] P. A. Grassi, M. E. Garcia, and J. L. Fenton. *NIST Special Publication 800-63-3 Digital Identity Guidelines*. 2017.

[78] M. Green and M. Smith. „Developers are Not the Enemy: the Need for Usable Security APIs“. In: *Security and Privacy* 14.5 (2016), pp. 40–46.

[79] F. Hao and P. Ryan. *J-PAKE: Authenticated Key Exchange Without PKI*. Cryptology ePrint Archive, Report 2010/190. 2010. URL: https://eprint.iacr.org/2010/190 (visited on 11/15/2017).

[80] J. Ho and D. K. Kang. „One-class naïve Bayes With Duration Feature Ranking for Accurate User Authentication Using Keystroke Dynamics“. In: *Applied Intelligence* 47.2 (2017), pp. 1–18.

[81] J. Hong. „The State of Phishing Attacks“. In: *Communications of the ACM* 55.1 (2012), pp. 74–81.

[82] E. Ivannikova, G. David, and T. Hämäläinen. „Anomaly Detection Approach to Keystroke Dynamics Based User Authentication“. In: *Symposium on Computers and Communications*. IEEE, 2017, pp. 885–889.

[83] B. Ives, K. R. Walsh, and H. Schneider. „The Domino Effect of Password Reuse“. In: *Communications of the ACM* 47.4 (2004), pp. 75–78.

[84] A. K. Jain, A. Ross, and S. Prabhakar. „An Introduction to Biometric Recognition“. In: *IEEE Transactions on Circuits and Systems for Video Technology* 14.1 (2004), pp. 4–20.

[85] D. L. Jobusch and A. E. Oldehoeft. „A Survey of Password Mechanisms: Weaknesses and Potential Improvements. Part 1“. In: *Computers and Security* 8.7 (1989), pp. 587–604.

[86] D. L. Jobusch and A. E. Oldehoeft. „A Survey of Password Mechanisms: Weaknesses and Potential Improvements. Part 2“. In: *Computers and Security* 8.8 (1989), pp. 675–689.

[87] M. A. Simplício Jr. et al. *Lyra2: Password Hashing Scheme With Improved Security Against Time-memory Trade-offs*. Cryptology ePrint Archive, Report 2015/136. 2015. URL: http://eprint.iacr.org/2015/136 (visited on 10/23/2017).

[90] P. G. Kelley et al. „Guess Again (and Again and Again): Measuring Password Strength by Simulating Password-cracking Algorithms“. In: *Symposium on Security and Privacy*. IEEE, 2012, pp. 523–537.

[91] A. van Lamsweerde. „Formal Specification: a Roadmap“. In: *Conference on The Future of Software Engineering*. ACM, 2000, pp. 147–159.

[94] C. Mack. „The Multiple Lives of Moore’s Law“. In: *Spectrum* 52.4 (2015), pp. 31–37.

[95] T. Mackinnon, S. Freeman, and P. Craig. „Endo-testing: Unit Testing With Mock Objects“. In: *Extreme Programming Examined*. Boston, Massachusetts: Addison-Wesley, 2001, pp. 287–301.

[97] V. Matyas and Z. Riha. „Toward Reliable User Authentication Through Biometrics“. In: *Security and Privacy* 99.3 (2003), pp. 45–49.

[98] F. Monrose and A. D. Rubin. „Keystroke Dynamics as a Biometric for Authentication“. In: *Future Generation Computer Systems* 16.4 (2000), pp. 351–359.

[99] K. M. Moriarty, B. Kaliski, and A. Rusch. *PKCS #5: Password-based Cryptography Specification Version 2.1*. RFC 8018. Internet Engineering Task Force, 2017. URL: https://tools.ietf.org/html/rfc8018 (visited on 10/23/2017).

[101] D. M’Raihi et al. *HOTP: an HMAC-based One-time Password Algorithm*. RFC 4226. Network Working Group, 2005. URL: https://tools.ietf.org/html/rfc4226 (visited on 10/23/2017).

[102] D. M’Raihi et al. *TOTP: Time-based One-time Password Algorithm*. RFC 6238. Internet Engineering Task Force, 2011. URL: https://tools.ietf.org/html/rfc6238 (visited on 10/23/2017).

[103] T. Murakami, R. Kasahara, and T. Saito. „An Implementation and its Evaluation of Password Cracking Tool Parallelized on GPGPU“. In: *International Symposium on Communications and Information Technologies*. IEEE, 2010, pp. 534–538.

[104] P. Müller and A. Poetzsch-Heffter. „Formal Specification Techniques for Object-oriented Programs“. In: *Informatik als Innovationsmotor*. Springer, 1997, pp. 602–611.

[105] A. Narayanan and V. Shmatikov. „Fast Dictionary Attacks on Passwords Using Time-space Trade-off“. In: *Conference on Computer and Communications Security*. ACM, 2005, pp. 364–372.

[108] C. Percival. *Stronger Key Derivation via Sequential Memory-hard Functions*. Tech. rep. 2009. URL: http://www.tarsnap.com/scrypt/scrypt.pdf (visited on 10/23/2017).

[109] C. Percival and S. Josefsson. *The scrypt Password-based Key Derivation Function*. RFC 7914. Internet Engineering Task Force, 2016. URL: https://tools.ietf.org/html/rfc7914 (visited on 10/23/2017).

[110] A. Peslyak. *yescrypt — a Password Hashing Competition Submission*. Tech. rep. 2015. URL: https://password-hashing.net/submissions/specs/yescrypt-v2.pdf (visited on 10/23/2017).

[112] T. Polk and S. Turner. *Prohibiting Secure Sockets Layer (SSL) Version 2.0*. RFC 6176. Internet Engineering Task Force, 2011. URL: https://tools.ietf.org/html/rfc6176 (visited on 11/06/2017).

[113] T. Pornin. *The Makwa Password Hashing Function*. Tech. rep. 2015. URL: https://password-hashing.net/submissions/specs/Makwa-v1.pdf (visited on 10/23/2017).

[118] M. O. Rayes. „One-time Password“. In: *Encyclopedia of Cryptography and Security*. Boston, Massachusetts: Springer, 2005, pp. 446–446.

[119] M. Raza et al. „A Survey of Password Attacks and Comparative Analysis on Methods for Secure Authentication“. In: *World Applied Sciences Journal* 19.4 (2012), pp. 439–444.

[120] R. L. Rivest et al. *The RC6 Block Cipher*. AES Algorithm Submission. 1998. URL: https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/archived-crypto-projects/aes-development (visited on 11/14/2017).

[121] B. Schneier et al. *Twofish: a 128-bit Block Cipher*. AES Algorithm Submission. 1998. URL: https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/archived-crypto-projects/aes-development (visited on 11/14/2017).

[122] A. Serwadda et al. „Scan-based Evaluation of Continuous Keystroke Authentication Systems“. In: *IT Professional* 15.4 (2013), pp. 20–23.

[123] R. Shay et al. „Encountering Stronger Password Requirements: User Attitudes and Behaviors“. In: *Symposium on Usable Privacy and Security*. ACM, 2010, pp. 1–20.

[124] National Institute of Standards and Technology. *Advanced Encryption Standard (AES)*. Federal Information Processing Standards Publication 197. 2001. URL: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf (visited on 10/23/2017).

[127] X. Suo, Y. Zhu, and G. S. Owen. „Graphical Passwords: a Survey“. In: *Annual Computer Security Applications Conference*. IEEE, 2005, pp. 463–472.

[128] P. S. Teh, A. B. J. Teoh, and S. Yue. „A Survey of Keystroke Dynamics Biometrics“. In: *The Scientific World Journal* 2013 (2013), pp. 1–24.

[131] N. Tillmann, J. de Halleux, and T. Xie. „Parameterized Unit Testing: Theory and Practice“. In: *International Conference on Software Engineering*. IEEE, 2010, pp. 483–484.

[134] M. Weir et al. „Testing Metrics for Password Creation Policies by Attacking Large Sets of Revealed Passwords“. In: *Conference on Computer and Communications Security*. ACM, 2010, pp. 162–175.

[135] J. Wetzels. *Open Sesame: the Password Hashing Competition and Argon2*. Cryptology ePrint Archive, Report 2016/104. 2016. URL: http://eprint.iacr.org/2016/104 (visited on 10/23/2017).

[136] F. Wiemer and R. Zimmermann. „High-speed Implementation of bcrypt Password Search Using Special-purpose Hardware“. In: *Conference on Reconfigurable Computing and FPGAs*. IEEE, 2014, pp. 1–6.

[137] L. Williams and G. Kudrjavets. „On the Effectiveness of Unit Test Automation at Microsoft“. In: *International Symposium on Software Reliability Engineering*. IEEE, 2010, pp. 81–89.

[138] H. Wu. *The Hash Function JH*. Submission to NIST (Round 3). 2011. URL: https://csrc.nist.gov/projects/hash-functions/sha-3-project (visited on 11/14/2017).

[139] M. Wu, R. C. Miller, and S. L. Garfinkel. „Do Security Toolbars Actually Prevent Phishing Attacks?“ In: *Conference on Human Factors in Computing Systems*. ACM, 2006, pp. 601–610.

[141] J. Yan et al. „Password Memorability and Security: Empirical Results“. In: *Security and Privacy* 2.5 (2004), pp. 25–31.

[142] Y. Zhang, F. Monrose, and M. K. Reiter. „The Security of Modern Password Expiration: an Algorithmic Framework and Empirical Analysis“. In: *Conference on Computer and Communications Security*. ACM, 2010, pp. 176–186.

## Online References

[7] *Android Studio Java Features Support*. URL: https://developer.android.com/studio/write/java8-support.html (visited on 11/06/2017).

[8] *AppVeyor Homepage*. URL: https://www.appveyor.com/ (visited on 10/23/2017).

[9] *Archive of the Original AES Competition Webpage*. URL: https://web.archive.org/web/20011218002449/http://csrc.nist.gov/encryption/aes/index2.html (visited on 11/13/2017).

[10] *Argon2 Discussion of the Unit Testing Harness*. URL: https://github.com/P-H-C/phc-winner-argon2/issues/85 (visited on 11/06/2017).

[11] *Argon2 in Django Web Framework*. URL: https://docs.djangoproject.com/en/1.11/topics/auth/passwords/#using-argon2-with-django (visited on 10/23/2017).

[12] *Argon2 Unit Testing Harness*. URL: https://github.com/P-H-C/phc-winner-argon2/blob/68dd41f75c4c9df95f403154c0e19252fcf0f513/src/test.c (visited on 10/23/2017).

[28] *Boost Homepage*. URL: http://www.boost.org/ (visited on 11/15/2017).

[29] *Boost Library Submission Process*. URL: http://www.boost.org/development/submissions.html (visited on 11/06/2017).

[30] *Boost Mailing List*. URL: http://www.boost.org/community/groups.html (visited on 11/06/2017).

[31] *Boost.Test Data-driven Test Cases*. 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 10/23/2017).

[32] *Boost.Test Datasets for Data-driven Unit Tests*. 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 10/23/2017).

[33] *Boost.Test Documentation*. URL: http://www.boost.org/doc/libs/1_65_1/libs/test/doc/html/index.html (visited on 10/23/2017).

[34] *Boost.Test Operations on Datasets*. URL: http://www.boost.org/doc/libs/1_65_1/libs/test/doc/html/boost_test/tests_organization/test_cases/test_case_generation/operations.html (visited on 11/15/2017).

[41] *CircleCI Homepage*. URL: https://circleci.com/ (visited on 10/23/2017).

[42] *Comodo Certificate Authority Incident Report*. URL: https://www.comodo.com/Comodo-Fraud-Incident-2011-03-23.html (visited on 10/23/2017).

[57] *GitHub Pull Request of the Test Harness for Lyra2*. URL: https://github.com/leocalm/Lyra/pull/7 (visited on 10/23/2017).

[58] *GitHub Repository of the Argon2 Project*. URL: https://github.com/P-H-C/phc-winner-argon2/tree/54ff100b0717505493439ec9d4ca85cb9cbdef00 (visited on 11/06/2017).

[59] *GitHub Repository of the Argonishche Project*. URL: https://github.com/yandex/argon2/tree/685e5f4eb9f6453c08b1ea71a98873c7f4dee5c5 (visited on 11/06/2017).

[60] *GitHub Repository of the Catch Project*. URL: https://github.com/catchorg/Catch2 (visited on 11/15/2017).

[61] *GitHub Repository of the Catena-Axungia Project*. URL: https://github.com/medsec/catena-axungia/tree/9a65c86858ad9e423cad14552b664f14509284c1 (visited on 11/06/2017).

[62] *GitHub Repository of the Catena-Variants Project*. URL: https://github.com/medsec/catena-variants/tree/5baf9c34b96e495a52f8ded6612187c7f0033efb (visited on 11/06/2017).

[63] *GitHub Repository of the Google Test Project*. URL: https://github.com/google/googletest (visited on 11/15/2017).

[64] *GitHub Repository of the Lyra2 Comparison Project*. URL: https://github.com/alisianoi/lyra2-compare/tree/206851cd2f322e3dabae7439c7f9f75bef81d3a1 (visited on 11/06/2017).

[65] *GitHub Repository of the Lyra2 Java Project*. URL: https://github.com/alisianoi/lyra2-java/tree/2c05da3f6739859f3c3abe5116754666689028a2 (visited on 11/06/2017).

[66] *GitHub Repository of the Lyra2 Mobile Project*. URL: https://github.com/alisianoi/lyra2-mobile/tree/88904b25fc5f637f8c3ec14eec4bce88db176b41 (visited on 11/06/2017).

[67] *GitHub Repository of the Lyra2 Reference Implementation*. URL: https://github.com/leocalm/Lyra/tree/e686d1327bccfdbf5725bf39efccf58486cadb46 (visited on 11/06/2017).

[68] *GitHub Repository of the Lyra2 Reference Implementation Fork*. URL: https://github.com/alisianoi/Lyra/tree/e686d1327bccfdbf5725bf39efccf58486cadb46 (visited on 11/06/2017).

[70] *Google Security Blog: ANSSI Incident*. URL: https://security.googleblog.com/2013/12/further-improving-digital-certificate.html (visited on 10/23/2017).

[71] *Google Security Blog: DigiNotar Incident*. URL: https://security.googleblog.com/2011/08/update-on-attempted-man-in-middle.html (visited on 10/23/2017).

[72] *Google Security Blog: Google Chrome to Distrust Symantec Certificates*. URL: https://security.googleblog.com/2017/09/chromes-plan-to-distrust-symantec.html (visited on 10/23/2017).

[73] *Google Security Blog: Google Chrome to Distrust TURKTRUST*. URL: https://security.googleblog.com/2013/01/enhancing-digital-certificate-security.html (visited on 10/23/2017).

[74] *Google Security Blog: Google Chrome to Distrust WoSign and StartCom Certificates*. URL: https://security.googleblog.com/2016/10/distrusting-wosign-and-startcom.html (visited on 10/23/2017).

[75] *Google Security Blog: Google Chrome to Finalize Distrust in WoSign and StartCom Certificates*. URL: https://security.googleblog.com/2017/07/final-removal-of-trust-in-wosign-and.html (visited on 10/23/2017).

[76] *Google Security Blog: National Informatics Centre of India Incident*. URL: https://security.googleblog.com/2014/07/maintaining-digital-certificate-security.html (visited on 10/23/2017).

[88] *JUnit Homepage*. URL: http://junit.org/ (visited on 11/15/2017).

[89] *JUnit Parametrized Testing Documentation*. URL: https://github.com/junit-team/junit4/wiki/Parameterized-tests (visited on 11/15/2017).

[92] *Lyra2 Continuous Integration on TravisCI*. URL: https://travis-ci.org/alisianoi/lyra2-java (visited on 10/23/2017).

[93] *Lyra2 Package on Maven Central Version 1.3*. URL: http://search.maven.org/#artifactdetails%7Ccom.github.alisianoi%7Clyra2%7C1.3%7Clyra2 (visited on 10/23/2017).

[96] *Makwa Homepage*. URL: http://www.bolet.org/makwa/ (visited on 10/23/2017).

[100] *Mozilla Wiki Page: Mozilla to Distrust Symantec Certificates*. URL: https://wiki.mozilla.org/index.php?title=CA:Symantec_Issues&oldid=1182300 (visited on 11/06/2017).

[106] *Nose Homepage*. URL: https://nose.readthedocs.io/en/latest/ (visited on 11/15/2017).

[107] *Nose Parametrized Testing*. URL: http://nose.readthedocs.io/en/latest/writing_tests.html#test-generators (visited on 11/15/2017).

[111] *Picocli — a Mighty Tiny Command Line Interface*. URL: http://picocli.info/ (visited on 10/23/2017).

[114] *Project Jupyter Homepage*. URL: http://jupyter.org/ (visited on 10/23/2017).

[115] *py.test Homepage*. URL: https://docs.pytest.org/en/latest/ (visited on 11/15/2017).

[116] *py.test Parametrized Testing*. URL: https://docs.pytest.org/en/latest/example/parametrize.html (visited on 11/15/2017).

[117] *Python Homepage*. URL: https://www.python.org/ (visited on 11/15/2017).

[125] *Summary of the AES Competition Webpage*. URL: https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/archived-crypto-projects/aes-development (visited on 11/13/2017).

[126] *Summary of the SHA-3 Competition Webpage*. URL: https://www.nist.gov/publications/nist-hash-competition (visited on 11/13/2017).

[129] *TestNG Homepage*. URL: http://testng.org/doc/index.html (visited on 10/23/2017).

[130] *TestNG Parametrized Testing Documentation*. URL: http://testng.org/doc/documentation-main.html#parameters (visited on 11/06/2017).

[132] *TravisCI Homepage*. URL: https://travis-ci.org/ (visited on 10/23/2017).

[133] *unittest Homepage*. URL: https://docs.python.org/3.5/library/unittest.html (visited on 11/15/2017).

[140] *YAML Specification*. URL: http://www.yaml.org/spec/1.2/spec.html (visited on 11/15/2017).