Hashing It Out: An Analysis of Cryptographic Hash Functions for Password Storage

Kitty Wang
kittywang@college.harvard.edu

Jonathan Wu
jonathanwu@college.harvard.edu

Eric Gong
ericgong@college.harvard.edu

Abstract

Cryptographic hash functions play a critical role in secured password authentication. To be both secure and practical, these functions should exhibit a variety of mathematical guarantees, meet certain performance requirements, and be easily deployable to software stacks. In this paper, we present a holistic evaluation framework that considers all mentioned criteria, examining some of the most popular cryptographic hash functions to determine if there exists, among these popular hash functions, one option with clear advantages over all alternatives. We determine that, when properly configured, scrypt and Argon2 are generally good contenders and PBKDF2 is acceptable in memory-constrained environments.

1 Introduction

Cryptographic hash functions (CHFs) – a class of algorithms which map arbitrary length input to fixed size outputs, commonly referred to as a message digest or hash – have a myriad of applications, be it hash-based message authentication codes (HMACs), key derivation functions (KDFs), or – of greatest interest to this study – secure password authentication [13, 20]. In particular, over the past few decades, dozens of CHFs have been proposed and evaluated for the purpose of secured password authentication, albeit to varying degrees of success [16].

1.1 Cryptographic Hash Function Properties

CHFs that are conducive to secured password authentication exhibit three main properties: Collision Resistance, Pre-image Resistance, and Second Pre-image Resistance [1]. Collision Resistance is the property that given two random, distinct inputs, the probability of the resulting hash of these two inputs being the same should be low. Pre-image Resistance is the property that given a target hash value, it should be computationally intractable to determine an input that would yield the target hash value, whether through leveraging properties of the CHF in question, or through brute-force methods. Second Pre-image Resistance is the property that given both an input and the hash of the input, it should be computationally intractable to determine a distinct second input which yields the same output hash.

In practice, however, a variety of factors diminish the efficacy of otherwise theoretically secure CHFs. For instance, the tendency of humans to re-use passwords, or for different humans to choose identical passwords, results in a significant reduction of the search space for an attacker, making dictionary attacks or Rainbow Tables a far more viable option, given that the cost of computing a large number of hashes can be amortized over a countless number of users [7]. In response, techniques such as Peppering were proposed, where a secret string is used in conjunction with each user's password when hashing, so as to ensure that pre-computation of CHF hashes in brute force attacks could not be amortized across differing databases. However, given that the same secret pepper string is used across all users, this method still lends itself to frequency analysis, particularly in the case of passwords. Currently, Salting has become standard practice for secured password authentication: a string, not necessarily kept secret, is generated per user, and used in conjunction with the user's password when hashing [13]. This ensures that the computational cost of brute force attacks cannot be amortized across different users, even within the same database, nor are techniques such as frequency analysis viable, given that Salting ensures distinct users with the same password will have different password hashes.

In addition, to further inhibit the ability of attackers to conduct offline attacks, some CHFs have been designed to be memory-hard, such that the calculation of a single hash necessitates the continuous usage of a certain amount of memory for the duration of the hash calculation process. Given that RAM is a scarce resource, even on customized hardware, memory-hard CHFs diminish the ability of an attacker to utilize even ASICs to greatly accelerate the process of brute-force attacks [23]. However, it should be noted that some CHFs only partially exhibit memory-hardness, in that it is possible to reduce memory utilization by incurring a proportional increase in time expenditure [33]. Attacks which utilize these properties of a CHF are known as Time-Memory Tradeoff attacks; the ability to reduce the utilization of scare RAM resources at the cost of computation time severely hinder these CHFs' resistance to specialized hardware, which is optimized for reducing computation time.

1.2 Popular Cryptographic Hash Functions

Currently, popular CHFs that are considered sufficiently cryptographically secure including scrypt, yescrypt, Argon2 and PBKDF2, with Argon2 being recognized as the winner of the NIST's Password Hashing Competition, and yescrypt being recognized as Finalist [16, 33]. On the other hand, some CHFs, while widely used in the past, are known to be insufficiently secure, and are being slowly phased out, with one such example being bcrypt, a CHF that can be relatively easily attacked via Field Programmable Gate Arrays [11, 33].

While there may exist numerous popular CHFs, there exists little prior work on comprehensive evaluations of these CHFs. A majority of work relating to CHFs only examine a couple CHFs in a single study, resulting in a lack of a comprehensive evaluation due to difficulties of standardize results across differing studies with distinct methodologies [11, 23, 30]. Furthermore, many studies tend to examine the performance of CHFs under specific conditions – such as resource constrained applications – and typically only examine a couple of CHFs, making it difficult to standardize results across different studies for accurate comparisons [4, 23, 30]. However, the question of whether or not there exists a superior

CHF scheme among numerous schemes that are deemed secure and are widely used, evaluated holistically from various aspects, including theoretical security, resistance to attacks in practice, and ease of deployment in existing or new software stacks, is a question of interest to software engineers and system administrators alike. Indeed, even if a variety of CHFs are deemed viable or sufficiently secure for use, it is imperative to identify if there is any one CHF with clear advantages when evaluated holistically on a variety of fronts.

2 Methodology

As such, in this paper, we propose a centralized set of evaluation heuristics that evaluate CHFs on three different fronts: theoretical guarantees, performance on benchmarks, and ease of deployment in software. In particular, we will assess salted variations of PBKDF2, scrypt, yescrypt, and Argon2, comparing them against two baselines: plaintext passwords and SHA-256 without salting. These algorithms and baselines were chosen for their prevalency in large projects, packages, and libraries, and leaked databases.

2.1 Theoretical Guarantees

We evaluate the theoretical guarantees of the CHFs on two metrics. The first is each CHF's capacity for confusion and diffusion, and the second is each CHF's resistance to ASIC attacks, or similar specialized hardware attacks, through the memory-hardness of the CHF.

We first evaluate hashing functions by their capacity for confusion and diffusion. An emphasis on ensuring diffusion provides a more uniform distribution in the output ciphertext hash, such that if a single character in the plaintext password is changed, the resulting hash should observe multiple characters changing. An emphasis on confusion obfuscates the relationship between the password, any salts or peppers used, and the output hash, which ensures that an adversary cannot readily reverse-engineer or predict the output of the hashing function given some controlled input. Targeting these two criteria, we develop a method of measuring the correlation between output hashes and their corresponding plaintext passwords by developing metrics that quantify these properties. This procedure provides more empirical insight into the number of bits of entropy made available through each function. As a further investigation into the aspects of confusion and diffusion for each of the hashing functions, we examine the functions' application of noninvertible operations and invocations of operations such as S-Box and P-Box steps. At the conclusion of this theoretical dissection, we will evaluate the functions' resistance against lookup table attacks, and consider potential tradeoffs of similar functions with varying numbers of iterations or rounds of computation that they perform.

So as to gauge the ability of CHFs in resisting specialized hardware attacks, such as those conducted with ASICs, we aim to determine whether or not specialized hardware can result in non-trivial reductions in computation time, allowing for more efficient offline attacks. Specifically, given that memory availability can be a bottleneck for specialized hardware, and thus, leveraged as a viable defense against hardware attacks, we aim to examine the extent to which the selected CHFs exhibit memory hardness. In particular, we define three categories for classifying the memory hardness guarantees of CHFs: None, Mediocre, and Acceptable. CHFs that shall be rated as "None" will be CHFs which require very little memory to compute, and have no parameter that enables scaling memory utilization, thus making these CHFs highly parallelizable, and vulnerable to specialized hardware attacks. CHFs that shall be rated "Mediocre" will be those that have tunable memory hardness but are nonetheless susceptible Time-Memory Tradeoff attacks, which significantly reduces their resistance to specialized hardware attacks. CHFs that shall be rated "Acceptable" shall be those that have tunable memory hardness, such that the CHF is either always maintains memory hardness guarantees or can maintain memory hardness guarantees by increasing the CHF parameter values.

2.2 Benchmarking

We also evaluate the real-world security of CHF implementations. The security of a CHF is not just how strong it is theoretically, but also how it is being used in the wild. To benchmark this, we compiled a list of common and recommended configurations for each CHF, as well as how they could be changed to make them more time or memory intensive to compute. Computing hashes should not be overly burdensome on legitimate verifiers, and should be hard for adversaries to compute faster. Starting with evaluating the computation time of each CHF's default or recommended parameters, we evaluate the functions' resistance against naive brute force hashing attacks by iterating through the passwords of the top 32 most common passwords in the well-known password file rockyou.txt and computing the hash of each of them to identify the time necessary to both verify a legitimate hash and brute force a password. This is important to identify as many projects simply use the defaults and while most CHFs can be made harder based on tuning various parameters, the data that ends up leaked in data leaks tend to use default parameters. The second stage of benchmarks builds on that by pushing these CHFs further, to determine how their resistance scales with increase in parameters. This helps determine how the parameters should be adjusted based on general computing hardware power over time, or any weaknesses in the CHF itself. Lastly, we proceed with a consideration for memory usage, tracking the memory usage required to perform a hash (stack, heap, and off-heap allocation space), as well as the memory usage of the library that the implementation uses, to inform choices for either ASIC resistance, or memory-contained environments like in IoT devices.

2.3 Ease of Deployment

In evaluating the ease of deployment, we aim to determine the accessibility of the CHF for integration into existing software stacks, such that the CHF could be inserted into existing software solutions or intentionally selected for future software development projects. To measure ease of deployment, we investigate two main metrics: the degree to which each CHF is available for a variety of popular programming languages – which we shall refer to as Accessibility – and a detailed implementation evaluation – which we shall refer to as Ease-of-Implementation – for Python and C++, which are among the top 5 most popular programming languages in 2024 [8].

CHF Availability in various Programming Languages. We determine availability of a CHF for a given programming language by placing the CHF into one of three categories: "Widely Available," "Limited Third Party," or "Custom Implementation." CHFs that are classified as "Widely Available" for a particular programming language, shall be those such that the CHF is implemented and ready to use by satisfying at least one of three options. The first option is that the CHF is integrated in default or stable releases of a programming language through built-in functions. The second option is that the CHF is accessible through popular and reputable cryptographic libraries written for the programming language in question. The third option is that the CHF is accessible through well-developed wrappers which allow the porting of cryptographic libraries developed for a different programming language into the programming language in question. CHFs that are classified as "Limited Third Party" shall be those that do not satisfy any of the options to be deemed "Widely Available," but for which there nonetheless exist publicly accessible code that implements the CHF for the programming language in question, and could thus be used, provided the developers vet and verify the validity of the third-party implementation. CHFs that are classified as "Custom Implementation" shall be those for which few or no implementations exist, thus requiring developers to implement the CHF from scratch in the programming language in question.

We select popular programming languages based on popularity as assessed by the IEEE Spectrum Report on Popular Programming Languages in 2024 [8]. In particular, after selecting popular programming languages as defined by each language's weighted Spectrum Popularity Score, we remove programming languages for which secured password authentication is not a likely use-case of the language, like R, SAS, and Mathematics, as well as languages where secured password authentication is not relevant at all, like HTML and SQL. Overall, we examine 13 programming language that cover a variety of use cases, from being general purpose languages to those specifically for mobile app development. In particular, we classify each CHF as one of the three previously defined categories for each of the 13 programming languages. We then compute a normalized popularity score for each programming language by dividing the IEEE Spectrum Popularity Score for each language by the sum of Spectrum Popularity Scores across the 13 languages. Finally, for each CHF, we take the sum of the Normalized Popularity Scores of all programming languages for which the CHF is categorized as "Widely Available." The resulting Accessibility score thus estimates the proportion of software for which the given CHF can be integrated into, taking into account the differing degrees of prevalence for differing types of software stacks and use cases. In particular, an Accessibility score of 1 indicates that there exist either built-in methods or otherwise popular and accessible libraries that allow for the direct integration of the given CHF into a majority of existing use cases.

CHF Implementation Evaluation for C++ and Python. We create a detailed implementation evaluation of CHFs in Python and C++ based on the observed ease of deployment when implementing the benchmark tests. In particular, for each programming language and each CHF, we assign a Likert scale rating to describe the ease of deployment. Cumulatively, we aggregate the Likert scale rating for each CHF, such that we obtain an Ease-of-Implementation score out of 10 for each CHF that is examined. We design the Likert scale to range from 1 to 5, with 5 being a score assigned to programming-language-CHF pairs that are most easily deployed. In particular, CHFs that are rated 5 shall be those such that the CHF is built-in directly to the language or otherwise require only a few lines of code to implement. CHFs that are rated 4 shall be those such that straight-forward and routine processes – such as the installation of a package – are necessary, but no additional research or significant lines of code are required beyond that. CHFs that are rated 3 shall be those such that the CHF must be built from source, or thoroughly researched before integration due to a lack of documentation, or otherwise require significant time commitment for deployment. CHFs that are rated 2 shall be those for which little to no documentation exists, and for which even sources including developer forums are of little to no use; the deployment process rests solely upon the programmer, who must read source code to implement the CHF. The rating of 1 shall be specially reserved for those CHFs for which not only do they fail to meet the criteria for any higher rating, the CHF is also difficult to implement in a bug-free manner without significant time commitment, and requires knowledge of niche hash formats. As such, a higher Ease-of-Implementation score corresponds to CHFs that are easier to implement, with a score of 10 indicating that the CHF can be implemented with little to no difficulty for both C++ and Python.

3 Mathematical Analysis of CHF Security

To evaluate the security efficacy and performance of the selected CHFs from a theoretical perspective, we begin by considering the mathematical mechanisms potentially present to provide resistance against rainbow table exploits. Following this initial discussion and function breakdown, we aim to quantify the degree of confusion and diffusion through normalized indices developed using the Damerau-Levenshtein distance scheme [10] and the Strict Avalanche Criterion (SAC) [32]. In Claude Shannon's 1949 "Communication Theory of Secrecy Systems" [29], he identifies confusion and diffusion as the necessary properties to ensure the security of a cryptographic system, where confusion is defined as the obfuscation of the mapping from the plaintext to the hash, and diffusion aims to spread the statistical structure of the plaintext throughout the hash sample space. In practice, confusion provides a guarantee that each bit of the outputted hash should depend on multiple bits of the plaintext password, whereas diffusion strives to maximize the number of bits modified in the hash per bit of plaintext modification.

3.1 Table Lookup and Preimage Resistance

As the first component of the mathematical analysis of the security of the selected CHFs, we examine the presence of a numerical mechanism to provide the most basic of resistance against a lookup table (which stores the corresponding password in plaintext) and a rainbow table attack.

Plaintext (baseline). As a control, the baseline hashing function of outputting the plaintext as a hash fails to protect against a rainbow table attack. Since there are no mechanisms or manipulation of the plaintext password data before the function generates the corresponding hash value, adversaries may trivially invert $H_1(x)$, the "hashing" scheme, to then perform a search on the table entries before identifying the plaintext password and applying the same strategy to obtain arbitrary passwords. This baseline hashing scheme lacks mechanisms of confusion and diffusion.

SHA-256 without salting. Proceeding to the case of a general-purpose hashing function, we choose to examine the SHA-256 hashing scheme [28] due to its extensive applications in existing software systems. The SHA-256 hashing function is a keyless cryptographic hash function with digest length of 256 bits. Under the SHA-256 function, which we denote as $H_{\text{SHA256}}(x)$ from this point onward, the plaintext password is processed in blocks of $16 \cdot 32 = 512$ bits, with each block undergoing 64 rounds of computation. After padding the password to the nearest multiple of the block size, the algorithm then uses a combination of the following functions:

$$Ch(X, Y, Z) = (X \wedge Y) \oplus (\overline{X} \wedge Z)$$
$$Maj(X, Y, Z) = (X \wedge Y) \oplus (X \wedge Z) \oplus (Y \wedge Z),$$

where $Ch(X, Y, Z)$ defines a choice function in X, Y, Z and $Maj(X, Y, Z)$ defines a a majority function in the same inputs. When invoked in combination, these two functions are critical in enforcing a mechanism of diffusion. To demonstrate this on a smaller scale, we use a truth table conditioning on the assignments of each of the three parameters, which yields a near-1 probability of uniformly generating an output of either 0 or 1, thereby obfuscating the distribution of bits in the plaintext as follows:

X	Y	Z	$Ch(X, Y, Z)$	$Maj(X, Y, Z)$
0	0	0	0	0
0	0	1	0/1	0/1
0	1	0	0/1	0/1
0	1	1	1	0/1
1	0	0	0/1	0/1
1	0	1	0/1	0/1
1	1	0	0/1	0/1
1	1	1	0/1	1

In addition to these two functions, the padding process further contributes to the CHF's diffusion as it appends a 1 followed by any necessary number of 0s to reach a multiple of the block size. The step transformation of the CHF also employs four additional critical functions, which comprise the S-Box (Substitution-Box) computation necessary to provide confusion in the overall function as they are defined using right bit shifts (SHR(x)) and right rotations ROT R(x) [28]:

$$\Sigma_0(x) = \text{ROT R}^2(x) \oplus \text{ROT R}^{13}(x) \oplus \text{ROT R}^{22}(x)$$
$$\Sigma_1(x) = \text{ROT R}^6(x) \oplus \text{ROT R}^{11}(x) \oplus \text{ROT R}^{25}(x)$$
$$\sigma_0(x) = \text{ROT R}^7(x) \oplus \text{ROT R}^{18}(x) \oplus \text{SHR}^3(x)$$
$$\sigma_1(x) = \text{ROT R}^{17}(x) \oplus \text{ROT R}^{19}(x) \oplus \text{SHR}^{10}(x)$$

Through this complex chaining process, we observe that all of the bits in the first 16 words have some influence on the bits in the later 48 words, thereby giving us an enlarged internal intermediate message to work with. Each word of this expanded message is iterated over with a function that has a 256-bit state, where the initial state is a constant. During each iteration, the new state is a function of the previous round's state and a word from the expanded internal message. At the end of the 64 rounds, the internal state is returned as the outputted hash, which is now computationally infeasible to reverse by brute force due to the chaining of rotations and shifts within the internal message.

However, because there is no salt involved in this keyless general-purpose hashing scheme, an adversary can simply generate the hashes corresponding to a set of known probable plaintexts in a rainbow table. From there, one may identify the password if any of the SHA-256 hashes match the hash of the password of interest. Thus, while the SHA-256 cryptographic hashing function provides confusion and diffusion in its outputs, it fails to be resistant to rainbow table attacks.

PBKDF2 (600,000 iterations). PBKDF2 is typically used as a key-stretching algorithm, such that when given four input arguments, it maps a plaintext password to a hash that may be used as an encryption key in a cipher [17]. In terms of its inputs, the CHF accepts the following configurable arguments:

$$P = \text{plaintext password}$$
$$S = \text{salt}$$
$$C = \text{iteration count}$$
$$dkLen = \text{desired length of the output hash } H(P)$$

The security of the PBKDF2 hashing function stems from the foundational application of a pseudorandom function (PRF) for C rounds [21]. For the purpose of this paper, we focus on the 2023 OWASP recommendation of 600,000-iteration PBKDF2 function, which utilizes the SHA-256 function as its PRF. From the input arguments, only P needs to be kept secret by an authentic user. Letting $hLen$ represent the output length of the PRF, in this case 256 bits for SHA-256, ℓ be the number of $hLen$ sized blocks required to derive the output hash, and r be the number of bits required from the last block, we calculate l, r respectively as

$$\ell = \left\lceil \frac{dkLen}{hLen} \right\rceil$$
$$r = dkLen - (\ell - 1) \cdot hLen$$

From here, the function derives each block of the hash, T_i by computing the function F with the inputs P, S, C, i, where i denotes the index of the target block undergoing computation, where F can be defined as an exclusive-or sum of C iterations of the PRF:

$$F(P, S, C, i) = (U_1 \oplus U_2 \oplus \ldots \oplus U_C),$$
$$\text{where } U_1 = PRF(P, S||\text{int}(i))$$
$$U_2 = PRF(P, U_1)$$
$$\vdots$$
$$U_C = PRF(P, U_{C-1})$$

This gives a sequence of blocks comprising the final hash as follows,

$$T_1 = F(P, S, C, 1)$$
$$T_2 = F(P, S, C, 2)$$
$$\vdots$$
$$T_C = F(P, S, C, \ell)$$

thereby yielding an outputted hash that is the concatenation of the ℓ blocks:

$$H(P) = T_1 || \, T_2 \, || \, \ldots \, || \, T'_\ell,$$

where from the last block T_ℓ, first r bits are selected, depending on the length of derived key, *dkLen*. The underlying primitive of PRF (SHA-256 via HMAC) provides an existing robust mechanism of confusion and diffusion, as demonstrated in the previous subsection. The objective of the chaining process, combined with the XOR present at each iteration, is to increase computational expense and further provide confusion.

Since PBKDF2 accepts as an argument a generated salt, if plaintext passwords are hashed using the function such that every password has a mapping to a unique salt, a typical approach implemented in password storage, the function provides significant resistance against basic rainbow table attacks. Even though the salt may be stored in plaintext along with the hash of the salted password, because of the addition of the randomized salt, this additional prehashing process renders hashed passwords incomparable to even an identical password in the same or other database, and invalidating large pre-generated lists of common password that adversaries may use in an attempt to perform a lookup attack.

PBKDF2 (1,000,000 iterations). Similar to the PBKDF2 hash function with 600,000 iterations, the PBKDF2 with 1,000,000 iterations satisfies the properties of confusion and diffusion in the definition of an effective hash function thanks to the underlying foundation of SHA-256 used as the PRF every iteration. While increasing the iteration count in PBKDF2 does not directly improve its intrinsic "confusion" property in the classical cryptographic sense, increasing the number of iterations primarily aims to amplify the computational cost of the derived key computation, thereby slowing down brute-force and rainbow table attacks as an adversarial party is forced to compute the function for additional iterations as a part of their brute-force efforts.

By the same reasoning as PBKDF2 with 600,000 iterations as PBKDF2 under 1,000,000 iterations also employs a salting process, we determine that this version of PBKDF2 also provides resistance against rainbow table exploits.

scrypt. As a memory-hard CHF, scrypt is built on top of PBKDF (with HMAC-SHA-256) and a specialized mixing function (ROMix), which employs a BlockMix stage [25]. Breaking down the computational steps of the function, scrypt accepts the following input arguments

$$\begin{aligned} P &= \text{plaintext password} \\ S &= \text{salt} \\ C &= \text{iteration count} \\ N &= \text{CPU/memory cost parameter} \\ p &= \text{parallelization parameter} \\ dkLen &= \text{intended output length in octets of the output hash} \end{aligned}$$

The function begins with PBKDF2 to first derive an initial pseudorandom key from the password and salt, before applying the ROMix function to generate a large pseudorandom array of data dependent on that key. From there, the function then passes the output of the ROMix procedure through PBKDF once again to obtain the final hash.

Within the BlockMix stage of the ROMix computation, the Salsa20/8 function is invoked [25]. Assume that we have some input array of $2r$ blocks at this phase represented as $B_0, B_1, \ldots, B_{2r-1}$, where each block is 128 bits long. Setting $X = B_{2r-1}$, the BlockMix procedure iterates from $i = 0$ to $i = 2r - 1$, and computes $X = X \oplus B_i$ before applying the Salsa 20/8 function $X = \text{Salsa20/8}(X)$ and appending X to the output array. Following the termination of this loop, the outputted blocks are compiled such that the even-indexed outputs comprise the first half of the new array and the odd-indexed outputs form the latter half.

Building on top of the confusion and diffusion provided by PBKDF2, the invocation of the Salsa20/8 function further strengthens the security guarantees of scrypt overall. As a cipher based on Add-Rotate-XOR (ARX) operations, Salsa20/8 retains strong diffusion properties as each 512-bit input block is transformed by 8 rounds of the Salsa20 quarter-round function. Each quarter-round then nonlinearly mixes bits by additions, rotations, and XOR operations across multiple 32-bit words. Due to the nonlinearity of Salsa20's internal transformation, which also incorporates rotation and a modulus operation by 2^{32}, a change in any bit of the input will result in flips in multiple output bits due to the chained computation process. This ensures that scrypt satisfies the confusion requirements of an effective CHF.

Because each round of the Salsa20 function further spreads the influence of every input bit throughout the state, after a few rounds, the function has achieved the diffusion ideal, such that each output hashed bit is dependent on a combination of all of the input bits, thereby providing a completeness guarantee.

As scrypt accepts a salt as an argument at a per-password granularity, the function provides significant resistance against basic rainbow table attacks by the same rationale as observed with PBKDF2 as users' passwords can each be hashed and stored with a unique, randomized salt. Extending this basic resistance against rainbow tables, because the ROMix step requires a large array V in memory to store intermediate states to perform iterative mixing that must be executed in a sequential manner (it is non-parallelizable due to dependencies between each iteration), an adversary attempting a brute-force rainbow table approach simply must perform all of the memory-intensive computational steps. Consequently, scrypt introduces additional resistance to table lookup exploits through its memory hardness.

bcrypt. Unlike many of the other listed CHFs in this paper, bcrypt relies on a different underlying core function, namely the Blowfish block cipher [15], to perform its hashing operation. The bcrypt hashing function [26] accepts as arguments the following:

$$\begin{aligned} P &= \text{plaintext password} \\ S &= \text{salt} \\ C &= \text{iteration count for total of } 2^C \text{ rounds} \end{aligned}$$

Using the plaintext password as the key to the Blowfish encryption scheme, bcrypt operates in two phases, the first starting with a set of fixed constants for the Permutation Boxes (P-Boxes) and Substitution Boxes (S-Boxes) used in the cipher rounds, which are typically referred to as the initial Blowfish constants. The Blowfish key schedule initializes and modifies internal state consisting of this set of P-boxes (subkeys) and four S-boxes, each containing multiple

32-bit words. From there, the function proceeds to the second phase, which is where bcrypt spends most of the computational time. In the key schedule computation procedure, bcrypt iteratively mixes the password and salt to update the P-boxes and S-boxes to derive a final version, which is then used in the process of encrypting a fixed block of data (the 192-bit value "OrpheanBeholderScryDoubt") 64 times using $eksblowfish$ in ECB mode. The final output hash is compiled by concatenating the iteration count, sometimes also referred to as computational cost, 128-bit salt, and the results of the iterations of encryption.

The confusion and diffusion guarantees provided by bcrypt stems from the Blowfish key schedule, which leverages an iterative process that takes the original P-boxes and S-boxes (which are fixed constants derived from π) and transforms them using the given key. Each step involves:

(1) XORing the key material into the P-boxes.
(2) Encrypting a fixed all-zero block (or a block derived from previous steps) using the current P-boxes and S-boxes.
(3) Replacing P-box entries with parts of the resulting ciphertext.
(4) Using updated P-boxes for subsequent stages and eventually similarly updating S-boxes.

The nonlinearity in Blowfish's round function supplies confusion, as each round can be expressed as

$$\text{Left} \oplus = F(\text{Right}),$$

where $F(\text{Right})$ can be derived by breaking Right into four bytes and using them as indices into the S-boxes, and then combining the results with addition and XOR operations. It follows that we observe a single bit change in the input or password key results in multiple changes in the resulting ciphertext after nonlinear S-box lookups and mixing. Strong diffusion is then supplied by the chained key schedule procedure, since by the termination of the encryption rounds, the resulting interdependent changes spreads the statistical distribution of output hashes.

Thanks to the inclusion of a salt and a potentially exponential number of rounds in addition to the robust confusion and diffusion properties from $eksblowfish$, bcrypt boasts decent resistance against rainbow table lookups. The repeated mixing increases the computational cost that an adversarial party must incur during an attempt to brute-force or reverse-engineering the function.

yescrypt. Like scrypt, the cryptographic security of yescrypt is based on that of SHA-256, HMAC, and PBKDF2 [22]. The rest of processing, while crucial for increasing the cost of password cracking attacks, may be considered non-cryptographic. Building on top of the robust confusion and diffusion guarantees of scrypt, yescrypt introduces indexed memory access during the ROMix stage. In addition to the password and salt, yescrypt accepts as arguments the following configurable parameters:

N = cost parameter
r = block size
p = parallelization parameter
t = time-cost parameter, increases computation expense
g = garlic parameter, additional memory usage adjustment

In the ROMix stage of the function computation, a large array V is used in an iterative process, such that we start from an initial state X_0 obtained from existing known constants and the input password and salt [22]. Then, iterating from 0 to $N-1$, we have

$$\begin{aligned} &V[i] = X_i \\ \Rightarrow\ &X_{i+1} = F(X_i) \oplus V[\text{index derived from } X_i] \end{aligned}$$

where F is a nonlinear permutation function (e.g., Salsa20/8). By choosing indices derived from the output of the previous step, the function F repeatedly mixes unpredictable data from V. This circular dependency means that if one bit of X_0 changes, it changes the indices selected from V, which in turn affects all subsequent X_i. The result is a chaotic dependency web, ensuring that no attacker can guess or simplify the structure. This construction spreads changes widely, achieving strong diffusion.

Since it is based on scrypt, yescrypt provides resistance against rainbow table attacks through the guarantees demonstrated in its dependencies. The CHF additionally aims to increase computational intensity and cost for adversarial parties by incorporating diffusive re-indexing and chaotic mixing.

Argon2. Argon2 is a memory-hard CHF which utilizes a large memory array M that is filled and then processed through an internal compression function based on a hashing function such as BLAKE2b [14] or SHA-256. Although there exists two different flavors of Argon2, Argon2d and Argon2i, as the latter uses data-independent memory access, which is recommended for password hashing and password-based key derivation, our analysis will focus in particular on Argon2i, which follows the extract-then-expand concept [5]. As arguments, Argon2i accepts two types of inputs: primary and secondary. Primary inputs are message P and nonce S, which are password and salt, respectively, for the password hashing. Secondary arguments include

p = parallelization parameter
τ = tag length
m = memory size parameter
C = iteration count
v = version number
K = secret value, which may serve as key if necessary

First, Argon2i extracts entropy from the password and salt by hashing it. All the other parameters are also added to the input. The variable length inputs P, S, K are prepended with their lengths:

$$H_0 = \mathcal{H}(p, \tau, m, C, v, y, \langle P\rangle, P, \langle S\rangle, S, \langle K\rangle, K)$$

where H is the BLAKE2b function. From here, Argon2i allocates a memory array M of m blocks of 1024 bytes and initializes the first few blocks from the hashed inputs before iteratively filling the rest of the memory array. In each step, Argon2i picks one or two previously generated blocks (based on a pseudorandom indexing scheme) and uses the compression function to combine them, producing a new block. This process continues for several passes over the entire memory. At the heart of the source of the CHF's confusion and diffusion properties lies the compression function G, which is rooted in a construction using ARX operations. G is built upon the BLAKE2b round function, denoted as $\mathcal{P}$. The function operates on the 128-byte inputs, which can be viewed as eight

16-byte registers as follows:

$$\mathcal{P}(A_0, A_1, \ldots, A_7) = (B_0, B_1, \ldots, B_7).$$

The compression function $G(X, Y)$ operates on two 1024-byte blocks X and Y, and first computes $R = X \oplus Y$. Then R is viewed as an 8×8 128-bit matrix: $R_0, R_1, \ldots, R_{63}$. Then P is first applied rowwise, and then columnwise to yield

$$\begin{aligned}(Q_0, Q_1, \ldots, Q_7) &\leftarrow P(R_0, R_1, \ldots, R_7);\\ (Q_8, Q_9, \ldots, Q_{15}) &\leftarrow P(R_8, R_9, \ldots, R_{15});\\ &\vdots\\ (Q_{56}, Q_{57}, \ldots, Q_{63}) &\leftarrow P(R_{56}, R_{57}, \ldots, R_{63});\\ \\ (Z_0, Z_8, Z_{16}, \ldots, Z_{56}) &\leftarrow P(Q_0, Q_8, Q_{16}, \ldots, Q_{56});\\ (Z_1, Z_9, Z_{17}, \ldots, Z_{57}) &\leftarrow P(Q_1, Q_9, Q_{17}, \ldots, Q_{57});\\ &\vdots\\ (Z_7, Z_{15}, Z_{23}, \ldots, Z_{63}) &\leftarrow P(Q_7, Q_{15}, Q_{23}, \ldots, Q_{63});\end{aligned}$$

As the final step, G outputs $Z \in R$:

$$G : (X, Y) \mapsto R = X \oplus Y \mapsto P \mapsto Q \mapsto P \mapsto Z \mapsto Z \in R.$$

Since the CHF performs this series of nonlinear iterative mixing in the compression function, each bit of the output evidently depends on multiple bits of input in a nonpredictable approach. This design ensures confusion: there is no simple, direct relationship between input bits and output bits. It also ensures diffusion: changing any single bit in the primary or secondary input fields will drastically modify the output hash thanks to the intensive mixing and chained computation.

Through the internal compression function and the underlying primitive of BLAKE2b, in additional to the use of salting, Argon2i, like many of the other CHFs examined in this paper, provides reasonable resistance against rainbow table attacks.

3.2 Confusion Index

While the previous discussion focuses on demonstrating the existence of confusion and diffusion properties supported by the mathematical computation in each CHF, our objective in developing a centralized evaluation metric for CHFs necessitates the quantification of these two properties. Recall the definition of confusion, which outlines successful confusion in a cryptographic hashing function as the measure of obfuscation of the relationship between the plaintext password and the output hash. In other words, the CHF must ensure that any bit of the output hash state must depend on multiple input bits. To quantify the measure of confusion present in each CHF, we have developed an index based on the Damerau-Levenshtein edit distance algorithm [10], where the confusion index of a CHF H_i can be computed as follows:

$$I_{C,H_i} = \frac{\mathrm{DL}(P, H_i(P))}{\max(|P|, |H_i(P)|)}, \quad I_{C,H_i} \in [0, 1]$$

where $\mathrm{DL}(P, H_i(P))$ expresses the Damerau-Levenshtein edit distance between the password and its hash, and the division by the length of the longer out of $\{P, H_i(P)\}$ accomplishes normalization of the index. A higher score indicates a greater level of confusion guaranteed by the particular CHF. The Damerau–Levenshtein edit distance function measures how many single-bit edits (insertions, deletions, substitutions, and transpositions of adjacent characters) are required to transform one sequence of input into the other. This specific edit distance function was selected in developing the index as it is traditionally implemented to compute string similarity scores and has been conventionally used as a complexity measure of the relationship between plaintext and ciphertext as it tracks the number of edits required to transform one string into the other. Compared to other edit distance functions such as the Hamming distance or Jaro-Winkler distance functions, the Damerau-Levenshtein function is more robust in its larger set of possible transition operations.

In calculating the indices for the selected set of CHFs, we iterate the following procedure for 100,000 rounds per function: We begin by generating a random 256-bit-long input password, and a corresponding randomized salt held constant across the functions, in addition to any configurable parameters. From here, we hash the password using the CHF, and compute the Damerau-Levenshtein edit distance between the password and the hash before outputting the index post-normalization. The detailed index results can be found in Section 6. Note that a confusion index is not computed for bcrypt due to its empirical weakness despite its popularity in application.

3.3 Diffusion Index

To quantify the measure of diffusion present in each CHF, we have developed an analogous index that is based on the strict avalanche criterion [32]. Recall the definition of diffusion, which entails hiding the statistical relationship between the hash and the plaintext by ensuring that a change in any bit of plaintext should yield changes in multiple bits in the hash. For instance, diffusion ensures that any patterns in the plaintext, such as redundant bits, are not apparent in the output hash. In the most ideal case, a given CHF achieves the strict avalanche criterion. The strict avalanche criterion (SAC) is a formalization of the avalanche effect. It is satisfied if, whenever a single input bit is complemented, each of the output bits changes with a 50% probability. As such, we define a diffusion index I_{D,H_i}:

$$I_{D,H_i}(P, j) = 1 - \frac{\left|d_j - \frac{N}{2}\right|}{\frac{N}{2}} = \frac{2d_j}{N}, \quad I_{D,H_i} \in [0, 1]$$

where d_j is the difference in number of bits between the hash of P and P with bit j flipped, and N is the total number of bits of the output hash (256 in testing). A higher score indicates a greater level of diffusion guaranteed by the particular CHF.

In calculating the indices for the selected set of CHFs, we perform the following procedure per function over the length of the 256-bit password, flipping bit $j : j \in [0, 255]$ per iteration: We begin by generating a random 256-bit-long input password, and a corresponding randomized salt held constant across the functions, in addition to any configurable parameters. From here, we apply the CHF to compute $H_i(P_j)$, then flip bit j before computing the hash $H_i(P'_j)$ and calculating the index based on the SAC. The detailed index results of each CHF can be found in Section 6. Note that a diffusion index is not computed for bcrypt due to its empirical weakness despite its popularity in application.

3.4 Resistance to Specialized Hardware

Based on the evaluation metrics for determining whether a given CHF is resistant to specialized hardware, as described in the methodology, we derive classifications for each CHF of interest, with the justification for each classification provided in Table 1.

CHF	Memory Hardness	Notes	Sources
SHA-256	None	Low, non-tunable, memory utilization	[9]
PBKDF2	None	Low, non-tunable, memory utilization	[33]
scrypt	Mediocre	Vulnerable to Time-Memory Tradeoff Attacks	[2, 27]
Yescrypt	Acceptable	Vulnerable to Time-Memory Tradeoff Attacks when the smallest secure parameters are used	[6]
Argon2	Acceptable	Password-independent variant (Argon2i) exhibits a low re-computation penalty, enabling attacks resulting in up to quadratic reduction in memory usage	[5, 9]

Table 1: Resistance to specialized hardware attacks as measured by the property of memory-hardness for CHFs of interest.

In particular, SHA-256 and PBKDF2 are deemed as having little to no memory hardness guarantees, given that they require very little memory to compute, and have no parameter that enables scaling memory utilization, and are highly parallelizable and vulnerable to specialized hardware attacks [9, 33]. Scrypt is deemed to have mediocre memory hardness guarantees, given that while it does possess tunable memory hardness, it is nonetheless susceptible to Time-Memory Tradeoff attacks, which significantly reduces resistance to specialized hardware [2, 27]. Yescrypt is deemed to be acceptable, given that although it susceptible to Time-Memory Tradeoff attacks, this is only the case for the smallest secure parameters, and thus nonetheless possesses memory hardness guarantees for larger parameter values [6]. Finally, Argon2 is deemed to be acceptable, given that while specific attacks can result in a reduction in the amount of memory used – when compared to the guarantee – this reduction is at best quadratic, and thus, can be compensated by increasing the parameters to ensure that specialized hardware nonetheless is incapable of significant speed-ups over normal hash computation [5, 9].

4 Empirical Analysis of CHF Security and Parameters

To supplement the theoretical analysis of each of the CHFs, we evaluated a series of empirical benchmarks to see how they compared in their default or recommended configurations, and how they could be tuned to provide the best time-memory and security compromise.

While each of the CHFs may be strong as an algorithm, what determines its strength in the real world is the common implementations, and a poorly configured CHF will not be much better than a non-cryptographic hash function.

4.1 Testing setup

Two hardware stacks were used for testing, to simulate an older server chip and a recent (but not top-of-the-line) server chip. These configurations were:

(1) 6x AMD Ryzen 5 5600G with 4GB RAM running under a Hyper-V Generation 2 virtual machine
(2) 3x Unspecified circa 2015 Intel Xeon processor with 6GB RAM running under KVM on a public cloud

Both were running Ubuntu 22.04.4 LTS with Python 3.10.12 and g++-11. C++ test programs were compiled with `-O2 -march=native`. Ubuntu 22.04 was chosen to represent a stable and mature LTS release that is still fully supported. Both came with `libxcrypt` and `libcrypto` (OpenSSL crypto library) preinstalled, however, `libssl-dev` needed to be installed through the system package manager to acquire the header files required to compile the test programs.

Both hardware configurations were used to run the default + recommended configurations to determine their performance over a range of hardware performance, while only the more powerful first configuration was used to test configurations as the scaling would apply the same way to older hardware.

Additionally, `libxcrypt` required hugepage support for scrypt and yescrypt beyond their default configurations, and the testing environments were configured with 200 hugepages of 2MB each.

All of the timed tests were performed by hashing a list of (32) passwords in a list and accumulating the time required to set up and perform the hash (but not auxilliary operations like reading the passwords from a file) using `std::chrono::high_resolution_clock` (C++) and `time.process_time` (Python).

For Python, the memory tests were done by polling the memory usage every 0.05s while hashes were being computed, and subtracting the initial memory usage from the maximum memory usage. In C++, the memory tests were done by checking the maximal resident set size (maxRSS) as reported by getrusage(2) of a forked child process before and after hashing, as well as continuously polling for hugepage usage.

4.2 Default and Recommended Configurations

In determining the common configurations, we looked at specification or package defaults, recommendations from important organizations (such as NIST), as well as the defaults used by software that use these algorithms. They are:

Argon2: Argon2 uses 3 paramters, m, t, and p to control the memory cost, time cost, and parallelism, respectively. As well, it has three versions, Argon2i, Argon2d, and Argon2id. We used Argon2id with $m = 65536$, $t = 3$, and $p = 4$ as used in python's argon2-cffi Argon2 bindings

PBKDF2: PBKDF2 uses an iteration count to control the number of iterations of the underlying hash algorithm to use. We used

600, 000 (600k) and 1, 000, 000 (1m) iterations representing OWASP's recommendation as well as the defaults in recent versions of Django. We also used 100, 000 (100k) iterations to represent weaker (older) hashes that could be found in systems that haven't updated their iteration count recently.

Scrypt: Scrypt uses 3 parameters, n, r, and p to control the work factor, internal hash width, and parallelism, respectively. We had three configurations of $\{2^{17}, 8, 1\}$, $\{2^{15}, 8, 3\}$, and $\{2^{13}, 8, 10\}$ representing OWASP's memory, balanced, and CPU recommendations, as well as the default in Flask.

Yescrypt: Because yescrypt is a scrypt derivative, it uses mostly the same parameters as scrypt. We used $\{2^{12}, 32, 1\}$ as used by passwd(1) on Ubuntu 22.04+, amongst other modern Linux distributions. The time-cost and garlic parameters were not available in this libxcrypt implementation.

bcrypt: Bcrypt was not tested due to its known weakness to FPGAs, as well as numerous recommendations not to use it, including on the Python module's first line and on OWASP's password storage guide.[3, 31] It is taken to be empirically weak.

When required, a 128-bit salt generated from reading 16-18 bytes from `/dev/urandom` was used, as commonly recommended from a variety of online sources.

4.3 Default Configuration: Hash Time

In the first set of benchmarks, we determined the time required to hash passwords using the above configurations of each CHF, along with a baseline of regular sha256 and plaintext. These would also be the same rate a naive brute-force attack would hash at, without specialized hardware or parallelism. The goal is to have the algorithm take between 0.1s and 0.5s per hash [24]. We used the top 32 passwords from rockyou.txt to test the runtimes, so the target runtimes should be 3.2 to 16 seconds. Each CHF was evaluated in both C++ and Python, two of the most popular programming languages, except for yescrypt in Python due to a lack of major package supporting it. These results are summarized in Figures 1 and 2.

Between the two languages, we see roughly the same graph shape, but with C++ generally hashing faster. The exception to this is PBKDF2, which took roughly the same amount of time in each language, probably because the Python implementation calls directly into the OpenSSL library.[12] Through these graphs, we can see an obvious issue with the 0.1-0.5s recommendation: scrypt (memory-hard) is on the lower end of that range on the more powerful AMD hardware under C++ (0.17s per hash), but 34% above the high end on the Intel system under Python (0.67s per hash).

scrypt and yescrypt shine in this series of benchmarks as the performance ratio between the two systems is the lowest compared to the other algorithms. This means that it is relatively more resistant to increases in hardware performance, and can last longer in a production system before the parameters become too weak over time. Argon2 and PBKDF2 may be acceptable depending on hardware, but tuning specific to each system is recommended to find the sweet spot between acceptable performance and brute force protection.

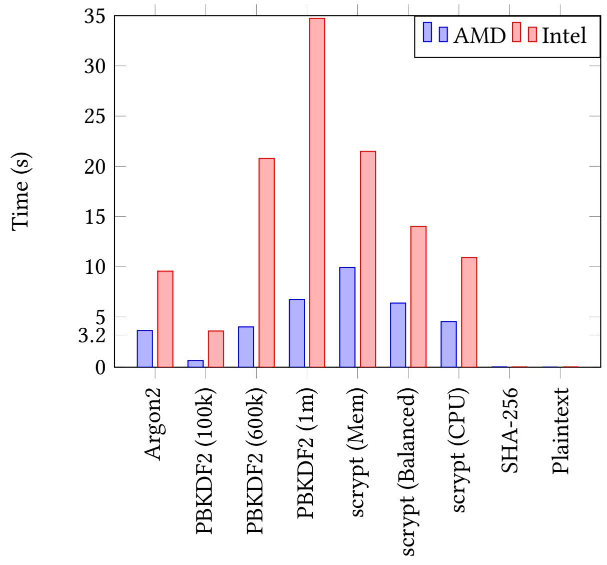

Figure 1: Timings on the two hardware configurations, in Python

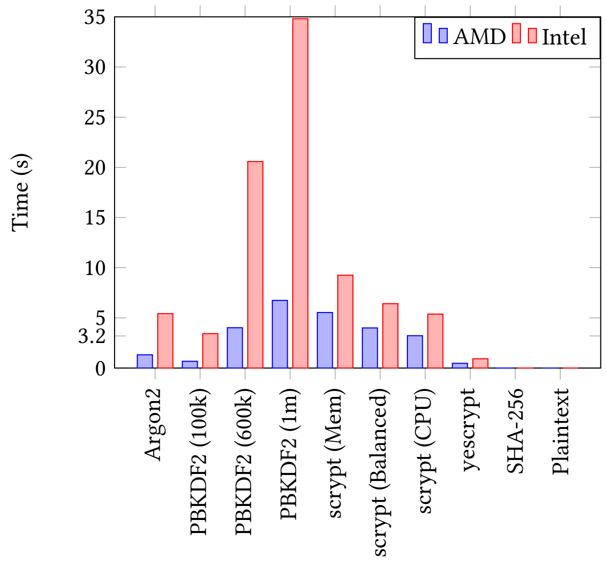

Figure 2: Timings on the two hardware configurations, in C++

4.4 Default Configuration: Memory Usage

In the second set of benchmarks, we determined the memory usage required to hash passwords using the above configurations of each CHF, along with a baseline of regular SHA-256 and plaintext (Figure 3). Memory hardness is a useful property for a CHF to have to defend against specialized hardware cracking, and is also an important consideration for developers tareting IoT platforms with limited memory. Operators should increase the memory cost for their given hardware. Each CHF was evaluated in both C++ and Python, two of the most popular programming languages, except for yescrypt in Python due to a lack of major package supporting it.

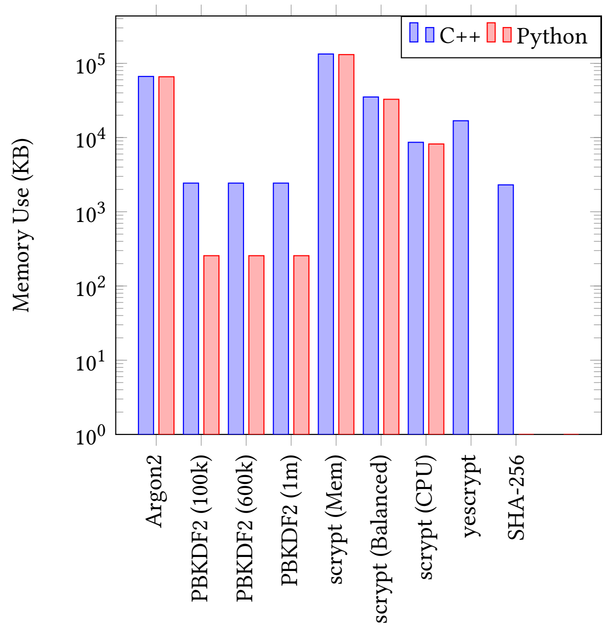

Figure 3: Dynamic memory usage in C++ and Python

Here, we can observe a clear linear dependence between scrypt's N factor and memory usage, and Argon2 appears to be relatively memory-hard by default. Increasing PBKDF2's iteration count does not affect its memory usage, making it potentially susceptible to ASIC attacks in the future. No data was collected for yescrypt for Python due to a lack of a reputable library for it.

This memory data was collected for dynamic memory usage, which includes heap space, stack space, and other dynamic memory such as that allocated by mmap(2). In addition to that, we can also consider library memory usage as loaded by ld(1), but this will differ by library, platform, version, and architecture. For our specific testing environments, with the default preinstalled versions of libssl and libxcrypt, and the latest version of libargon2 on the x86-64 architecture, the measured library usage was 441 KB for `libcrypto.so.1` (for the OpenSSL CHF implementations), 10 KB for `libargon.so.1`, and 50 KB for `libcrypt.so.1` (for the libxcrypt CHF implementations).

4.5 Strengthening by Increasing Parameters

Given that computing time only decreases as hardware gets more powerful, many years-old recommendations are now too weak. For example, LastPass, a popular password manager, was only using 100,100 iterations of PBKDF2 just a few years ago[18], while the recommendation recently has grown to at least 600,000.[31][19] The purpose of these benchmarks is to determine how time- and memory-hardness scale with the parameters of the different CHFs.

For Argon2, increasing the memory cost parameter is directly proportional to an increasing memory and time cost as seen in Figure 4.

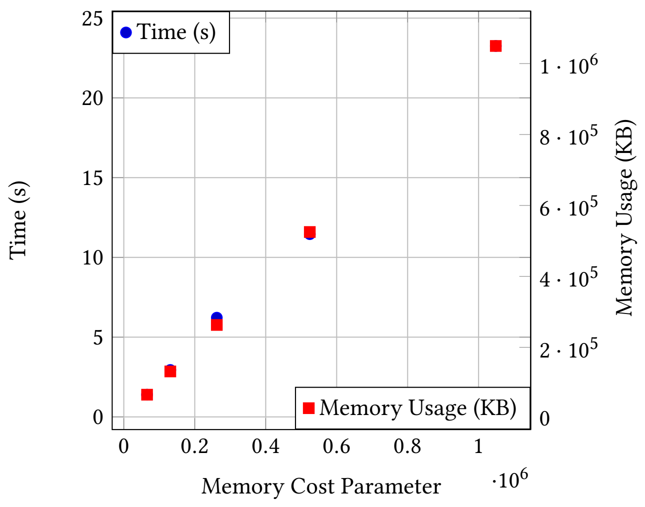

Figure 4: Effects of changing the memory cost parameter of Argon2 on time and memory usage

However, increasing the time cost parameter is only directly proportional to an increasing time cost, as seen in Figure 5.

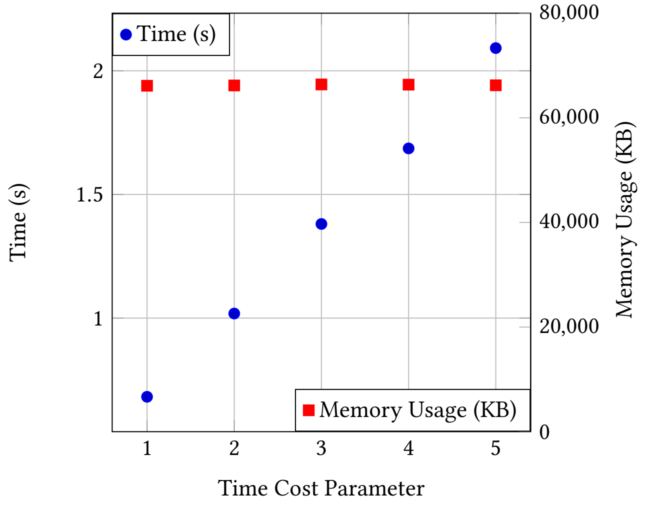

Figure 5: Effects of changing the time cost parameter of Argon2 on time and memory usage

PBKDF2's iteration count is analogous to Argon2's time cost parameter, with an linear increase in iterations directly proportional to a linear increase in time taken, with no impact on memory usage. This makes sense as it is just repeated computing a hash a a few more times.

For scrypt and yescrypt, only the N parameter was changed as this is the recommendation. The memory usage was directly 1:1 with the N parameter (plus a few hundred KB of overhead), however, it has to be a power of 2. The time was directly correlated with the N parameter, although scrypt's $N = 4096$ takes time closer to $N = 8192$ than expected. Notably, unlike the other algorithms, increasing N past the defaults required hugepages to be enabled, which may not be feasible or possible on every platform. Changing N was analogous to changing Argon2's memory cost parameter.

Target	Time Change	Memory Change
Argon2 memcost	Linear	Linear
Argon2 timecost	Linear	No Change
PBKDF2 iters	Linear	No Change
scrypt N	Linear (power of 2)	Linear (power of 2)
yescrypt N	Linear (power of 2)	Linear (power of 2)

Table 2: Summary of effects of changing CHF parameters on time and memory usage

In essence, to determine the best parameters to use for a given hardware configuration, the memory cost parameter should be increased until either the time or memory usage becomes unreasonable, and then if the time taken is still too little and the CHF has a time parameter, the time parameter should be increased until the time becomes reasonable. None of the hash functions had a dedicated memory-only parameter.

4.6 Other Considerations

A test was also performed to look for any relationship between password length and hashing time or memory cost. There was no correlation at all for any of the CHFs, and they all took constant time relative to the length of the password in 4-character increments from 4 to 128 characters (a reasonable limit for password lengths).

5 Ease of Deployment for CHFs

Based on the evaluation metrics for accessibility and implementation as described in the methodology, we derive Accessibility scores and Ease-of-Implementation Scores for each CHFs of interest, summarizing the results in Table 6.

Target	Accessibility Score	Ease-of-Implementation Score
Plaintext	1.000	10
SHA-256	1.000	8
PBKDF2	0.9856	8
scrypt	0.9856	6
Yescrypt	0.5189	2
Argon2	0.9856	7

Table 3: Accessibility and Ease-of-Implementation Scores for CHFs of interest alongside a plaintext baseline.

5.1 Accessibility Scores

The Normalized IEEE Spectrum Popularity scores used to calculate the final Accessibility Score can be found in Table 6. In particular, after eliminating programming languages used primarily for statistical analysis, mathematics, as well as markup languages, we are left with a list of 13 Programming Languages which we assess each CHF's accessibility on. Through the evaluation, featured in Table 5, we determine that other than Yescrypt, which lacks even third-party implementations for numerous programming languages, all other CHFs are roughly equivalent in accessibility for the 13 examined programming languages, which represent among the most frequently used programming languages within the software development community. In particular, given that the 13 examined programming languages encompass a majority of popular programming languages, across a wide range of applications, the Accessibility score provides insight onto the proportion of software in which each CHF could be utilized. For instance, while most of the CHFs can be relatively easily utilized in most projects, yescrypt is only available for use in about half of software projects, when based on the weighted popularity of the examined programming languages. However, it should be noted that the Accessibility score only examines the theoretical accessibility of a CHF for differing programming languages; it does not take into consideration specifics on difficulty of implementation; for example, a CHF might be classified as “Widely Available” for a given programming language due to its availability in a popular library, but nonetheless be difficult to implement from a practical standpoint due to details that can only be observed and assessed when the CHF is implemented.

5.2 Ease-of-Implementation Score

While it is beyond the scope of this study to implement CHFs in over a dozen programming languages to assess implementation difficulties for each CHF and language combination, we evaluate the true implementation difficulty of all target CHFs for Python and C++, which are the languages upon which the benchmarks are conducted. The Ease-of-Implementation Scores for each CHF, alongside the observations which motivate the score, can be found in Table 7. In particular it is noted that the Ease-of-Implementation Score can reveal more granular and nuanced insight into the efficacy of utilizing any given CHF as it offers detailed insight into the implementation and deployment process which can only be observed and noted by trying to use the CHFs.

6 Cumulative Results

To summarize the various metrics of the centralized evaluation, we create a summary table containing key insights from each metric, featured in Table 4.

7 Evaluation

It is inherently difficult to evaluate a set of prominent cryptographic hash functions as they each have been developed and designed with considerations for potentially different performance-security trade-offs. However, given the results from this paper, we would recommend scrypt as a primary choice due to its default strength and prevalence, and Argon2 as a secondary choice due to its customizability, memory hardness, and relative ease of deployment. PBKDF2 can be used for memory-constrained environments as a hard hash function to crack without specialized hardware, with the additional benefit of ease-of-deployment. At a minimal, scrypt, Argon2, and PBKDF2 all provide a foundational guarantee of confusion and diffusion to ensure that their respective operations are noninvertible and cryptographically secure, as demonstrated by the similar confusion and diffusion index results.

*: must be a power of 2

	Plaintext	SHA-256	PBKDF2 600K	PBKDF2 1M	scrypt	yescrypt	Argon2
Confusion	0	0.294	0.297	0.294	0.290	0.293	0.295
Diffusion	0	1.000	0.995	0.999	1.000	1.000	0.998
Resistance to Look-up Attacks	No	No	Yes	Yes	Yes	Yes	Yes
Memory Hardness	None	None	None	None	Mediocre	Acceptable	Acceptable
Hash Time (32 passwords, seconds)	$5 * 10^{-7}$	$2.4 * 10^{-5}$	4.02	6.74	4.00	0.47	1.32
Memory Usage (MB)	0	2.2	2.4	2.4	34.4	16.4	65
Time Change by Parameter	NA	NA	Linear	Linear	Linear*	Linear*	Linear
Memory Utilization change by Parameter	NA	NA	Constant	Constant	Linear*	Linear*	Linear
Accessibility Score	1.000	1.000	0.9856	0.9856	0.9856	0.5189	0.9856
Ease-of-Implementation Score	10	8	8	8	6	2	7

Table 4: Cumulative Summary of Evaluated Metrics

7.1 Benchmarking

When comparing the default and/or recommended configurations for each CHF, all of the common configurations of Argon2, PBKDF2, scrypt, and yescrypt were within the target range of 0.1 to 0.5 seconds per hash in some configuration. However, if the goal is to protect against offline brute forcing, then taking C++ performance on the stronger AMD platform, only PBKDF2 and scrypt were strong enough to still require at least 0.1s per hash.

On the memory-hardness side, yescrypt, scrypt, and Argon2 both had good memory hardness out of the box, with options to increase it more, whereas PBKDF2's memory usage was locked in place. Argon2, yescrypt, and scrypt all had options to increase both the time and memory hardness linearly, while PBKDF2 only had a parameter to increase time only.

We decided to use C++ and Python to represent the fast and slow end of popular languages, ensuring that most other languages would lie somewhere in the middle performance-wise. Additionally, only 32 passwords were used per trial, because many of the algorithms perform many repeated operations, and at 32 passwords the run-to-run deviation was small enough to trust that they were close to the true mean times. Some difficulty was encountered in trying to find implementations and their documentation, as well as in debugging them. In practice, it is expected that implementors of these CHFs may also run into similar issues, and in the case of hugepage support for scrypt and yescrypt, might choose a weaker than optimal configuration simply because they get an error when increasing the parameters.

Due to this combination of factors, empirically, we would rank these algorithms from best to worst (as tested) as scrypt, Argon2, yescrypt, PBKDF2. This roughly matches the expected results, with the exception being yescrypt as it was meant to be a better scrypt. This could be due to the fact that it is one of the newer algorithms and hasn't had enough market share to develop a good set of defaults and packages in other languages.

7.2 Mathematical Analysis

The theoretical security analysis we perform in this paper can be interpreted as a two-stage construction to encapsulate the qualitative and quantitative facets of ensuring soundness in each of the CHFs. In particular, we choose to begin by discussing the computational mechanisms, if present, in the CHFs that provide confusion and diffusion, which builds upon Claude Shannon's prescribed definition of security in hash functions and ciphers. Alternative systems of defining theoretical security may be considered, which may incorporate a stronger emphasis on other more specific criteria, such as pre-image resistance. Our choice in applying Shannon's definition stems from a prioritization of the foundational conditions required for a CHF to be considered a cryptographically complete and sound hashing scheme. Furthermore, basing our analysis on the two central branches of confusion and diffusion enables us to readily develop systems of quantification.

From the index computation, we observe that with regards to confusion and diffusion, most of the CHFs exhibit comparable levels of the two properties, with the exception of the plaintext control group. In particularly, SHA-256, both versions of PBKDF2, scrypt, yescrypt, Argon2 all yield near-ideal levels of diffusion, meaning that per bit of change in the input password, approximately 50% of bits in the output hash are also changed.

These results corroborate the qualitative deep dive into the mathematical operations used by each CHF, since most of them build on top of the confusion and diffusion foundation provided by SHA-256 as an underlying primitive. However, it is important to note that while the conditions of confusion and diffusion are critical to evaluating the theoretical security of CHFs, they do not fully capture facets of the functions such as computational expense. In a number of these CHFs, extensive iterative chaining and compression functions are utilized to increase computation time, which may be interpreted as both a defense against brute-force rainbow table lookup attacks and also a deterrent to performance.

The slight deviations and differences presented in Table 4 can be attributed to the randomness introduced during the computation and crystallization of the index numbers.

7.3 Ease of Deployment

Based on a combination of the Accessibility Score and the Ease-of-Implementation score, we determine that PBKDF2 is the easiest to deploy, followed by Argon2, then scrypt, and finally by yescrypt. We notice that among the Accessibility Scores for these four CHFs, only yescrypt's score is different – specifically worse – than the Accessibility Scores determined for the other CHFs. Indeed, we find that the Accessibility Score, which examines a low-granularity categorization of the CHFs over a number of programming languages, conducted at the current scale and level of granularity, offers significantly less distinguishability between the relative ease-of-deployment for differing CHFs, when compared to the Ease-of-Implementation score, which not only has higher granularity, but is based on the laborious process of actually implementing each CHF and noting the relative difficulties of implementing each CHF. Indeed, it is to be expected that the actual process of implementing the CHF will offer the best insight as to which CHF is the easiest to deploy.

8 Future Works

Having established a framework upon which it is possible to holistically judge various differing CHFs, it would be of great value to expand this CHF to be more fully comprehensive, examining less popular CHFs, so as to determine if a promising candidate has been mistakenly overlooked in the past. Expanding the framework would also allow for an identification of potential edge-cases – CHFs for which the framework is an imperfect evaluation – which can then be used to further refine the framework to be more accurate and reflective across all CHFs evaluated.

In addition to expanding the scope of the framework however, there are also various points for improvement which would serve to make the framework more rigorous and useful. One such improvement is to increase the accuracy of the Accessibility Score. This can be accomplished by increasing the granularity of the categorization for each CHF-Language pair: the use of text-mining or similar techniques could enable the collection of insights from developers, documentation, and various CHF implementations which can inform a more accurate categorization. Notably, even if a more fine-grained categorization had been implemented in this study, we would have been limited by the amount of data we could collect to properly inform the categorization. As such, collecting sufficient information to inform the categorization is of key interest. Another potential improvement is to expand the scope of the mathematical assessment of the various CHFs of interest. In particular, it would be beneficial to assess each CHF using a variety of numerical criterion in addition to the strict avalanche criterion, to give a more holistic mathematical evaluation. Furthermore, there is potential in examining alternative, potentially more comprehensive and representative, methods for the index calculations.

References

[1] Abdulaziz Ali Alkandari, Imad Fakhri Al-Shaikhli, and Mohammad A. Alahmad. 2013. Cryptographic Hash Function: A High Level View. In *2013 International Conference on Informatics and Creative Multimedia*. 128–134. https://doi.org/10.1109/ICICM.2013.29

[2] Joël Alwen, Binyi Chen, Krzysztof Pietrzak, Leonid Reyzin, and Stefano Tessaro. 2016. Scrypt is Maximally Memory-Hard. Cryptology ePrint Archive, Paper 2016/989. https://eprint.iacr.org/2016/989 https://eprint.iacr.org/2016/989.

[3] Python Cryptographic Authority. 2024. bcrypt. https://pypi.org/project/bcrypt/ https://pypi.org/project/bcrypt/.

[4] Gustavo Banegas, Koen Zandberg, Adrian Herrmann, Emmanuel Baccelli, and Benjamin Smith. 2021. Quantum-Resistant Security for Software Updates on Low-power Networked Embedded Devices. *arXiv preprint arXiv:2106.05577* (2021).

[5] 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

[6] Alex Biryukov and Dmitry Khovratovich. 2015. Tradeoff Cryptanalysis of Memory-Hard Functions. Cryptology ePrint Archive, Paper 2015/227. https://eprint.iacr.org/2015/227 https://eprint.iacr.org/2015/227.

[7] Joseph Bonneau, Cormac Herley, Paul C. van Oorschot, and Frank Stajano. 2012. The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes. In *2012 IEEE Symposium on Security and Privacy*. 553–567. https://doi.org/10.1109/SP.2012.44

[8] Stephen Cass. 2024. The Top Programming Languages 2024. *IEEE Spectrum* (2024). https://spectrum.ieee.org/top-programming-languages-2024

[9] Donghoon Chang, Arpan Jati, Sweta Mishra, and Somitra Kumar Sanadhya. 2017. Cryptanalytic Time-Memory Tradeoff for Password Hashing Schemes. Cryptology ePrint Archive, Paper 2017/603. https://eprint.iacr.org/2017/603 https://eprint.iacr.org/2017/603.

[10] Fred J. Damerau. 1964. A technique for computer detection and correction of spelling errors. *Commun. ACM* 7, 3 (March 1964), 171–176. https://doi.org/10.1145/363958.363994

[11] Markus Dürmuth and Thorsten Kranz. 2014. On Password Guessing with GPUs and FPGAs. *Horst Görtz Institute for IT-Security (HGI), Ruhr-University Bochum, Germany* (2014).

[12] Python Software Foundation. 2024. hashlib — Secure hashes and message digests. https://docs.python.org/3/library/hashlib.html https://docs.python.org/3/library/hashlib.html.

[13] Praveen Gauravaram. 2012. Security Analysis of salt||password Hashes. In *2012 International Conference on Advanced Computer Science Applications and Technologies (ACSAT)*. 25–30. https://doi.org/10.1109/ACSAT.2012.49

[14] Jian Guo, Pierre Karpman, Ivica Nikolic, Lei Wang, and Shuang Wu. 2014. Analysis of BLAKE2. *IACR Cryptol. ePrint Arch.* 2013 (2014), 467. https://api.semanticscholar.org/CorpusID:10099494

[15] Peter Gutmann and Bruce Schneier. 1999. *Description of the Blowfish Cipher*. Internet-Draft draft-schneier-blowfish-00. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-schneier-blowfish/00/ Work in Progress.

[16] George Hatzivasilis, Ioannis Papaefstathiou, and Charalampos Manifavas. 2015. Password Hashing Competition - Survey and Benchmark. Cryptology ePrint Archive, Paper 2015/265. https://eprint.iacr.org/2015/265

[17] Gaurav Kodwani, Shashank Arora, and Pradeep K. Atrey. 2021. On Security of Key Derivation Functions in Password-based Cryptography. In *2021 IEEE International Conference on Cyber Security and Resilience (CSR)*. 109–114. https://doi.org/10.1109/CSR51186.2021.9527961

[18] LastPass. 2019. LastPass Technical Whitepaper. https://assets.cdngetgo.com/1d/ee/d051d8f743b08f83ee8f3449c15d/lastpass-technical-whitepaper.pdf https://assets.cdngetgo.com/1d/ee/d051d8f743b08f83ee8f3449c15d/lastpass-technical-whitepaper.pdf.

[19] David Lord. 2024. increase pbkdf2 iterations. https://github.com/pallets/werkzeug/issues/2969 https://github.com/pallets/werkzeug/issues/2969.

[20] Wahome Macharia. 2021. Cryptographic Hash Functions. (05 2021).

[21] Kathleen Moriarty, Burt Kaliski, and Andreas Rusch. 2017. PKCS #5: Password-Based Cryptography Specification Version 2.1. RFC 8018. https://doi.org/10.17487/RFC8018

[22] Samuel Neves. 2015. yescrypt - a Password Hashing Competition submission. https://api.semanticscholar.org/CorpusID:111384133

[23] Marut Pandya. 2024. Performance Evaluation of Hashing Algorithms on Commodity Hardware. *arXiv preprint arXiv:2407.08284* (2024).

[24] Colin Percival. 2009. Stronger Key Derivation via Sequential Memory-Hard Functions. In *Proceedings of BSDCan'09*. https://www.tarsnap.com/scrypt/scrypt.pdf

[25] Colin Percival and Simon Josefsson. 2016. The scrypt Password-Based Key Derivation Function. RFC 7914. https://doi.org/10.17487/RFC7914

[26] Niels Provos and David Mazières. 1999. A Future-Adaptable Password Scheme. In *1999 USENIX Annual Technical Conference (USENIX ATC 99)*. USENIX Association, Monterey, CA. https://www.usenix.org/conference/1999-usenix-annual-technical-conference/future-adaptable-password-scheme

[27] Ayse Nurdan Saran. 2024. On time-memory trade-offs for password hashing schemes. *Frontiers in Computer Science* 6 (2024). https://doi.org/10.3389/fcomp.2024.1368362

[28] A. Arul Lawernce Selvakumar and R. S. Ratastogi. 2014. Study the function of building blocks in SHA Family. arXiv:1402.1314 [cs.CR] https://arxiv.org/abs/1402.1314

[29] C. E. Shannon. 1949. Communication theory of secrecy systems. *The Bell System Technical Journal* 28, 4 (1949), 656–715. https://doi.org/10.1002/j.1538-7305.1949.tb00928.x

[30] Yang Su, Yansong Gao, Omid Kavehei, and Damith C. Ranasinghe. 2019. Hash Functions and Benchmarks for Resource Constrained Passive Devices: A Preliminary Study. *arXiv preprint arXiv:1902.03040* (2019).

[31] OWASP Cheat Sheet Series Team. 2024. Pasword Storage. https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html.

[32] Riley Vaughn and Mike Borowczak. 2024. Strict Avalanche Criterion of SHA-256 and Sub-Function-Removed Variants. *Cryptography* 8, 3 (2024). https://doi.org/10.3390/cryptography8030040

[33] Jos Wetzels. 2016. Open Sesame: The Password Hashing Competition and Argon2. *arXiv preprint arXiv:1602.03097* (2016).

A Supplemental Figures

Language	IEEE Spectrum Popularity Score	Normalized Popularity Score
C	0.1989	0.0580
C++	0.3749	0.1093
Python	1	0.2917
Java	0.4855	0.1416
JavaScript (Node.js)	0.4451	0.1298
PHP	0.1196	0.0349
Ruby	0.0632	0.0184
Go	0.2052	0.0599
Rust	0.1506	0.0439
C# (.NET)	0.2089	0.0609
Swift	0.0495	0.0144
Kotlin	0.0656	0.0191
Dart	0.0615	0.0179

Table 5: IEEE Explore Programming Language Spectrum Popularity Score and Normalized Score for 13 Common Programming Languages. The Normalized Popularity Score is calculated by dividing a given Language's Spectrum Popularity Score with the sum of Popularity scores across the 13 selected Programming Languages.

Language	SHA-256	PBKDF2	scrypt	yescrypt	Argon2
C	OpenSSL [1]	OpenSSL [2]	OpenSSL [3]	Openwall [4]	PHC Argon2 Reference [5]
C++	OpenSSL [1]	OpenSSL [2]	OpenSSL [3]	Openwall [4]	PHC Argon2 Reference [5]
Python	Hashlib [6]	Hashlib [6]	Hashlib [6]	pyescrypt [7]	Passlib [8]
Java	BouncyCastle [9]	BouncyCastle [10]	BouncyCastle [10]	Custom	BouncyCastle [11]
JavaScript (Node.js)	Crypto [12]	Crypto [12]	Crypto [12]	3rd Party Defuse [13]	Argon2-browser [14]
PHP	Hash [15]	hash_pbkdf2 [16]	Sodium [17]	3rd Party Defuse [13]	password_hash [18]
Ruby	Digest [19]	OpenSSL::KDF [20]	OpenSSL::KDF [20]	3rd Party Defuse [13]	argon2 gem [21]
Go	Crypto [22]	Crypto [23]	Crypto [23]	Openwall [4]	Crypto [24]
Rust	Ring Crate [24]	Ring Crate [25]	scrypt Crate [26]	3rd Party RustCrypto [27]	Argon2 Crate [28]
C# (.NET)	System.Security. Cryptography [29]	System.Security. Cryptography [29]	BouncyCastle Wrapper [30]	3rd Party petrsnm [31]	BouncyCastle Wrapper [32]
Swift	CryptoKit [33]	3rd party kvzayanowski [34]	3rd party greymass [35]	Custom	3rd Party tmthecoder [36]
Kotlin	MessageDigest [37]	SecretKey-Factory [38]	BouncyCastle Wrapper [39]	Custom	BouncyCastle Wrapper [39]
Dart	Cryptography [40]	Cryptography [40]	Hashlib [41]	Custom	Cryptography [40]
Overall Score	1.000	0.9856	0.9856	0.5189	0.9856

Table 6: Accessibility Score of various CHFs as justified by the listed documentation, libraries, or repositories. Entries color-coded green are classified as "Widely Available," entries color-coded orange are classified as "Limited 3rd Party," and entries color-coded red are classified as requiring "Custom Implementation." Reference links for each of the documentation, libraries, or repositories can be found in Table 8.

Score out of 10	**C++ Notes**	**Python Notes**
yescrypt 1+1=2	Available through the crypt interface of libxcrypt which is available on most Linux systems. However, no formal documentation could be found, and sources such as StackOverflow and Wikipedia were necessary to aid in determining the appropriate implementation methodology. The hash format is very niche and it was found that for n > 4096, the crypt function would return `ENOMEM` if hugepages was not enabled.	Only a single package that provides bindings for yescrypt in python was found. Even then, the corresponding repository lacked documentation, making implementation challenging.
scrypt 2+4=6	Available through the crypt interface of libxcrypt which is available on most Linux systems. However, no formal documentation could be found, and sources such as StackOverflow and Wikipedia were necessary to aid in determining the appropriate implementation methodology. The hash format is however, less niche than yescrypt, although it was found that for n > 8192, the crypt function would return `ENOMEM` if hugepages was not enabled.	Scrypt is the new default for Flask and it is very accessible through the built-in hashlib module. One flaw was discovered in that maxmem must be set for n > 2^{13} otherwise the implementation throws an error. This error is not formally documented anywhere.
Argon2 3+4=7	Requires libargon2, which must be built from source, however the process is straightforward. Although it was noted that documentation was in some cases lacking in specificity.	Available through the argon2-cffi or cryptography modules, and is thus easy to implement.
PBKDF2 3+5=8	Available through the EVP interface in the OpenSSL library which is installed on most Linux systems. However, the documentation is somewhat lacking.	PBKDF2 is the default for Django and used to be the default for Flask. As such, it is easily accessible through the built-in hashlib module.
SHA-256 3+5=8	Available through many interfaces in the OpenSSL library which is installed on most Linux systems. Very straightforwards to implement, but it is noted that the documentation may be overwhelming at times.	Very accessible through the built-in hashlib module.
Plaintext 10	Trivial	Trivial

Table 7: Comparison of Ease-of-Implementation Scores for CHFs of interest in C++ and Python.

No.	URL
1	https://docs.openssl.org/3.1/man3/SHA256_Init/
2	https://docs.openssl.org/3.1/man7/EVP_KDF-PBKDF2/
3	https://docs.openssl.org/3.1/man7/EVP_KDF-SCRYPT/
4	https://github.com/openwall/yescrypt
5	https://github.com/P-H-C/phc-winner-argon2
6	https://docs.python.org/3/library/hashlib.html
7	https://github.com/0xcb/pyescrypt
8	https://passlib.readthedocs.io/en/stable/lib/passlib.hash.argon2.html
9	https://downloads.bouncycastle.org/java/docs/bcprov-jdk14-javadoc/org/bouncycastle/crypto/digests/SHA256Digest.html
10	https://downloads.bouncycastle.org/java/docs/bcprov-jdk14-javadoc/org/bouncycastle/crypto/util/package-summary.html
11	https://downloads.bouncycastle.org/java/docs/bcprov-jdk14-javadoc/org/bouncycastle/crypto/generators/Argon2BytesGenerator.html
12	https://nodejs.org/api/crypto.html
13	https://www.npmjs.com/package/argon2-browser
14	https://www.php.net/manual/en/function.hash-algos.php
15	https://www.php.net/manual/en/function.hash-pbkdf2.php
16	https://www.php.net/manual/en/book.sodium.php
17	https://github.com/defuse/yescrypt
18	https://www.php.net/manual/en/function.password-hash.php
19	https://ruby-doc.org/stdlib-2.4.0/libdoc/digest/rdoc/Digest/SHA2.html
20	https://ruby-doc.org/stdlib-2.4.1/libdoc/openssl/rdoc/OpenSSL/KDF.html
21	https://rubygems.org/gems/argon2/versions/0.0.2?locale=en
22	https://pkg.go.dev/crypto
23	https://pkg.go.dev/golang.org/x/crypto
24	https://docs.rs/ring/latest/ring/digest/static.SHA256.html
25	https://docs.rs/ring/latest/ring/index.html
26	https://docs.rs/scrypt/latest/scrypt/
27	https://github.com/RustCrypto/password-hashes/tree/master/yescrypt
28	https://docs.rs/argon2/latest/argon2/
29	https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography?view=net-9.0
30	https://github.com/bcgit/bc-csharp
31	https://github.com/petrsnm/yescrypt-net
32	https://www.bouncycastle.org/resources/nist-pqc-support-and-more-bouncy-castle-c-net-2-5-0/
33	https://developer.apple.com/documentation/cryptokit/sha256
34	https://github.com/krzyzanowskim/CryptoSwift
35	https://github.com/greymass/swift-scrypt
36	https://github.com/tmthecoder/Argon2Swift
37	https://developer.android.com/reference/java/security/MessageDigest
38	https://developer.android.com/reference/javax/crypto/SecretKeyFactory
39	https://docs.keyfactor.com/bouncycastle/latest/how-to-use-the-bouncy-castle-kotlin-api
40	https://pub.dev/documentation/cryptography/latest/cryptography/#classes
41	https://pub.dev/documentation/hashlib/latest/hashlib/Scrypt-class.html

Table 8: Reference links to the documentation, libraries and repositories cited in Table 6 for justifying the classification of each CHF-Language pair.