PHAs: Comparing Hardware and Cost of Password Hashing Algorithms, and the Importance of Upgrading Hardware For Better Password Storage Security

Ronan Chay Loong
University of Guelph

ABSTRACT

Very little literature currently explores how the parameter inputs and the hardware used for password hashing algorithms impact the cost, which is regrettable considering password security today heavily relies on cost. In this paper, I analyze the cost of three types of hashing algorithms for password hashing: a general purpose hashing algorithm, a popular password hashing algorithm, and a state-of-the-art password hashing algorithm. In particular, I analyze the SHA-512, BCrypt, and Argon2id hashing algorithms in terms of time against varying parameter values and hardware capabilities of the device the algorithm is running on. Comparing these factors will provide a richer understanding of their relationship and allow for more optimal decision-making when choosing a password hashing algorithm to implement in a system. Results indicate that there indeed is a significant hardware impact on the cost of some of these hashing algorithms.

ACM Reference Format:
Ronan Chay Loong. 2024. PHAs: Comparing Hardware and Cost of Password Hashing Algorithms, and the Importance of Upgrading Hardware For Better Password Storage Security. *Final Project Report, School of Computer Science, University of Guelph (CIS*4520 Introduction to Cryptography),* 8 pages.

1 INTRODUCTION

Password storage is a part of security that falls solely on developers' shoulders to ensure these vital passwords are stored securely. Data breaches are worryingly common [16] and can have massive repercussions worth hundreds of thousands of US Dollars for a company [10]. Not to mention the malicious actors getting their hands on more and more powerful hardware for cracking password hashes. There is a need for software developers to be increasingly aware of what are the optimal password hashing algorithms and relevant parameter values to be used to ensure good security while also bearing low cost. However, the current way of thinking when it comes to increasing security is not to create unbreakable hashing algorithms: creating a perfectly secure password hashing algorithm is very difficult. Even worse, malicious actors today are capable of feasibly brute-forcing a password that was hashed with a general purpose hashing algorithm like SHA-256 as well simply with powerful hardware and a little patience. To combat this, security is instead improved by increasing the cost of computing a password hash. By making each password hash take longer to compute, developers are able to reasonably protect against brute force attacks as malicious actors would need exponentially more time to perform such attacks, therefore making the password hash more secure. Still, there is a limit to how much the cost can be increased as an exaggerated cost would result in two problems: 1) users of the system would notice the slow performance, and 2) malicious actors may perform Denial of Service attacks by performing many login attempts in a short period of time [14]. As a result, there needs to be a balance between cost and security. As a software developer, getting a deep understanding of this topic felt critical to my learning and the motivation behind this research project.

This research project aims to analyse and compare three types of hashing algorithms: 1) a general purpose hashing algorithm that is not meant for password hashing, 2) a popular password hashing algorithm used by many in the industry, and 3) a state-of-the-art password hashing algorithm that is regarded as the best currently available. As such, the SHA-512, BCrypt, and Argon2 algorithms were chosen for these reasons respectively. The correlation between the parameter and the cost of each of these password hashing algorithms will be analysed to find the parameter-cost combination needed for the perfect balance between minimum cost and maximum security. The impact of hardware with different capabilities on the parameter-cost relationship will also be examined to determine whether upgrading the hardware of a system for better password security is a viable option that is worthwhile to pursue. An Android mobile device and a MacBook laptop will be used to compare the difference between different hardware devices.

2 RESEARCH OBJECTIVES

The first objective of this research project is to gain an in-depth understanding of the effect of password hashing algorithm parameters on cost for three types of password hashing algorithms currently available in the industry. By analysing the effects of varying parameters against the resource costs, we can get an idea of how the two are correlated. The relationship between parameter and cost will allow for developers to choose an appropriate combination that will allow for the maximum possible cost for the system while still being within the acceptable limit or the performance requirements for the system. As mentioned in the introduction, the perfect amount of resource costs can be difficult to achieve: too low of a cost and malicious actors will be able to feasibly perform brute-force attacks on a password hash with the right tools and amount of time. Too high of a cost however, and the system performance degrades significantly and not only would users notice this performance drop,

*CIS*4520 Introduction to Cryptography, School of Computer Science, University of Guelph, Guelph, ON, Canada*

but malicious actors may then change their attack vector by instead performing Denial of Service attacks.

Once the parameter-cost relationship is thoroughly explored, using two different devices with significantly different hardware capabilities in terms of memory and processing power to compare the parameter-cost relationships for the algorithms on the two devices will then allow getting an understanding of the impact of hardware on these algorithms. By doing so, a related question can then be answered: would it be viable to upgrade the hardware of a system to obtain a better cost to security ratio? Hardware is relatively expensive to purchase and set up. It could also result in possibly critical downtime for the system and consequently significant losses for a company. Before committing to such an endeavour, we need to know whether it will pay off. An understanding of the impact of hardware on a password hashing algorithm's cost is therefore necessary, which is the second objective of this research project.

Combining these two objectives together will subsequently allow finding the optimal parameter values and password hashing algorithm to obtain the right balance of cost versus security based on the hardware capabilities of a system based on its use case, and future plans for the system.

3 RELATED WORK

3.1 Background

Currently, the main security measure for password storage is to hash a user's password using a hashing algorithm first before storing the hashed version instead of the plain text version. Other techniques such as salting can also be used to further improve security. Salting involves adding a unique random string of characters to the start or end of each password before hashing it, and then storing the hashed password together with the salt. This process aims to protect against hash and rainbow tables where a malicious actor compares pre-computed hashes or a list of passwords from data breaches in an attempt to find the correct password that corresponds to a hash. The random string added to the password before hashing it will prevent this type of attack as the hash and rainbow tables become unusable without additional processing [2].

SHA, or more specifically the SHA-2 family of hashing algorithms, is a very popular hashing algorithm used everywhere in the industry. It is part of the specifications in several common security protocols such as TLS/SSL and SSH. However, it is usually not recommended for password hashing, as it is designed to be a swift hashing algorithm. This is not ideal in a password hashing context, as it implies that malicious actors are able to compute hashes and crack password hashes much faster than other intentionally slower password hashing algorithms. Still, many have tried to adapt the SHA-2 algorithm for password hashing [5, 9]. I wanted to include this algorithm in this research to compare the differences between a general purpose hashing algorithm and hashing algorithms designed for password hashing to have a better understanding of why SHA-2 - more specifically SHA-512 - may not be the optimal choice here. Since SHA-512 does not have a parameter input normally, I adapted the algorithm by adding a work factor parameter based on paper [9] where the password hashing is chained.

BCrypt is one of the industry go-to options when it comes to password hashing algorithms. BCrypt was first introduced back in 1999 by Niels Provos and David Mazières in their paper "A Future-Adaptable Password Scheme" [17]. The main logic behind it was that, by adapting the Blowfish algorithm it is based on to be more computationally expensive, it would be possible to securely store passwords hashed using their proposed algorithm. Despite being two decades old now, it is still a very popular password hashing algorithm today. Case in point, Auth0 - a widely used authentication service - is a BCrypt supporter [3].

A state-of-the-art password hashing algorithm would be Argon2, the winner of the 2015 Password Hashing Algorithm Competition [6]. It boasts a fine-tuned control over resource costs through the use of three parameters, namely execution time, memory required, and degree of parallelism. Its efficiency and security has been highlighted by OWASP who recommends implementing Argon2id - a derivation of Argon2 made by the same group - for password hashing when possible [14]. This specific derivation will be analysed in this project.

3.2 Current Approaches and Research Gaps

Although many research papers have explored the cost and security of these three hashing algorithms for password hashing either independently or comparatively [4, 9, 11, 12, 15, 19], there is one main criticism with these papers: the cost analysis is not very thorough and the parameter-cost relationship is not adequately explored, if even mentioned. Most papers use an arbitrary set of parameter values and then measures the cost of that specific parameter set without even mentioning their input parameters in their paper [15, 19]. Without specifying the parameter inputs, it is difficult to determine whether the cost is reasonable or not as there is no point of reference. Comparing the cost and security of different hashing algorithms is also confusing as we cannot know if the costs that were measured were based on equivalent parameter values.

Moreover, I noticed a lack of research on less powerful or different architectures such as mobile phones and laptops. In fact, the original Argon2 algorithm was optimized for x86 architecture [6]. Any cost comparison could therefore be significantly impacted by the hardware used, which makes current research unusable in most situations. This research project aims to fill that gap in knowledge and possibly gain insights on the significance of differences in hardware capabilities and architectures, and how each algorithm copes with these differences.

4 IMPLEMENTATION METHOD

The three hashing algorithms that were explored in this research project are SHA-2 (SHA-512 specifically), BCrypt, and Argon2 (Argon2id version 10 specifically). They were implemented in both PC and Android to compare the hardware impact. The time taken to hash the generic plaintext password *password123* was recorded for each parameter combination for each algorithm on each device. An average time was calculated per parameter combination per device and the results were graphed. The overall steps taken to conduct this research project have been outlined in Figure 1.

Note that although analysing the security of the hashes was initially planned for the research project, the security was ultimately not explored for the following reasons:

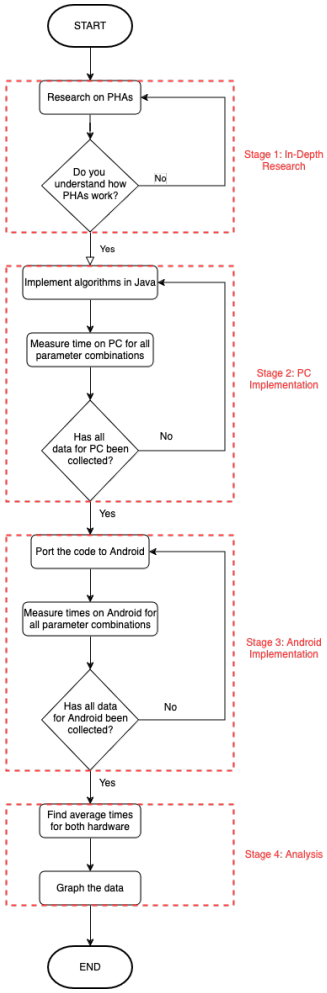

Figure 1: Flowchart for the steps taken to conduct the research. The red areas denote each of the four stages of the project: 1) In-Depth Research, 2) PC Implementation, 2) Android Implementation, and 4) Analysis

- No password dataset included all three of SHA-512, BCrypt, and Argon2id for adequate security testing. Argon2id is also relatively new, so no extensive hash or rainbow table has been compiled yet. Therefore, using different datasets in an attempt to test the security of the hashes by cracking them could be biased.
- Salting was used for BCrypt and Argon2id making any off-the-shelf dataset unusable without significant modifications or a large amount of additional cracking time.
- A high work factor/number of iteration parameter guards against brute force and dictionary attacks, making security analysis insignificant or infeasible to do.

Therefore, using a weak password like *password123* has little to no bearing on the results in this scenario.

4.1 In-Depth Research

The first step - performing in-depth research on the three hashing algorithms to get a general understanding of how they work - did not involve anything of note. I read through the specification papers for BCrypt [17] and Argon2 [6], and found articles online explaining the more complicated aspects of the algorithms. For SHA-512, I read through the lecture slides to refresh my understanding of it.

4.2 PC Implementation

The PC Implementation of the algorithms were done in Java with Gradle using the popular BouncyCastle cryptography library [1]. More specifically, the SHA-512 algorithm was implemented using the *Security* Java class, and BCrypt and Argon2 were implemented using the BouncyCastle library.

I decided not to implement the algorithms from scratch for several reasons. First, the complexity of BCrypt and especially Argon2id was too much for the scope of this project. In fact, Argon2 uses parallelism and complex memory management techniques which are both difficult to implement. Second, I would not be able to use any algorithm optimizations that libraries may have, which could skew the data. Finally, I wanted to replicate a real-world scenario as closely as possible. Those in a professional setting would likely use popular cryptography libraries like BouncyCastle to implement the hashing algorithms rather than implement them from scratch. Consequently, the results of the research would likely be more realistic and usable by others if such libraries were used instead.

The relationship between the parameters and the resource costs of running the algorithm on the plaintext password *password123* was investigated through a quantitative experimental approach on a MacBook laptop. The algorithm parameters were the independent variables and the time taken to hash the password was the measured variable. If the time taken to hash a password with a given parameter combination for an algorithm took longer than 3 minutes, more expensive parameter combinations were not recorded as 3 minutes is already too long to hash a password in a real-world scenario.

4.2.1 SHA-512. As previously mentioned, since SHA-512 normally does not have any parameter inputs, a work factor parameter was introduced by repeatedly chaining the SHA-512 algorithm on the password 2^{wf} times, where wf is the work factor, similar to the work factor parameter in the BCrypt algorithm. The time taken to hash the password for work factor values ranging from 0 to 31 was recorded. The work factor, output hash, and time for each work factor value tested was then stored in a csv file called *"sha512-results.csv"* for analysis. This was repeated three times to create three csv files.

4.2.2 BCrypt. BCrypt also has the work factor parameter as input, and so the time taken to hash the password for work factor values ranging from 4 to 31 was recorded. These numbers are the minimum and maximum possible values for the work factor according to the BouncyCastle documentation for their BCrypt class. A random 16-byte salt was also generated using the *SecureRandom* Java class every time the password was hashed. The work factor, salt, output hash, and time for each work factor value tested was

then stored in a csv file called *"bcrypt-results.csv"* for analysis. This was repeated three times to create three csv files.

4.2.3 Argon2id. Conversely, Argon2id has four parameters: number of iterations, memory limit, length of the output hash, and number of threads (or parallelism). The output hash length and the number of threads parameters were kept constant for all tests at 32 bytes and 1 thread respectively. The 32-bytes hash length was chosen as it is between the hash lengths for the BCrypt and SHA-512 algorithms, and the single thread was chosen to make it comparable to the other two algorithms which only use one thread as well. The memory limit was varied from 10 MiB to 50 MiB inclusive with increments of 10 MiB based on the recommended parameter combinations of OWASP [14]. For each memory limit value, the number of iterations was varied from 4 to 31, the same as for BCrypt. The time taken to hash the password for each of the memory limit and number of iteration parameter combinations was recorded. A random 16-byte salt was also generated using the *SecureRandom* Java class every time the password was hashed. The number of iterations, memory limit, hash length, number of threads, salt, output hash, and time for each number of iterations and memory limit combination tested was then stored in a csv file called *"argon2id-results.csv"* for analysis. This was repeated three times to create three csv files.

4.2.4 Code Organization. The PC implementation was coded using IntelliJ IDEA in Java. A generic file structure for Java projects was used, as seen in Figure 2. The *algorithms* package contains all the hashing algorithm implementation classes *Argon2idAlgo.java, BcryptAlgo.java*, and *SHA512Algo.java*. The *Algorithm.java* class in the *algorithms* package is an interface class that each algorithm class implements, and has two required methods: a parameter input method called *getInputParams()* for the manual one-time tests, and a method called *hashPassword()* that processes and returns the output hash of the plaintext password as a byte array. *ResearchProject.java* is the main runner class for the terminal-based menu and also contains the code for the manual one-time tests. *ResultGenerator.java* has methods called *generateSHA512Results(), generateBCryptResults(), and generateArgon2idResults()* to automatically record the times for each parameter combination for SHA-512, BCrypt, and Argon2id respectively, as described earlier. *Utils.java* contains two utility methods to convert a bytes array to a hex string (*convertBytesToHex()*) and to generate a random 16-byte salt with *SecureRandom* (*generateSalt16Bytes()*). The project can be found at this GitHub repository [7].

4.2.5 Compiling and Running. Although the program was developed on a MacBook laptop, since Java with Gradle was used, the program should be able to run on any computer, as long as the correct JDK and Gradle version is installed (see Section 5 for version details). To compile and run the program, execute the command [`gradle --console plain clean build run`]. The [`--console plain`] flag is used to remove unnecessary Gradle prompts that otherwise would pollute the terminal output.

4.2.6 Sample Output. When the program is running, a terminal will appear with 4 possible options to choose from. Selecting one of the first three options will generate the respective csv file mentioned earlier for the SHA-512, BCrypt, and Argon2id algorithms

```
app/
├─ src/main/java/code/
│  ├─ algorithms/
│  │  ├─ Algorithm.java
│  │  ├─ Argon2idAlgo.java
│  │  ├─ BcryptAlgo.java
│  │  ├─ SHA512Algo.java
│  ├─ ResearchProject.java
│  ├─ ResultGenerator.java
│  ├─ Utils.java
├─ build.gradle
```

Figure 2: The file structure for the PC implementation of the research project.

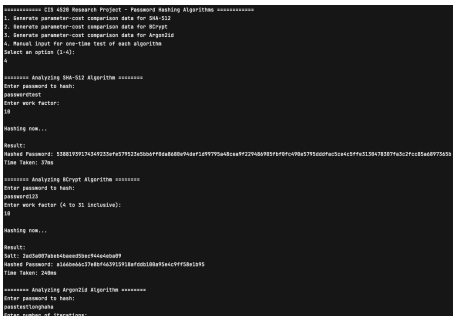

Figure 3: Sample terminal output when selecting the manual one-time test for the PC implementation.

respectively. Selecting the fourth option will start a manual one-time test for each of the three algorithms in order, where the user can input a plaintext password and the work factor or number of iterations parameter for the algorithm, and the salt (if any), output hash and time taken is displayed (see Figure 3).

4.3 Android Implementation

After the PC implementation, I ported the code to Android using Android Studio so that I could run the tests again on a mobile device with significantly worse hardware capabilities compared to a MacBook laptop. The Android implementation was somewhat simple as it uses the PC implementation as a foundation. First, I learned how to work with Android and Android Studio using this guide [13]. I used the Basic Activity template given by Android Studio as a foundation. The *algorithms* package and the *ResultGenerator.java* and *Utils.java* files from the PC implementation were copied over to the android implementation repository as well to use the same code to ensure the results are not biased. Then, the same test procedure as in the PC implementation were run again on an Android mobile device. Doing so will create another set of three csv files for each algorithm. The two sets of nine csv files (3 algorithms * 3 repeats/algorithm) can then be compared and analysed.

4.3.1 Code Organization. Figure 4 shows the file structure of the repository for the Android implementation. Note that only the important files and directories are shown. The entire *code* package contains all the classes that were copied over from the PC

```
app/
├─ src/main/
│  ├─ java/com/example/cis4520researchprojectphone/
│  │  ├─ code/
│  │  │  ├─ algorithms/
│  │  │  │  ├─ Algorithm.java
│  │  │  │  ├─ Argon2idAlgo.java
│  │  │  │  ├─ BcryptAlgo.java
│  │  │  │  ├─ SHA512Algo.java
│  │  │  ├─ ResultGenerator.java
│  │  │  ├─ Utils.java
│  │  ├─ ProjectDisplayFragment.java
│  │  ├─ ResearchProject.java
│  ├─ res/
│  │  ├─ layout/
│  │  │  ├─ project_display_fragment.xml
│  │  ├─ values/
│  │  │  ├─ colors.xml
│  │  │  ├─ string.xml
├─ build.gradle
```

Figure 4: The file structure for the Android implementation of this research project. Note that only important files and directories have been included in this diagram.

implementation. The *ProjectDisplayFragment.java* class and the *project_display_fragment.xml* file represent the logic and the layout for the UI of the Android application respectively. *ResearchProject.java* is the main class for the project and contains boilerplate code that will run the application. The *colors.xml* and *string.xml* files are resource files that contain the color codes used for the UI components and the text displayed in the application respectively. The project can be found at this GitHub repository [8]

4.3.2 Compiling and Running. Although this was produced and tested on a MacBook laptop, you can run the project with Android Studio and an Android device. If you do not have a physical Android device, Android Studio can emulate one for you. Once the project is ready, press the *Run App* button in the top right (the green "play" button) to start the application on the selected device. It will open in the *Running Devices* tab on the right sidebar where the screen of your device will be displayed. See guide [13] for more details on how to run the application.

4.3.3 Sample Output. A simple UI will be displayed once the application is running (see Figure 5). A text information section is on display in the middle of the screen, and three purple buttons labeled "SHA-512", "BCrypt", and "Argon2id" can be seen below the text. These buttons each correspond to generating the test results for the specified algorithm, the same as in the PC implementation. For example, if the "BCrypt" button is pressed, it will generate the csv file for the BCrypt result data. The generated file can be found using the *Device Explorer* tab in the right sidebar at */data/data/com.example.cis4520researchprojectphone/files*. Note that the manual one-time test is not implemented here.

4.4 Analysis

Now that the 18 csv files have been generated, an average time for each parameter combination for each algorithm for each device was calculated using the three csv files for each algorithm and the unnecessary data for the graphs such as the output hash was removed with Python. The processed data is then stored in another csv file called *"<algorithm>-results-processed.csv"*, where <algorithm> is the name of the relevant algorithm. The processed csv files for each device was then manually merged together into a single file called *"<algorithm>-results-processed-both.csv"* for each algorithm. Then, these files were graphed using Python with Pandas, Seaborn and

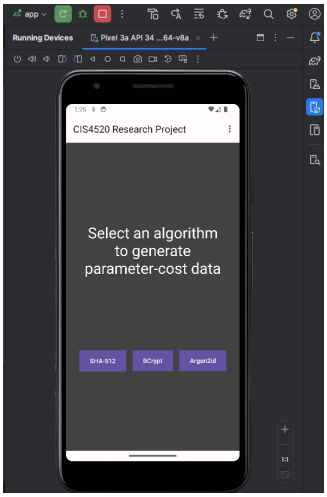

Figure 5: Sample view of how the app will be displayed on Android Studio. A Pixel 3a device was emulated here.

Matplotlib. The parameters against the time for both devices for each algorithm was graphed to see the relationship between the parameter and the cost (i.e. time) as well as the difference in cost between the Android device and the MacBook laptop. Then, a 1s and 0.5s threshold line was drawn to determine the maximum parameter combination that is still within these thresholds. These values were chosen as they represent the acceptable time a server may take to hash a password. The graphs were compared to gain insight on the data (see Section 6 for the results).

4.4.1 Code Organization. In the same repository as the PC implementation, there is also a *graphs* directory that contains all the csv files and Python scripts used in the project. The *data* directory contains all the csv files. Note that the directory structure is important and the Python scripts will not work if they are not followed. *process-data.py* takes the raw csv data from the tests and generates the processed csv files for each algorithm. *plot-graph.py* plots the processed data for the algorithms into relevant graphs. See the README.md in [7] for more details.

4.4.2 Running. VSCode and Python3.9 was used here. For *process-data.py*, run the the following command to process the data:

```
python3 process-data.py <hardware> <algorithm>
```

<hardware> represents the hardware device to process ("pc" or "android") and <algorithm> is the name of the algorithm to process ("sha512", "bcrypt", or "argon2id"). For *plot-graph.py*, run the following command to create the graphs:

```
python3 plot-graph.py <datafile path> <algorithm>
<graph title> <output filename>
```

<datafile path> represents the path of the data file to be used to create the graph. <graph title> is the title of the graph that the user chooses. It should be within double quotes if there are spaces in the title. (For eg: "SHA-512 Parameter-Cost Comparison"). <output filename> is the name of the output file for the graph. Specify the file type in the filename as well (For eg: *testgraph.png* will make the graph into a png file called *testgraph.png*)

5 EXPERIMENTAL SETUP

Below are the specs of the hardware, software, and libraries used.

5.1 Hardware

- **MacBook Laptop**
  - *Device Type*: MacBook Pro 13-inch, 2020
  - *Processor*: Apple M1 (ARM Architecture)
  - *RAM*: 16GB
  - *OS*: Sonoma 14.3.1
- **Android Mobile Device**
  - *Device Type*: Samsung Galaxy A52 5G
  - *Processor*: Octa-core (2x2.2 GHz Kryo 570 & 6x1.8 GHz Kryo 570)
  - *Chipset*: Qualcomm SM7225 Snapdragon 750G 5G (8 nm)
  - *RAM*: 6GB
  - *OS*: Android 14

5.2 Software

- **IntelliJ IDEA 2023.3.4 (Community Edition)**
- **Android Studio Iguana | 2023.2.1 Patch 1**
- **Visual Studio Code Version 1.88.0**

5.3 Languages and Libraries

- **Java**
  - Version 8: For Android implementation (newer versions incompatible with Android SDK)
  - Version 18: For PC implementation
- **Gradle Version 7.5.1**
- **BouncyCastle Version 1.77 (from jdk18on)**
- **Android API (minSDK: 24, targetSDK: 34)**
- **Python 3.9.16**
  - Pandas, Seaborn, Matplotlib, and CSV libraries

6 RESULTS

An analysis of each of the algorithms based on the parameter-cost graphs and data has been detailed below.

6.1 SHA-512 Analysis

From Figure 6, we see that the cost of both devices increase exponentially for the SHA-512 algorithm, with similar growth between 0 and 15 work factor. However, Android increases significantly faster than PC after more than 15 work factor and this is when excessive growth starts for both devices. The 3-minute cut-off point is vastly different for Android (wf=26,t=264,222.67ms) and PC (wf=30,t=249,080.67ms). These results show that there is a significant hardware impact on the cost of SHA-512 at work factors larger than 15, as denoted by the large difference in cut-off points. Moreover, depending on the hardware capabilities of the system, work factors greater than 15 are most likely excessive and should not be considered.

Figure 7 shows the same graph for SHA-512 as Figure 6, but with a different scale and the 1s and 0.5s thresholds have been marked by the red horizontal lines in the graphs. There is a significant difference between the maximum work factors at each threshold for the two devices (PC: wf=21,t=461.33ms; wf=22,t=890.67ms | Android: wf=15,t=363.67ms; wf=16,t=689.67ms). This further shows

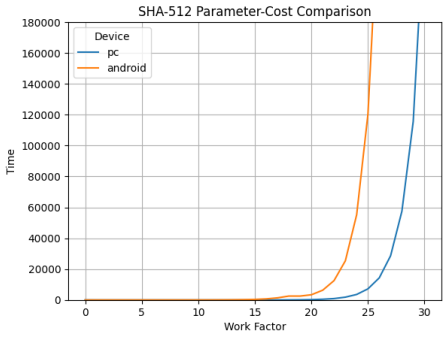

Figure 6: Parameter-cost graph for the SHA-512 algorithm. Axes represent Work Factor against Time (in ms).

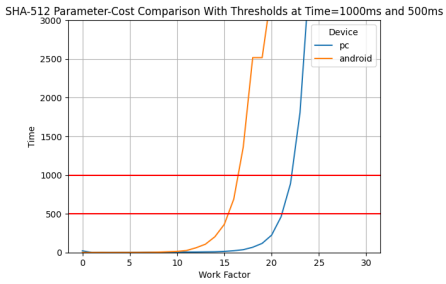

Figure 7: Scaled parameter-cost graph for the SHA-512 algorithm. Axes represent Work Factor against Time (in ms). Red horizontal lines denote the 1000ms and 500ms time thresholds.

that there is a significant hardware impact on the SHA-512 algorithm costs for the same parameter values. Additionally, the exponential increase of the cost makes it difficult to reach each threshold exactly so there is "wasted" security potential. For example, for the 1s threshold for PC, it is reached with a maximum of 22 work factor. However, the time taken for this work factor value is 890.67ms. Increasing the work factor would go over the threshold and therefore is not usable. The 110ms difference between the actual cost and the threshold denotes that there is additional security through increased cost that cannot be utilized properly.

6.2 BCrypt Analysis

Figure 8 shows that the cost of BCrypt for both devices increase exponentially, with similar growth between 0 and 10 work factor. However, Android increases slightly faster than PC after more than 10 work factor and excessive growth for both devices starts at around 12 work factor. The 3-minute cut-off is relatively similar for Android (wf=21,t=233,537.33ms) and PC (wf=22,t=301,768ms). Contrary to the results for SHA-512, these results show that there is not a significant hardware impact on the cost of BCrypt, as denoted by the relatively small difference in cut-off points. Moreover, depending on the hardware capabilities of the system, work factors greater than 12 are most likely excessive and should not be considered.

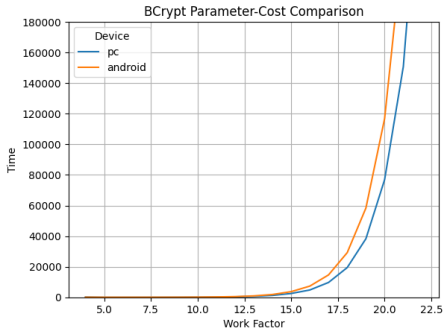

Figure 8: Parameter-cost graph for the BCrypt algorithm. Axes represent Work Factor against Time (in ms).

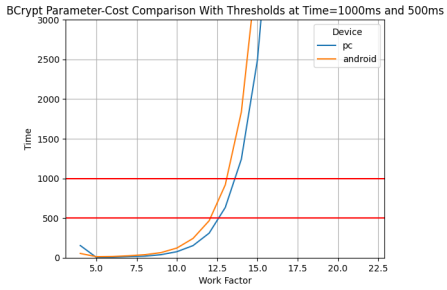

Figure 9: Scaled parameter-cost graph for the BCrypt algorithm. Axes represent Work Factor against Time (in ms). Red horizontal lines denote the 1000ms and 500ms time thresholds.

Figure 9 shows the same graph for BCrypt as Figure 8, but with a different scale and the 1s and 0.5s thresholds have been marked by the red horizontal lines in the graphs. Again, unlike with SHA-512, there is no significant difference between the maximum work factors at each threshold for the two devices (PC: wf=12,t=309.33ms; wf=13,t=632ms | Android: wf=12,t=465.67ms; wf=13,t=916.67ms). This further demonstrates that the hardware has little impact on the cost of BCrypt hashes. Still, the exponential increase of the cost makes it difficult to reach each threshold exactly so there is still the "wasted" security potential. For example, for the 1s threshold for PC, it is reached with a maximum of 13 work factor yet the time taken is 632ms. The 368ms difference is additional security through increased cost that cannot be utilized properly.

6.3 Argon2id Analysis

Conversely, Figure 10 reveals that the cost of Argon2id for both devices increases reasonably linearly. However, Android is significantly slower than PC in all cases. Note that Argon2id is still significantly faster than SHA-512 and BCrypt at similar work factors/number of iterations. The 3-minute cut-off is not reached for either of the two devices at 31 iterations. Similar to the results for SHA-512, these results show that there is a significant hardware impact on the cost of Argon2id. Moreover, the increasingly large

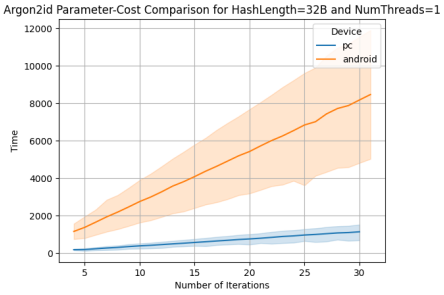

Figure 10: Parameter-cost graph for the Argon2id algorithm with HashLength=32B and NumThreads=1. Axes represent Number of Iterations against Time (in ms). Shaded areas denote ranges for 10MiB up to 50MiB memory limits, with solid line representing mean line for each device.

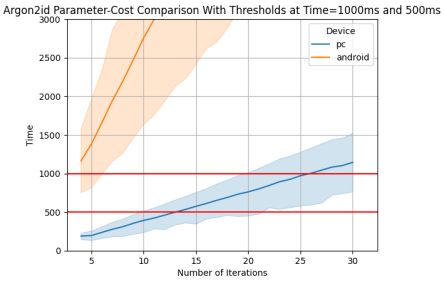

Figure 11: Scaled parameter-cost graph for the Argon2id algorithm with HashLength=32B and NumThreads=1. Axes represent Number of Iterations against Time (in ms). Shaded areas denote ranges for 10MiB up to 50MiB memory limits. Red horizontal lines denote the 1000ms and 500ms time thresholds.

variance in time between memory limits for Android as the number of iterations increases as shown by the increasing shaded area further exemplifies the impact of hardware on the Argon2id algorithm. The larger the memory limit, the faster the increase in cost of the hash, with the 50MiB memory limit increasing the fastest. Comparatively, there is a relatively small variance for PC and the shaded area remains reasonably constant.

Figure 11 also reveals some interesting insights on the Argon2id algorithm. There are completely different parameter combinations that are within each threshold. The 1s threshold is reached at most of the parameter combinations for PC (n=22,ml=30720KiB, t=951.5ms; n=16,ml=40960KiB, t=954.5ms; n=21,ml=51200KiB, t=949.75ms) but only few parameter combinations were within the 1s threshold for Android (n=11,ml=10240KiB, t=946.67ms; n=5,ml=20480KiB, t=866.33ms). For the 0.5s threshold, none of the parameter combinations were usable for Android but some were within the threshold for PC (n=17,ml=20480KiB, t=485.5ms; n=11,ml=30720KiB, t=486ms; n=8, ml=40960KiB, t=491.5ms; n=11,ml=51200KiB, t=493.75ms). It appears that Argon2id needs a minimum hardware requirement to be viable, but once reached it is relatively efficient and stable.

6.4 Summary

The three algorithms analysed in this project (SHA-512, BCrypt, and Argon2id) are all viable password hashing algorithms if they are properly implemented and configured. Fine-tuning the parameters for maximum security and lowest cost is more difficult because of the exponential work factor parameter in SHA-512 and BCrypt, but it is easier with the linear parameters used in Argon2id. In terms of hardware impact, BCrypt and Argon2id are relatively stable against hardware differences, but SHA-512 is highly affected by hardware differences. Consequently, SHA-512 benefits the most from a hardware upgrade, Argon2id benefits moderately, while BCrypt barely benefits from a hardware upgrade.

7 CONCLUSION

Through this research project, a thorough and in-depth analysis of the effect of parameter input on the cost to hash a password based on the time taken was made. The analysis demonstrates that each of the three hashing algorithms are viable for password hashing, and the hardware impact on the cost of the algorithm varies significantly depending on the algorithm, with BCrypt being the least impacted and SHA-512 being the most impacted. If a hardware upgrade is not planned in the near future or within budget, using a stable algorithm like BCrypt that is not significantly affected by hardware could be a good choice. On the other hand, if a future hardware upgrade is possible or planned within the near future, an algorithm that is significantly impacted by hardware differences like Argon2id could be best in this scenario. Still, the end goal is to use parameter combinations for any of the algorithms with the highest cost that is within the required performance metrics. Do note though that OWASP indicates to avoid parameters with excessive costs as Denial of Service attacks become increasingly easier to execute as cost increases [14].

This research project has laid the groundwork for future comparisons between other or newer password hashing algorithms, and has also provided practical guidelines on choosing a suitable password hashing algorithm and its parameters.

8 OPEN PROBLEMS

While increasing cost to improve security is both a simple and easy solution to protect passwords, there are many other factors that influence password storage security. For example, a regular server may not be able to keep up with hardware dedicated for password cracking such as GPUs or even distributed computing, so there is a limit to the amount of cost that can be increased. With quantum computing as well, its effect on password cracking is still undetermined, but it is likely to have a significant effect. Instead of using passwords, zero-knowledge proofs may allow users to identify themselves without needing to transmit their passwords, resulting in near-certain security [18]. However, such solutions are still a long way from becoming the norm, so until then, we need to ensure that the traditional password system stays as secure as possible.

9 FUTURE WORKS

There are still gaps in current literature on hardware and password hashing algorithms. Possible future works on this topic include exploring the hardware impact on the password hashing algorithms on various types of hardware devices, such as tablets, servers, or even cloud infrastructure. Another possible future work involves redoing this research but instead analysing and comparing other hashing algorithms such as MD-5, SHA-1 and SHA-3, PBKDF2 and SCrypt.

REFERENCES

[1] [n. d.]. ([n. d.]). https://www.bouncycastle.org/

[2] Dan Arias. 2021. Adding Salt to Hashing: A Better Way to Store Passwords. (Feb 2021). https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/

[3] Dan Arias. 2021. Hashing in Action: Understanding bcrypt. (2 2021). https://auth0.com/blog/hashing-in-action-understanding-bcrypt/

[4] Toras Pangidoan Batubara, Syahril Efendi, and Erna Budhiarti Nababan. 2021. Analysis Performance BCRYPT Algorithm to Improve Password Security from Brute Force. *Journal of Physics: Conference Series* 1811, 1 (mar 2021), 012129.

[5] Philip Joseph F. Bemida, Ariel M. Sison, and Ruji P. Medina. 2021. Modified SHA-512 Algorithm for Secured Password Hashing. In *2021 Innovations in Power and Advanced Computing Technologies (i-PACT)*. 1–9. https://doi.org/10.1109/i-PACT52855.2021.9696928

[6] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. 2016. Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications. In *2016 IEEE European Symposium on Security and Privacy (EuroSP)*. 292–302. https://doi.org/10.1109/EuroSP.2016.31

[7] Ronan Chay Loong. 2024. cis4520-research-project. (2024). https://github.com/RonanChay/cis4520-research-project

[8] Ronan Chay Loong. 2024. cis4520-research-project-android. (2024). https://github.com/RonanChay/cis4520-research-project-android

[9] Savarala Chethana, Sreevathsa Sree Charan, Vemula Srihitha, D. Radha, and C. R. Kavitha. 2022. Comparative Analysis of Password Storage Security using Double Secure Hash Algorithm. In *2022 IEEE North Karnataka Subsection Flagship International Conference (NKCon)*. 1–5. https://doi.org/10.1109/NKCon56289.2022.10127057

[10] IBM Corporation. 2023. Cost of a Data Breach Report 2023. (7 2023). https://www.ibm.com/reports/data-breach

[11] Levent Ertaul, Manpreet Kaur, and Venkata Arun Kumar R Gudise. 2016. *Implementation and Performance Analysis of PBKDF2, Bcrypt, Scrypt Algorithms* (2016). https://docslib.org/doc/412065/implementation-and-performance-analysis-of-pbkdf2-bcrypt-scrypt-algorithms

[12] Siwoo Eum, Hyunjun Kim, Minho Song, and Hwajeong Seo. 2023. Optimized Implementation of Argon2 Utilizing the Graphics Processing Unit. *Applied Sciences* 13, 16 (2023).

[13] lmf. 2023. Build your first Android app in Java. (Sep 2023). https://developer.android.com/codelabs/build-your-first-android-app#0

[14] OWASP. 2024. Password Storage Cheat Sheet. (2024). https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html [Accessed 26-01-2024].

[15] Rohan Patra and Sandip Patra. 2021. Cryptography: A quantitative analysis of the effectiveness of various password storage techniques. *Journal of Student Research* 10, 3 (Oct 2021). https://doi.org/10.47611/jsrhs.v10i3.1764

[16] Ani Petrosyan. 2023. Annual number of data compromises and individuals impacted in the United States from 2005 to 2022. (8 2023). https://www.statista.com/statistics/273550/data-breaches-recorded-in-the-united-states-by-number-of-breaches-and-records-exposed/

[17] Niels Provos and David Mazieres. 1999. A future-adaptable password scheme.. In *USENIX Annual Technical Conference, FREENIX Track*, Vol. 1999. 81–91.

[18] Ameer Rosic. 2023. Demystifying zero knowledge proofs: A comprehensive guide. (Nov 2023). https://blockgeeks.com/guides/zero-knowledge-proofs/

[19] C. Skanda, B. Srivatsa, and B.S. Premananda. 2022. Secure Hashing using BCrypt for Cryptographic Applications. In *2022 IEEE North Karnataka Subsection Flagship International Conference (NKCon)*. 1–5. https://doi.org/10.1109/NKCon56289.2022.10126956