Clipaha: A Scheme to Perform Password Stretching on the Client

Downloaded from: https://research.chalmers.se, 2026-04-04 13:09 UTC

Citation for the original published paper (version of record):

Izquierdo Riera, F., Almgren, M., Picazo-Sanchez, P. et al (2023). Clipaha: A Scheme to Perform Password Stretching on the Client. Proceedings of the 9th International Conference on Information Systems Security and Privacy: 58-69. http://dx.doi.org/10.5220/0011653200003405

N.B. When citing this work, cite the original published paper.

research.chalmers.se offers the possibility of retrieving research publications produced at Chalmers University of Technology. It covers all kind of research output: articles, dissertations, conference papers, reports etc. since 2004. research.chalmers.se is administrated and maintained by Chalmers Library

(article starts on next page)

Clipaha: A Scheme to Perform Password Stretching on the Client

Francisco Blas Izquierdo Riera[1][a], Magnus Almgren[1][b], Pablo Picazo-Sanchez[2][c] and Christian Rohner[3][d]

[1]*Chalmers University of Technology, 412 96 Göteborg, Sweden*

[2]*School of Information Technology, Halmstad University, Sweden*

[3]*Uppsala University, Box 534, 751 21 Uppsala, Sweden*

Keywords: Password Stretching, Password-based Authentication, IoT Security, Server Relief, Web Security, Argon2.

Abstract: Password security relies heavily on the choice of password by the user but also on the one-way hash functions used to protect stored passwords. To compensate for the increased computing power of attackers, modern password hash functions like Argon2, have been made more complex in terms of computational power and memory requirements. Nowadays, the computation of such hash functions is performed usually by the server (or authenticator) instead of the client. Therefore, constrained Internet of Things devices cannot use such functions when authenticating users. Additionally, the load of computing such functions may expose servers to denial of service attacks. In this work, we discuss client-side hashing as an alternative. We propose Clipaha, a client-side hashing scheme that allows using high-security password hashing even on highly constrained server devices. Clipaha is robust to a broader range of attacks compared to previous work and covers important and complex usage scenarios. Our evaluation discusses critical aspects involved in client-side hashing. We also provide an implementation of Clipaha in the form of a web library[1] and benchmark the library on different systems to understand its mixed JavaScript and WebAssembly approach's limitations. Benchmarks show that our library is 50% faster than similar libraries and can run on some devices where previous work fails.

SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

Despite being around for sixty years and having many issues, passwords are still a prevalent authentication method (Corbató, 1963; Bauman et al., 2015; Van Acker et al., 2017). Conceptually, a client provides its identity (username u) and a secret (password p) which is matched against a database entry on the server to be accessed. Getting access to passwords has ever since been a target for attacks (Lee et al., 1992). Because users tend to reuse the same password on different systems, a breach in one system can have consequences over a much larger scope.

In particular, with the advent of the Internet of Things (IoT), where potentially every embedded low-power device acts as a server, the issues with passwords have become a dilemma. As IoT devices are usually more vulnerable, getting access to their password database is relatively easy. Also, their ability to handle password authentication is significantly hindered compared to traditional client/server systems due to limited computation, energy and amounts of memory available for code and data.

Nowadays, database breaches do not reveal the password per se since stored passwords are protected by a secure one-way function $f(.)$ (e.g., a cryptographic hash). $f(.)$ includes a random salt s prepended to the password to make the password unique and avoid that equal passwords are easily recognized during dictionary attacks. Since the database stores the triplet $\langle u, s, f(s||p)\rangle$, the hash function has to be computed at every authentication. To prevent dictionary attacks from being accelerated after a breach using powerful GPU, FPGA, or ASIC implementations; password hashing functions have increased their complexity.

Modern password hashing algorithms (e.g., Argon2 (Biryukov et al., 2017)) are specifically designed to eliminate the possible attacker advantage gained from accelerators. Removing such advantage comes at the cost of excessive computation and memory usage

[a] https://orcid.org/0000-0003-1509-142X
[b] https://orcid.org/0000-0002-3383-9617
[c] https://orcid.org/0000-0002-0303-3858
[d] https://orcid.org/0000-0002-1527-734X

Izquierdo Riera, F., Almgren, M., Picazo-Sanchez, P. and Rohner, C.
Clipaha: A Scheme to Perform Password Stretching on the Client.
DOI: 10.5220/0011653200003405
In *Proceedings of the 9th International Conference on Information Systems Security and Privacy (ICISSP 2023)*, pages 58-69
ISBN: 978-989-758-624-8; ISSN: 2184-4356
Copyright © 2023 by SCITEPRESS – Science and Technology Publications, Lda. Under CC license (CC BY-NC-ND 4.0)

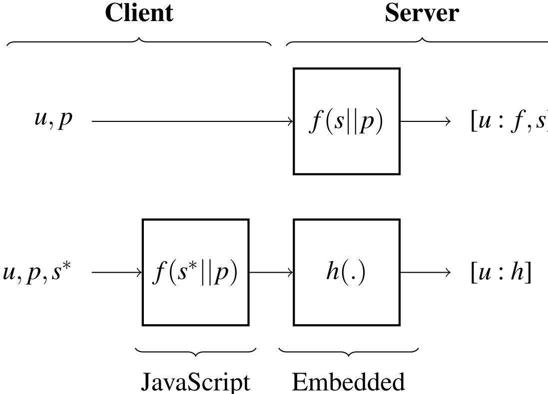

Figure 1: Clipaha moves the resource-intensive hash function $f(s||p)$ to the client. The salt s^* is a deterministic string derived from the username u and the server's domain name.

(e.g., the specification recommends 1 GiB memory and 0.5 s runtime for front-end authentication (Biryukov et al., 2016)). However, there is still a lack of use of such algorithms with strong security parameters on regular systems and more so on resource-constrained systems (Eliasen, 2019; Ntantogian et al., 2019).

Currently, most devices perform the password hashing step in the server (or authenticator) (Blanchard et al., 2019). This prevents low-power IoT devices lacking either the memory or the power for running the CPU long enough from using modern hash functions with strong hash parameters. The huge resource demand may also open non IoT devices to Denial of Service attacks by allowing resource exhaustion through multiple login attempts.

***Clipaha*.** To address the security versus complexity challenge in highly-constrained devices, we propose moving the resource-hungry password hashing step to the client. We propose *Clipaha*, a Client-side Password Hashing scheme, based on Argon2, that allows user authentication in highly-constrained devices. Figure 1 illustrates the concept of Clipaha in comparison to server-side hashing. By moving the resource intensive operation $f(s^*||p)$ to the client, the server only needs to compute a low cost one-way hash function $h(.)$ (e.g., SHA-256) to prevent that leaked database entries can directly be used to access the server. We provide an implementation of Clipaha in the form of a web library[1] and benchmark it on different systems to understand the limitations that its mixed JavaScript and WebAssembly implementation faces. Benchmarks show that our library is 50% faster than similar libraries and can even run on some devices where others fail. We also provide a proof-of-concept of the server-side of the code for the ESP8266,[1] an embedded wireless device popular on Do It Yourself (DIY) environments featuring an 80 MHz processor and 80 KiB of memory for user data (Espressif Systems, 2020), and show that also constrained server devices can benefit from modern password hashing algorithms with their increased security against brute force attacks.

[1]Artifact available at: https://github.com/clipaha/clipaha/releases/tag/ICISSP_2023.

Contributions. We summarize our contributions as follows:

- We present Clipaha, a Client-side Password Hashing scheme that does not compromise on the security compared with server-side approaches and allows stronger user authentication in highly-constrained devices. Clipaha is robust against a wider range of attacks than existing client-side approaches.
- We implement, test, and publicly release an implementation of Clipaha in the form of a Javascript library.[1] Our prototype is 50% faster than existing approaches.
- We thoroughly evaluate Clipaha and demonstrate, based on a set of eight scenarios, that it is ready to be deployed in practical settings.

The rest of the paper is structured as follows. Section 2 covers the related work. Section 3 introduces the adversarial model. We present the objectives and motivation for Clipaha in Section 4. Section 5 describes the implementation while in Section 6 we evaluate the library. Finally, Section 7 concludes the paper.

2 BACKGROUND AND RELATED WORK

Modern password hashing schemes use three techniques to slow down attackers. First, they combine a unique string, i.e., the salt, with the password before using a one-way function (e.g., a cryptographic hash) to make each password entry unique so that offline attackers performing a dictionary attack have to execute the hash once per guess and entry (Wilkes, 1968). Second, significant processing power is used to ensure each hash execution is slow to execute (Morris and Thompson, 1979). Third, significant amounts of memory space and bandwidth are utilized in a way that prevents efficient use of GPUs, FPGAs, or ASICs to compute the hash (Hatzivasilis, 2017).

In 2013, Argon2 (Biryukov et al., 2016) won a competition aiming to modernize one-way functions for hashing passwords concerning new attack methodologies (Aumasson et al., 2013). Argon2 has two main variants, Argon2d and Argon2i, and various supplementary variants, including Argon2id.

Table 1: Impact of security and functional issues over crackstation, libsodium and Clipaha's approaches. ✓ means the approach is not affected; ✗ means it is, and; (✓) means it is partially affected.

	Salt Coll.	User Enum.	Ref. Lib.
Crackstation	✗	✓	✗
Libsodium	✓	✗	(✓)
Clipaha	✓	✓	✓[1]

Argon2d uses data-dependent memory accesses, whereas Argon2i uses data-independent memory accesses. The consequence is that Argon2d is vulnerable to side-channel attacks but offers the best resistance to Time-Memory Trade-Offs (TMTOs), while Argon2i is secure against side-channel timing attacks but is weaker against TMTOs (Alwen and Blocki, 2017; Biryukov et al., 2017).

Argon2id works as Argon2i for the first half of the first iteration over the memory and as Argon2d for the rest, thus providing both protection against side-channel attacks and against cost savings due to TMTOs (Biryukov et al., 2017).

Inherent to Argon2, and modern password hashing functions in general, is that one can parametrize the "hardness" of the hash function in different ways to make it more resistant to attacks, e.g., by specifying the needed memory and iterations to create the hash. In particular, Argon2 has various input parameters relevant to our use: password, salt, memory required, execution time (iterations), the maximum degree of parallelism allowed (lanes), and hash length.

The high memory and execution time parameters needed to deter attackers using powerful systems entail that constrained IoT systems cannot run the algorithm natively. A possible solution is using *server relief*, one of the evaluation criteria of PHC, so that the server can offload the computation to the client. Thus, more constrained systems can benefit from these algorithms. However, server relief is not commonly deployed (Blanchard et al., 2019).

There are two mechanisms to implement server relief: crackstation (Security, 2019) and libsodium (Denis, 2019). Crackstation's approach (Security, 2019) does not recommend separating nor distinguishing the parameters used to generate the hash's salt, which could result in the same salt value being used for different accounts on different sites. In Clipaha we ensure a unique salt is used; we provide a comparison with Clipaha in Table 1.

Libsodium's approach (Denis, 2019) has three main issues. First, compared to Crackstation's and Clipaha, it introduces an additional step (and the corresponding round trip time) to obtain the user's salt. Second, it is prone to user enumeration attacks. For example, an attacker can detect a registration by noticing the change in the salt belonging to the user. Furthermore, when following the recommendations, an attacker may also guess whether users are registered on systems where usernames are canonicalized (converted into a standard representation) when the pseudorandom salt remains the same for different inputs resulting in the same canonical username. In Clipaha, we remove the risk for user enumeration. Third, libsodium only provides the tools to perform the authentication flow but has no recommendations on parameter values. We provide a user-friendly API with preset parameters and a security analysis of several use cases. We have also compared the client performance of Clipaha with Libsodium and can show that the former is 25%–50% faster.

Aside from modern password hashes, other techniques to protect password-based authentication exist, such as Multi-Factor Authentication (MFA) and Password-Authenticated Key Exchange (PAKE). MFA uses factors other than the user's knowledge of a password to protect an account. Similarly, PAKE is complementary to Clipaha in that it protects against an attacker sniffing the password in transit or impersonating the authentication server. Clipaha is compatible with MFA and PAKE and can be combined with them (see Subsection 6.5).

3 THREAT MODEL

In our adversarial model, the attackers' primary objective is obtaining valid user credentials. This would give them access to the system and the ability to move laterally to other systems where the credentials have been reused. In most cases, attackers also have as an objective to access the service without authorization.

Our model considers both online attackers and offline attackers. Online attackers have access to the login system (Ntantogian et al., 2019). Offline attackers have read access to the entire database of users' password hashes (Bai and Blocki, 2021). They may also have write access to the database or access to the service. However, in this case, their objective is not accessing the service, which would be trivial, but recovering the original credentials. We explain below the concrete attacks these attackers can perform.

In *credential bruteforcing*, online attackers make requests to the server to find a valid <*user, password*> pair (Saad and Mitchell, 2020). Offline attackers have full read access to the user database, including their hashed passwords, and therefore can utilize optimizations, e.g., TMTO techniques like rainbow tables

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

(Oechslin, 2003). This is particularly relevant because it allows them to exploit salt collisions. In general, the effort for successful credential bruteforcing depends linearly on the computational cost of the hash function and exponentially on the entropy of the passwords.

In *salt collisions*, two or more passwords are hashed using the same salt, hash function, and hash parameters. Such collisions are problematic because attackers can test all colliding accounts performing only one hash operation per guess. This cost reduction makes affected accounts more likely to be targeted by attackers who try to minimize the economic costs of their attack. Also, salt collisions allow an attacker to see if various accounts have the same password since the resulting hash will be the same in that case.

In *user enumeration*, attackers check for the existence of users in the system by directly querying the targeted system or detecting differences in responses (Skowron et al., 2021).

Client-side approaches, including server relief, introduce the risk for so-called *pass-the-hash attacks*. In a pass-the-hash attack, the (offline) attacker tries to use the already hashed credential stored in the breached database. Such an attack would work not only on the site but also on other sites with the same salt, password, and hash parameters.

4 DESIGN OF *Clipaha*

In this section, we discuss the requirements behind our work and then present the design of Clipaha, a Client-side Password Hashing scheme that allows user authentication in highly-constrained devices.

4.1 Objectives

In Table 2, we list the main objectives of Clipaha. We sorted them into three categories: security-related, functional, and performance-related. The first three security-related objectives are related to the threat model: guarantee password hashes are unique across accounts and systems (RQ-S1), ensure server behavior does not allow to find if users are present or not (RQ-S2), and make sure the hash stored in the database cannot be used directly as input for authentication (RQ-S3). As for credential bruteforcing, we assume that an attacker cannot circumvent the limitations imposed by the hashing algorithm but is not limited otherwise.

To guarantee overall security, the algorithm needs to be at least as secure as if performed by a server (RQ-S4). This objective implies that we consider a use case with constrained servers but sufficiently powerful clients. This aspect is closely related to the

Table 2: The security (S), functional (F) and performance (P) objectives for Clipaha.

Label	Description
RQ-S1	Resistance to salt collisions
RQ-S2	Resistance to user enumeration
RQ-S3	Resistance to pass-the-hash attacks
RQ-S4	Ensure the algorithm is at least as secure as if performed by the server
RQ-F1	Not require software developers to choose security relevant parameters
RQ-F2	Suitable for the many contexts where passwords are used
RQ-P1	Limit the delay of the authentication process for legitimate users

Algorithm 1: Clipaha's client-side password hashing process.

function CLIPAHAHASH(Domain, Username, Password)
  NUsername ← CANONICALIZE(Username)
  Salt ← DELIMIT(Domain, NUsername)
  return PASSWORDHASH(Salt, Password)
end function

performance-related objective that the adopted library must keep a reasonable delay (RQ-P1) no matter which type of client device they use.

To increase the use and deployment of modern password hashing techniques without putting security at risk, we need to provide a reference library which is easy-to-use for software developers who may not be security experts (RQ-F1). Clipaha should work with other techniques (MFA, PAKE) and be compatible with a range of important use cases, e.g., changing the passwords or authentication of non-human devices (RQ-F2).

4.2 Password Hashing Algorithm

We propose an algorithm to perform password hashing on the client. With server relief, the client will provide the brunt of the work to hash the password. Hence, the client must be provided with all necessary parameters, including the salt. Traditionally, said salt would be stored in the password database on the server and looked up when the client provides the username. However, this would open the system to *enumeration*

Algorithm 2: Clipaha's server-side hashing process.

function CLIPAHASERVERHASH(ClipahaHash)
  return HASH(ClipahaHash)
end function

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Table 3: Proposed security levels. Iterations is the number of rounds the algorithm runs; Memory is the amount of memory used by the hash function.

Level	`low`	`medium`	`high`	`ultra`
Iterations	6	5	3	3
Memory (MiB)	192	384	1024	2016

attacks, as online attackers can easily detect the lack of salt for unregistered users or see how they change over time.

To circumvent *enumeration attacks* (RQ-S2), as well as avoiding *salt collisions* (RQ-S1), Clipaha does not involve the server in the hashing process and lets the client calculate the salt locally based on a deterministic *database identifier* (e.g., the server's domain) and the username. This choice deviates from the standard approach of using a random salt. We argue that only uniqueness, and not randomness, is required to avoid *salt collisions*. Therefore, credential bruteforcing will not gain a significant structural advantage by having a predictable, low-entropy salt compared to the case where it has access to a random, high-entropy one.

Figure 1 depicts the general idea behind Clipaha's design while Algorithms 1 and 2 describe respectively the client- and server-side hash more formally.

As can be seen on Algorithm 1, CLIPAHAHASH performs three steps. First, the function translates the username into its canonical representation using the CANONICALIZEUSERNAME function (e.g., lowering the case to attain case-insensitive usernames). This ensures the system will return the same hash for usernames that should be considered equivalent. Second, to further reduce the risk of *salt collisions*, the function DELIMIT must ensure the database identifier and the canonical username cannot be mixed when their strings are concatenated (RQ-S1). Clipaha does so by prepending the 32-bit Little Endian representation of the size of the strings. Finally, it computes and returns the hash of the password and the salt using a strong hash function PASSWORDHASH.

Continuing with Algorithm 2, the input to CLIPAHASERVERHASH would be the output of CLIPAHAHASH. This function only performs a call to a one-way function HASH, which does not need to be computationally expensive. This prevents *pass-the-hash attacks* (RQ-S3). Otherwise, an offline attacker (with read-only access to the database) could replay the entry in the database to access the service.

4.3 Client API and Security Levels

We implemented a JavaScript server relief library called clipaha.js that developers can include in their projects on the client side.[1] We used the Argon2id (Biryukov et al., 2017) hash algorithm to demonstrate that state-of-the-art algorithms can be moved from the server to the client side. Depending on the parameters' configuration, Argon2id can run on a wide range of devices. However, the choice of these parameters directly impacts the security level and is, therefore, sensitive.

Our library, clipaha.js, leverages the lessons learned from (Wijayarathna and Arachchilage, 2018) to provide a *developer-friendly* API (RQ-F1). We simplify the usage by hidding the parameters of the underlying hash function and instead provide four *security levels* (RQ-P1 and RQ-S4) which allow balancing between security and compatibility. The four levels represent parameter settings to target different classes of devices:

`low` is intended to maximize the support for ancient devices, like smartphones, ebook readers, or even computers from the early 2000s.

`medium` targets all computers from the mid-2000s and most smartphones and ebooks from the last 5 years.

`high` is the value that we would recommend for new developments where only computers from the last 5 years are considered.

`ultra` uses the maximum amount of RAM possible currently (a bit less than 2 GiB due to browser limits as explained on Subsection 5.1) and will run flawlessly on most modern laptops with at least 3 GiB of RAM.

The parameters used when hashing the password with Argon2id in the current implementation of clipaha.js are chosen to fit the memory of the device class and use as many/few iterations to keep the execution time below 20 seconds (RQ-P1). Since higher memory requirements involve more computation per iteration, the number of iterations needed in lower security levels is higher (see Table 3).

Introducing security levels represents a potential security threat. An attacker preferably targets systems using the `low` security level, which has lower requirements on computation and memory. Therefore, allowing a powerful offline attacker to execute credential bruteforcing at a higher rate. However, the execution time on the `low` security level still requires a non-trivial amount of computing time (e.g., 200 ms on a Core i7-7700k) compared to conventional server-side hash functions. Also, the legacy devices targeted with the `low` security level are expected to be phased out, increasing security requirements over time.

5 IMPLEMENTING THE *Clipaha* LIBRARY

We implemented and publicly released an implementation of Clipaha.[1] We briefly describe the server implementation in MicroPython, followed by a more detailed discussion of the challenges for the client implementation in Javascript/WebAssembly.

Server Implementation. We implemented a proof-of-concept of the server part of Clipaha on an ESP8266 with 80 KiB of user data SRAM using MicroPython 1.11.0 and created a web interface for user login.

Client Implementation. One critical objective of Clipaha is to keep authentication times reasonable (RQ-P1). Our implementation of the client side is based on the Emscripten framework (Zakai, 2011) to use WebAssembly (Haas et al., 2017), when available, and fall back to JavaScript otherwise. Clipaha also leverages workers, when available, to ensure the main thread can process events as the hashing proceeds. Performing password hashing in JavaScript presents various challenges which arise mainly because we aim for our library to be compatible with various devices.

Below, we discuss the four major challenges we found while implementing the library: memory, multithreading, performance, and data remanence.

5.1 Memory

Most JavaScript virtual machines limit the programmer's ability to make huge allocations, e.g., for ArrayBuffers, the size is limited to $2^{32}-1$ bytes (Pierron et al., 2020). However, a paging approach could help overcome the limitation. On WebAssembly, work is in progress to support 64-bit addressing (Smith et al., 2020).

In any case, manual testing in conjunction with our evaluation (Subsection 6.3) shows that those memory limits are more stringent in reality. On desktops, the limit is 2 GiB for the whole memory stack (reason behind our use of 2016 MiB of memory instead of 2048 MiB for `ultra` in Table 3) on both Firefox and Chrome. On mobile phones, the limits are significantly lower. For example, the mobile version of Chrome limits available memory to a bit over 1 GiB even if the device has more. On such devices, the memory requirements for `high`, 1 GiB, or `ultra`, almost 2 GiB, cannot be attained even when allocating memory early.

Instead of a statically allocated buffer, we also use `malloc` because the generated code will otherwise include very large memory files (Zakai, 2017), which are undesirable. Providing the statically allocated buffer would allow for disabling `malloc` and reduce the risk of an allocation failure caused by fragmentation. However, as we did not see fragmentation issues on any browsers in our evaluation, and the only `malloc` allocation done is the huge buffer used by Argon2, we have opted for not using a statically allocated buffer.

5.2 Multithreading

After discovering the SPECTRE vulnerability (Kocher et al., 2019), some browsers disabled the possibility of using shared memory buffers across web workers to avoid side-channel attacks. The situation has now improved, with some browsers allowing their usage under certain specific conditions (van Kesteren, 2020). Even in such cases, to allow threads to share memory, it is necessary to send specific headers on the web application which isolate the application and restrict its ability to add external resources. Not all browsers, e.g., Safari, support this currently. Nevertheless, to prepare for future threading support, we raised the number of parallelizable work units (called lanes by Argon2) to allow for up to 256 simultaneous threads.

5.3 Performance

JavaScript supports only 32-bit integer operations. This conflicts with the core of Argon2: Lyra2's BlaMka hash function (Andrade et al., 2016), which heavily uses 64-bit arithmetic. This, together with the fact that JavaScript is usually interpreted by the browser, makes the pure Javascript implementation of Clipaha slow (see Subsection 6.3).

To address this, Clipaha relies on WebAssembly, which increases the performance by providing 64-bit integer support and compilation into native code. Nevertheless, WebAssembly presents certain performance limitations over native code. As it becomes more widely available, SIMD support and other similar improvements will bring WebAssembly's performance more on par (Rossberg et al., 2018). For example, preliminary testing hints that support for 128-bit vector SIMD provides an improvement in run time of around 20%.

5.4 Data Remanence

JavaScript does not provide any guarantees regarding the lifetime of variables and handles strings as immutable objects. Consequently, the password to be hashed may remain unmodified on memory over a large period of time until it is eventually overwritten. Overwritting the memory ourselves is also unfeasible

Table 4: An overview of the evaluations of Clipaha.

Label	Description
Eval-1:	Security analysis
Eval-2:	Server-side: Proof-of-concept with a microcontroller
Eval-3:	Client-side: performance comparison of WebAssembly vs Javascript
Eval-4:	Client-side: baseline performance comparison in 35 settings
Eval-5:	Analysis of Clipaha in real scenarios

since the password is provided as an immutable object. Consequently, trying to reduce the lifetime in memory of intermediate results by overwritting them is futile as it will not necessarily reduce the amount of time during which password derived data will remain in memory but will impact performance significantly. Therefore, Clipaha does not clean-up the stack nor the allocated hash memory after a computation as doing so will not improve security.

6 EVALUATION

To comprehensively evaluate Clipaha, we have considered the work from four complementary perspectives with a total of five parts, as shown in Table 4. Eval-1 analyzes the security of Clipaha according to our threat model. Eval-2 focuses on the server, with a proof-of-concept of Clipaha running on very constrained hardware (as found on certain IoT devices) with appropriate measurements. Eval-3 provides a comparison between WebAssembly and Javascript. Eval-4 is a performance comparison between the state-of-the-art and Clipaha on a total of 35 devices/softwares grouped into three capability classes: *phone*, *tablet*, and *computer*. Finally, Eval-5, considers the deployment requirements of password systems and discuss how Clipaha addresses them in several scenarios.

The experiments on Eval-3 and Eval-4 were run five times on each setup, using different salt and password values on each run. We used the median to represent each setup's result.

6.1 Security Analysis

In the following, we discuss the security requirements presented in our threat model.

Regarding *Salt collisions* (RQ-S1), Clipaha's security stems from its salt components being unambiguously delimited (see Subsection 4.2). Due to the unique database identifier, Clipaha's salts are secure against collisions even across systems using the same salt calculation method. Confronted with systems using different ways of choosing their salt, it is unlikely that salts will collide since Clipaha prefixes the unique database identifier with its size as a 32-bit integer. For normal usage, i.e., with database identifiers under 65536 bytes, the size prefix will contain a sequence of two consecutive null bytes. Such a sequence is uncommon in text strings and would appear randomly with probability 2^{-16}. If we tackle the full string, the probability of a full match on a random string is 2^{32+8*n} where n is the length in bytes of the database identifier.

To avoid timing side-channels leading to *User enumeration attacks* (RQ-S2), an implementation should perform all extra server-side steps, even if the password is wrong. Similarly, an attacker may detect changes in password hashing parameters or salts provided by the server for a specific user. Consequently, an approach must ensure all provided data is either global and not depending on the username or constant for a specific username during the whole database life. Regarding the first attack solving it will always depend on the way the system is implemented. Clipaha minimizes the risk of this issue by choosing Argon2id which masks the side-channels by first doing a data-independent pass. Also, we take the issue into account when designing the scenarios and clearly explain how to implement them avoiding timing side-channels. As for the second attack, Clipaha only uses globally public parameters and user-provided inputs without any server-side processing as described in Subsection 4.2 and is, consequently, not affected by it.

In *Pass-the-hash attacks* (RQ-S3), the (offline) attacker tries to use the already hashed credential stored in the breached server database. Clipaha avoids this by adding an additional lightweight hash on the server side. As a result, the attacker would still need to revert this hash to be able to use the credentials stored in the database. Since the result of Argon2 should be indistinguishable from a random string, this means the attacker will need to perform on average 2^{n-1} guesses where n is the length in bits of the Argon2 computation. We provide a performance measure of such an extra hash in Eval-2.

Credential bruteforcing becomes significantly more difficult when using modern password hashing algorithms like Argon2 (RQ-S4). This is a consequence of increasing significantly the amount of resources, i.e., CPU time and memory amount and bandwidth, needed for each guess attempt. Argon2 explicitly targets optimizations and TMTOs (Hatzivasilis, 2017) to ensure that the guess time does not depend on the attacker's ability to use accelerators. With Clipaha, online attackers now have to perform the computation

themselves instead of moving it to the authentication server. Offline attackers still have to perform the computation themselves and, since all the hash parameters would be known beforehand, could try to precompute valid outputs (Bernstein and Lange, 2013).

An offline attacker trying to precompute password hashes will see a reduction of the actual, i.e., wall, time needed to perform the attack since the attacker could create beforehand a hash table mapping hash function outputs to inputs. Nevertheless, the CPU time, memory bandwidth and amount, and long term storage costs for an attacker would be raised significantly. First, the attacker would need to target a specific, i.e., linear, number of usernames as the unique salt would increase the cost of the attack with the number of targeted usernames. Similarly, even for the single username case, the attacker will need to calculate the hash for all the password guesses.

Finally, Clipaha provides a *reference library* (RQ-F1)[1] with an easy-to-use API and pre-defined security levels to minimize implementation mistakes by software developers and make Clipaha readily deployable.

6.2 Microcontroller Proof-of-Concept

To show the feasibility of deploying our approach, we created a proof-of-concept in MicroPython targeting the login and registration processes on an ESP8266 (Espressif Systems, 2020) with only 80 KiB of SRAM available. Thus, we could successfully show that the server side of Clipaha can run even on such constrained hardware.

Next, we benchmarked two versions of our code. The first uses an additional locally executed hash function, SHA256, to avoid *pass-the-hash* attacks (RQ-S3). The second performs a raw string comparison with a locally stored password.

We performed 1000 measurements of the time it took to hash and securely compare the encrypted password on the device along with the time it took to process a login request without the local hash. We measured that the hash measured for 3.7% of the time it took to perform the request with a median of 735.425 µs for the hash and 19 604.25 µs for the full requests. Requests without the hash had a median of 19 005.1 µs in comparison. As expected, with server relief, the code on the server side is fast even when run on a microcontroller (less than 20 ms).

6.3 WebAssembly vs JavaScript

Our client implementation of Clipaha uses WebAssembly and only falls back to JavaScript when necessary, as the latter is very slow. With Eval-3, we compare the native speeds on a desktop, a Core i7-4710HQ with 16 GiB of RAM, with WebAssembly and Javascript. As seen in Table 5, on a Desktop, WebAssembly incurs slowdowns of around 2-3.5x compared to native code, while JavaScript imposes a less consistent penalty of 30-130x. WebAssembly's penalty may be caused by optimization choices when compiling the middle code, the lack of native SIMD support, and other minor overheads. The JavaScript one is consistent with the expected impact of using an interpreted language.

6.4 Baseline Performance Comparison

Clipaha moves the complexity to the client-side. We have shown in Eval-2 that IoT devices can handle the remaining computation at the server side. In this section we focus on the complexity for the client-side devices, which we expect to be: mobile phones, tablets, and computers, with the aim to investigate Clipaha's portability and ability to run on them. This allows us to validate the security levels presented in Table 3.

We also use this opportunity to compare Clipaha with the similar libsodium (see Section 2). In particular we are comparing the benchmarks against a similar implementation using lisodium.js (Denis et al., 2020). We chose libsodium.js because it is the only library providing Argon2id support that we could find available for browsers.[2]

We run Clipaha and libsodium on a total of 21 devices, using different combinations of browsers and devices. This resulted in the 35 different benchmarks summarized in Table 6, where we roughly grouped the results based on the capability of the hardware. On the phone category, devices ranged from a Fairphone 2 with a Sanpdragon 801 CPU and 2 GiB of RAM to a Samsung S10+ with an Exynos 9820 CPU and 8 GiB of RAM. On the tablet category, devices ranged from an iPad with an A12 Bionic CPU and 3 GiB of RAM to a Surface Pro with a Core i5-7300U CPU and 8 GiB of RAM. Finally, on the computer category, device CPUs ranged from a Core i7-4710HQ CPU to a Core i5-9400T, including also an AMD Ryzen7 4700U and an ARM M1, while RAM amounts varied from 8 GiB to 24 GiB.

We see that phones are up to ten times slower than computers. Such a difference may be caused by slower phone memory, a slower processor, or worse code

[2] During benchmarking we found that Internet Explorer 11 and Edge are unable to run the libsodium.js code inside a dedicated web worker because the cryptographic `getRandomValues` API is not exposed. Since libsodium.js does not use this API when computing Argon2id hashes, we provided a trivial polyfill returning the input `TypedArray` to be able to proceed.

Table 5: Native speed versus WebAssembly and JavaScript on a Core i7-4710HQ with 16 GiB of RAM. “Factor” is normalized for the first row with native code. Chromium version is 85.0.4183.83 while Firefox version is 68.12.0esr.

Browser	Impl.	`low`		`medium`		`high`		`ultra`	
		Time (s)	Factor	Time (s)	Factor	Time (s)	Factor	Time (s)	Factor
Native	AVX2	0.653	1	1.446	1	2.083	1	4.068	1
Chromium	WebAssembly	1.873	2.87	3.181	2.20	5.269	2.53	10.845	2.67
	JavaScript	80.507	123.29	133.685	92.45	215.930	103.66	422.795	103.93
Firefox	WebAssembly	2.228	3.41	3.770	2.61	6.040	2.90	11.912	2.93
	JavaScript	29.109	44.58	47.909	33.13	77.254	37.09	151.300	37.19

optimization by phone-based browsers. Aiming at the best intercompatibility, we can appreciate that `low`, which improves on currently used defaults, did work on a vast majority of devices, providing delays of up to 20 s. Although this may look long (see RQ-P1), we find it acceptable as it allows even fairly obsolete clients to benefit from the security provided by Argon2. With caching the hashed credentials (see Eval-5), this could also be a one-time cost amortized over a larger number of logins. Given the relatively low failure rate, `medium` is well suited for most modern mobile devices with 3 GiB of RAM or more where such large allocations can be performed. Finally, `high` and `ultra` are restricted to browsers like Firefox which do not limit allocation size. These two security levels also require either a high-end mobile device with enough RAM or a desktop system.

We also observe that Clipaha is 25 to 50% faster than libsodium.js. This difference may be caused by additional overhead in libsodium.js, or by Clipaha's use of fixed Argon2 parameters allowing the compiler to produce faster code. Clipaha also completed the benchmarks on certain devices on which libsodium.js failed, probably because of the early allocation of memory made by Clipaha to avoid fragmentation.

6.5 Real Scenarios

Our final evaluation of Clipaha is concerned with its suitability for the many contexts where passwords are used (RQ-F2). While most systems will need a process for users to log in and, maybe, register and change their passwords, more complex requirements may also arise. As with any other authentication approach, implementing these requirements carelessly may result in new security problems. To address this risk and ensure that Clipaha is deployed in such systems, we provide implementation scenarios with an analysis of those security issues that Clipaha cannot solve.

As shown in Table 7, we consider: three sever-side authentication scenarios; three purely client-side scenarios; and two orthogonal techniques that can be used in conjunction with Clipaha: Password-Authenticated Key Exchange and Multi-Factor Authentication.

Registration. During the registration of a new user, the server sends the database identifier to the client. The client then sends the password in plain text and hashed.[3] The server validates all fields, including that the plaintext password matches the password policy.[4] If all went well, the server *inserts* the password hash along with the new user in the database (and proceeds with any further steps). If the verification fails, the server returns an error to the client. To avoid user enumeration, if the user is already present, we notify the user through an alternative communication channel without showing a different error message. Also, the timing of the insertion and any operations done afterwards must be the same whether the user is present or not.

Login. The client is provided with the database identifier and processes and sends the resulting hash and the username to the server. The server *verifies* the hash against the database. The user is allowed in if the username exists and the hash matches. To avoid user enumeration, the same message, e.g., *Invalid username or password*, must be displayed whether the user does not exist or the provided password is incorrect. Also, to avoid timing side-channel attacks, the verify operation should take the same amount of time if the user does not exist or the password is incorrect.

[3] We assume that communications between client and server are secure. One example of such a channel is a TLS connection. This avoids eavesdropping attacks.

[4] We assume that the client can be trusted to send the hashed password matching the sent plaintext password. If these were different, the client could bypass the password policy verification by sending instead a policy-breaking hashed password. Since this would only impact the user purposefully trying to break the policy and would be against the user's interest, the risk of this issue should be deemed low. Despite that, this could be avoided by either computing the hash on the server or, alternatively, by delegating this computation to a trusted and more powerful server. The server would then ensure the hashes match the ones sent. Given the resources needed to compute the hash, this may open the server for a DoS attack if clients repeatedly send requests.

Table 6: Benchmark results summary. Number of failures (F), minimum (Min) and maximum (Max) seconds taken to run are presented for each security level, device type and library. Only devices with at least a success are considered. Each security level was tested 5 times per device and browser.

Device type	Total	Library	low			medium			high			ultra		
			F	Max	Min	F	Max	Min	F	Max	Min	F	Max	Min
Phone	15	clipaha	0	19.134	7.362	1	31.571	12.228	13	25.900	20.012	13	54.175	41.648
		libsodium	0	33.313	12.611	2	54.887	21.076	13	48.320	34.648	13	100.377	69.114
Tablet	2	clipaha	0	2.279	2.015	0	4.048	3.739	0	7.689	5.940	1	17.674	17.674
		libsodium	0	4.287	3.440	0	7.253	5.739	1	11.913	11.913	1	25.857	25.857
Computer	18	clipaha	0	3.032	1.126	0	4.979	1.873	0	8.054	3.328	0	15.717	6.124
		libsodium	0	6.654	1.993	0	11.097	3.348	0	18.010	5.464	0	34.015	11.080

Table 7: An overview of the 8 evaluated scenarios.

Type	Description
Server	Registration, Login, Password change
Client	Client-side caching, Password managers, Non-human authentication
Support	Password-Authenticated Key Exchange, Multi-Factor Authentication

Password Change. To change passwords, the server sends the database identifier and the current session's username. The client then asks for the original password and the new password and hashes both. The client sends both passwords in plaintext[3] and their corresponding hashes to the server. The server proceeds to *verify* that the old password hash matches the one of the current user. If it does not, it returns an error message to the client. Otherwise, it checks that the new password plaintext complies with the password policy. If all goes through correctly, the server *updates* the database using the new password.

A Cross-Site Request Forgery (CSRF) (Watkins, 2001; KirstenS, 2020) attack allows an attacker to change the password to take over an account when the old password is not requested and verified. In such a case, appropriate measures must be taken, like using a CSRF token. Because this process depends on an authenticated session being already in place and will only affect the user, other security issues, e.g., user enumeration, are not a direct concern.

Client-Side Caching. To save resources (RQ-P1), client-side code using Clipaha may decide to cache the password hash computation. This may indeed be the case for obsolete devices as the login process otherwise can take almost 20 s (see Eval-4).

Since cached entries would be returned almost instantly, attackers with access to the shared cache could use a timing side-channel to see which entries it contains. Therefore, to avoid timing side-channels, caches should ensure the cached results are only provided once the same amount of time it initially took to calculate the hash has elapsed. Similarly, the keys and the contents of the cache could be used to impersonate users. Therefore, caches should be careful with data remanence and remove old entries periodically.

Password Managers. Password managers can interact with the client-side code using Clipaha and provide, along with the plaintext password, the already hashed password and the parameters used. The client can then use this hash to avoid doing the calculation. Similarly, the client-side code can send new hashes, along with the input parameters, to the manager for later retrieval.

To avoid pass-the-hash attacks, the manager must store the hashed passwords using the same security measures used for the plaintext. Similarly, the manager should also use the same access policies it would use for the plaintext password.

Non-Human Authentication Services using Clipaha may need to be available to embedded devices that try to authenticate themselves. Assuming accounts are not shared, such devices could replace the hashed password with a hash-length, i.e., 256-bit, token sampled from a good random source and registered during provisioning. The token would replace the Clipaha hashed password during *verification*. From a security perspective, this approach will work well if the token is stored securely and protected from physical access attacks and access by other domains or applications running on the device.

Password-Authenticated Key Exchange Integration. Clipaha is compatible with Password-Authenticated Key Exchange (PAKE), where a cryptographic key is negotiated using the password. Several steps must be done in constant time to avoid timing side-channels, even if the user does not exist. If the PAKE is well designed, it will also not be possible for the attacker to use server verifiers to impersonate the client.

Multi-Factor Authentication Integration. If Multi-Factor Authentication (MFA) is in place, the client

should provide its codes before *verification* along with the hashed password. These codes would be validated during *verification*. If extra actions from the server are required for MFA, they have to be performed after *verification* succeeds but before any further actions are carried out. Additional server actions and MFA value validation should be in constant time and look the same even when the user has no MFA or does not exist in order to prevent side-channels attacks.

7 CONCLUSIONS

This paper introduces Clipaha, an scheme for server relief. Clipaha allows using modern password hashing functions with high security parameters even on resource constrained IoT devices by moving the computation away from them and into the client.

We test Cliapaha's performance, security, and readiness to be deployed. We specify and analyze the security of eight authentication-related scenarios which leverage this scheme and publicly release an implementation[1].

From the security analysis, we conclude that Clipaha is a key solution to help build more secure authentication systems since it is resistant to salt collisions and user enumeration attacks as opposed to prior work.

The deployability tests show that Clipaha is ready to be used for web-based authentication. Clipaha's server-side is lightweight enough to work even on an ESP8266 with only 80 KiB of RAM. The benchmarks show that client-side, Clipaha performs over two times faster than the closest baseline: libsodium.js. Also, thanks to Clipaha's four security levels and the benchmarks we have performed, developers can balance between security and running Clipaha on most devices from the last five years while addressing any compatibility issues caused by their technical limitations.

In conclusion, Clipaha has the potential to impact embedded systems like SoHo network appliances and IoT gateways which are resource constrained and rarely require flows more complex than registration (during provisioning) and authentication.

ACKNOWLEDGEMENTS

This paper and most of the artifacts associated with it have been developed as part of the the Resilient IoT project and under a grant from The Swedish Civil Contingencies Agency (MSB).

The first author would like to acknowledge the feedback received from Vicent Nos and Ignacio Bedoya during his tenure as CISO for Lescovex when the ideas behind this paper started taking shape. The first author also would like to acknowledge the cofounders of Garmer Technologies OÜ for believing that his research could have commercial use.

Finally, the authors would like to acknowledge the benchmark data contributions from some members of the Networks and Systems unit at Chalmers and some individual members. The authors would like to specially thank the contributions from Carlo Brunetta (benchmarks on most Apple devices) and Elaine Monteagudo Sánchez (benchmarks on most browsers using her Xiaomi Redmi Note 8T and her Core i7-8750H laptop).

REFERENCES

Alwen, J. and Blocki, J. (2017). Towards practical attacks on argon2i and balloon hashing. In *EuroS&P*.

Andrade, E. R., Simplicio, M. A., Barreto, P. S., and dos Santos, P. C. (2016). Lyra2: Efficient password hashing with high security against Time-Memory Trade-Offs. *IEEE Transactions on Computers*, 65(10).

Aumasson, J.-P. et al. (2013). Password hashing competition (phc).

Bai, W. and Blocki, J. (2021). Dahash: Distribution aware tuning of password hashing costs. In *FC*.

Bauman, E., Lu, Y., and Lin, Z. (2015). Half a century of practice: Who is still storing plaintext passwords? In *ISPEC*.

Bernstein, D. J. and Lange, T. (2013). Non-uniform cracks in the concrete: The power of free precomputation. In *ASIACRYPT*.

Biryukov, A., Dinu, D., and Khovratovich, D. (2016). Argon2: New generation of memory-hard functions for password hashing and other applications. In *EuroS&P*.

Biryukov, A., Dinu, D., and Khovratovich, D. (2017). Argon2: the memory-hard function for password hashing and other applications. Technical report, Password Hashing Competition.

Blanchard, E., Coquand, X., and Selker, T. (2019). Moving to client-side hashing for online authentication. In *STAST*.

Corbató, F. J. (1963). *The Compatible Time-Sharing System: A Programmer's Guide*. The MIT Press.

Denis, F. (2019). Libsodium documentation: Password hashing.

Denis, F. et al. (2020). libsodium.js: README.

Eliasen, M. (2019). Developers, its 2019, hash passwords accordingly.

Espressif Systems (2020). ESP8266EX datasheet. Datasheet, Espressif Systems.

Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., and Bastien, J. (2017). Bringing the web up to speed with WebAssembly. In *PLDI*.

Hatzivasilis, G. (2017). Password-hashing status. *Cryptography*, 1(2).

KirstenS (2020). Cross site request forgery (csrf).

Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Werner, H., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. (2019). Spectre attacks: Exploiting speculative execution. In *S&P*.

Lee, J. A. N., Rosin, R., Corbató, F. J., Fano, R. M., Greenberger, M., Licklider, J. C., Ross, D. T., and Scherr, A. L. (1992). The Project MAC interviews. *IEEE Annals of the History of Computing*, 14(2).

Morris, R. and Thompson, K. (1979). Password security: A case history. *Communications of the ACM*, 22(11).

Ntantogian, C., Malliaros, S., and Xenakis, C. (2019). Evaluation of password hashing schemes in open source web platforms. *Computers & Security*, 84.

Oechslin, P. (2003). Making a faster cryptanalytic time-memory trade-off. In Boneh, D., editor, *CRYPTO*.

Pierron, N. B., Scholz, F., and Kettner, P. (2020). Rangeerror: invalid array length.

Rossberg, A. et al. (2018). Simd proposal for webassembly.

Saad, E. and Mitchell, R. (2020). Testing for account enumeration and guessable user account.

Security, D. (2019). Salted password hashing - doing it right.

Skowron, P. et al. (2021). Brute force attack.

Smith, B. et al. (2020). Memory64 proposal for webassembly.

Van Acker, S., Hausknecht, D., and Sabelfeld, A. (2017). Measuring login webpage security. In *SAC*.

van Kesteren, A. (2020). Safely reviving shared memory.

Watkins, P. (2001). Cross-site request forgeries (re: The dangers of allowing users to post images).

Wijayarathna, C. and Arachchilage, N. A. G. (2018). Why johnny can't store passwords securely? a usability evaluation of bouncycastle password hashing. In *EASE*.

Wilkes, M. V. (1968). *Time-sharing computer systems.* Number 5 in Computer monographs. American Elsevier, New York, NY, USA.

Zakai, A. (2011). Emscripten: an LLVM-to-JavaScript compiler. In *OOPSLA*.

Zakai, A. (2017). What can cause the size of the .js.mem file increase?

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS