Evaluation of Password Hashing Schemes in Open Source Web Platforms

Christoforos Ntantogian, Stefanos Malliaros, Christos Xenakis

Department of Digital Systems, University of Piraeus, Piraeus, Greece

{dadoyan, stefmal, xenakis}@unipi.gr

Abstract: *Nowadays, the majority of web platforms in the Internet originate either from CMS to easily deploy websites or by web applications frameworks that allow developers to design and implement web applications. Considering the fact that CMS are intended to be plug and play solutions and their main aim is to allow even non-developers to deploy websites, we argue that the default hashing schemes are not modified when deployed in the Internet. Also, recent studies suggest that even developers do not use appropriate hash functions to protect passwords, since they may not have adequate security expertise. Therefore, the default settings of CMS and web applications frameworks play an important role in the security of password storage. This paper evaluates the default hashing schemes of popular CMS and web application frameworks. First, we formulate the cost time of password guessing attacks and next we investigate the default hashing schemes of popular CMS and web applications frameworks. We also apply our framework to perform a comparative analysis of the cost time between the various CMS and web application frameworks. Finally, considering that intensive hash functions consume computational resources, we analyze hashing schemes from a different perspective. That is, we investigate if it is feasible and under what conditions to perform slow rate denial of service attacks from concurrent login attempts. Through our study we have derived a set of critical observations. The conjecture is that that the security status of the hashing schemes calls for changes with new security recommendations and updates to the default security settings.*

Keywords: Passwords, CMS, Web application frameworks, Guessing attacks, Hashing schemes

1 Introduction

Several corporates [1] have become victims of security breaches, resulting in the disclosure of billions of passwords. One of the most significant data breaches during 2016 disclosed a database containing 1 billion users' authentication details [2], and was put on sale for 300.000 dollars [3], while one of the biggest data breaches during 2017 included 145.5 million users' details. Hackers take advantage of the computing power of graphics processing units (GPU) and specialized hardware to crack the users' passwords. Although the price of top tier graphics cards is relatively high (e.g., 2999$ for an NVIDIA TITAN V [4]), hackers can also rent cloud infrastructure including dedicated GPUs for a monthly or pay-as-you-go price (e.g. Google rents a GPU for maximum 2.55$ per hour [5]), making password guessing attacks easier and faster to perform.

To counteract the increasing efficiency of such attacks, key derivation functions such as PBKDF2 and BCRYPT use repeated iterations of the employed hash function to slow down the execution time of password hashing, and subsequently increase the effort required by an attacker to perform password guessing attacks. Moreover, memory hard functions (MHF), such as SCRYPT, use the physical memory as much as possible to significantly increase the costs required to crack a password. While there is a significant body of research that analyze the security of hash functions or propose new password cracking techniques, we have pinpointed

that the security of the default hashing schemes in Content Management Systems (CMS) and web application frameworks has been neglected. Nowadays, the majority of web platforms originate either from CMS to easily deploy websites or by web applications frameworks that allow developers to design and implement web applications. WordPress alone stands for 31.3% of all websites in the internet. Considering the fact that CMS are intended to be plug and play solutions and their main aim is to allow even non-developers to deploy websites, we argue that the default settings including the hash functions are not modified. Also, recent studies [6] suggest that even developers do not use appropriate hash functions to protect passwords, since they may not have adequate security expertise. Therefore, the default settings of CMS and web applications frameworks play an important role in the security of password storage.

This paper evaluates the security of the default hashing schemes of popular CMS and web application frameworks. To this end, we propose a simple framework that allows us to quantify the cost time for password cracking. The proposed framework takes into account all parameters that influence password guessing attacks and considers both brute force and dictionary attacks. To put our framework into a practical context and derive numerical results, we use as input the default hashing schemes of popular CMS and web applications frameworks. For this reason, first we identify and analyze the default hashing schemes of CMS and web applications frameworks. We have discovered that many CMS use outdated hash functions, arbitrary number of hash iterations, lack of password policies and salt. For example, the popular WordPress still uses MD5 with low number of hash iterations. Subsequently, we apply our cost analysis framework to perform a comparative analysis between the CMS and application frameworks which allows us to deduce a set of critical observations. Next, considering that intensive hash functions consume computational resources, we analyze hashing schemes from a different perspective. That is, we investigate if it is feasible and under what conditions to perform slow rate denial of service attacks from concurrent login attempts. Lastly, we propose security practices and alternative solutions to enhance the overall security of passwords. Overall, the contributions of this paper are the following:

- We propose a framework to estimate the cost time of brute force and dictionary password guessing attacks.
- We pinpoint the default hashing schemes of the most commonly used CMS and web application frameworks and we derive a set of critical observations.
- We apply our framework to the CMS and web application frameworks that allows us to perform a comparative analysis by quantifying the cost time of cracking a password.
- We investigate the feasibility of slow rate denial of service attacks based on intensive hash functions.
- Finally, we discuss and propose best practices and alternative solutions to improve the security of password storage.

The rest of this paper is organized as follows. Section 2 presents the required background knowledge and the related work while section 3 analyzes the various hashing schemes. Section 4 proposes a cost time analysis framework for brute force and dictionary attacks, while section 5 evaluates the default hashing schemes of CMS and web application frameworks. Section 0 performs a comparative analysis of the cost time of CMS and web applications frameworks, while section 7 examines the feasibility of denial of service attacks based on intensive hashing schemes. Section 8 discuss recommendations and possible solutions, and, lastly, section 9 contains the conclusions.

2 Background and Related Work

2.1 Password guessing attacks

Password guessing (also known as password cracking) is an attack in which an adversary attempts to guess the users' password. We distinguish two password guessing attacks categories: i) Online and ii) Offline. In online attacks, an attacker can try to login to a website by selecting frequently used passwords. After a number of unsuccessful attempts, the IP address or the username that the attacker is trying to login can be locked. On the other hand, in an offline attack, the scenario is that an attacker has in her possession a database of users' password hash values and she can attempt to crack each user's password offline by comparing the hashes of likely password guesses with the stolen hash value. Because the attacker can check each guess offline it is no longer possible to lockout the adversary after several incorrect guesses. In this paper we consider offline attacks.

Moreover, we can classify password guessing attacks to three categories: i) brute force ii) dictionary and iii) rainbow tables. In a brute force attack, the adversary tries every possible password combination considering two parameters; a) the password length; and b) the character set. On the other hand, in a dictionary attack, the adversary uses passwords from a list, which are likely to be used as passwords by users. There are four types of dictionary attacks: i) pure ii) Probabilistic Context Free Grammar (PCFG) based [7], iii) Markov model based [8] and iv) mangling rules [9]. In the pure dictionary, the attacker simply uses a set of predefined words as candidate passwords. In the second type, PCFG theories are used to construct a dictionary containing modified passwords with assigned probabilities. In the third type, Markov-based models are applied to create candidate passwords based on the probability distribution over sequences of characters. In the fourth type (i.e., mangling rules), the attacker creates password variations from a dictionary by applying various modifications rules, such as *"add the symbol ! at the end of the password"*. Finally, the third category of guessing attacks is rainbow tables, in which the attacker uses a precomputed list to reverse the hash value. In this paper, the term password guessing (or cracking), unless stated otherwise, refers specifically to brute force and dictionary attacks but not rainbow tables. Moreover, from the four types of dictionary attacks we exclude mangling rules as these are specific to each cracking tool.

2.2 Hardware platforms for password guessing

An attack scales linearly with invested resources, mainly cost of the equipment and energy consumption, and thus we have to take their influence into account. General purpose computing on GPUs can boost the computation performance, since the multiple GPU processing cores can be used in parallel for high-power calculations. Typically, a GPU consists of hundreds of computing cores grouped into computing clusters sharing the same memory bus. Due to this architecture, GPUs are specialized in Single Instruction, Multiple Data (SIMD) computations [10], which refer to the simultaneous execution of the same instruction on multiple processors with different input data for each processor (i.e., parallel computing). Consequently, GPUs can accelerate password guessing, since the same hashing scheme (i.e., the same instruction) can be executed simultaneously by hundreds of computing cores with different passwords as input. In [11], the authors measured the performance of the password guessing functions, where it was observed that the time required for password guessing decreased by 97% with GPU acceleration, compared with the time required using only CPU.

Apart from GPUs, special purpose hardware such as field-programmable gate arrays (FPGAs) and more recently application-specific integrated circuits (ASICs) have been utilized to yield even higher hashrate values. Generally speaking, equipment cost is in favor of the graphic cards, as GPUs are a consumer product that is sold in large quantities. Also, older versions usually

receive a discount, making them more cost-effective. Interestingly, FPGA vendors use a different strategy: with the release of a new product line, the price of the old family stays roughly unchanged, while the new version is offered with a small discount to make the consumers switch away from the abandoned hardware platform. In this paper, we will consider GPUs as the hardware platform of password guessing attacks.

2.3 CMS and web application frameworks

Nowadays, the majority of websites originate either from CMS or by web applications frameworks. CMS are intended to be plug and play solutions and their main aim is to allow non-developers to deploy websites. CMS play an important role in the Internet. According to [12], 52.3% of websites in the Internet are based on CMS. Table 1 shows statistics of CMS usage among all websites in the Internet and among all CMS [12]. In particular, first comes the popular WordPress with a whopping 31.3% usage among all websites in the Internet, while 59.8% usage among CMS. Second is Joomla with a 3.1 percentage usage among all websites in the internet, while Drupal is third with 2%.

CMS	Market share among all websites in the Internet	Market share among CMS
WordPress	31.3%	59.8%
Joomla	3.1%	6. 0%
Drupal	2.0%	3.9%
Magento	1.1%	2.1%
PrestaShop	0.7%	1.4%
TYPO3	0.7%	1.4%
OpenCart	0.4%	0.8%

Table 1: Popular CMS usage statistics

On the other hand, web application frameworks are utilized by developers and aim at supporting the development of rich web applications by providing a standard way to build and deploy web applications. For web application frameworks, we could not find a reliable source of statistics regarding their market share in the Internet. Considering that many frameworks share the same programming language, it is difficult to determine which specific framework a website uses. Therefore, we used statistics from GitHub to discover the most popular open source frameworks [13]. Table 2 shows the number of stars that each web application framework has which can be considered as a popularity metric among web developers. Laravel which uses PHP has the largest number of stars, which is 44.465. The second most popular framework, Ruby on Rails, is based on Ruby with 40.263 stars, while MeteorJS, based on Javascipt, has 40.068 stars. Note that from Table 2 ASP.NET is excluded, since GitHub is used only open-source projects.

Web application framework	Programming Language	# of stars on GitHub
Laravel	PHP	44.465
Ruby on Rails	Ruby	40.263
MeteorJS	Javascript	40.068
ExpressJS	Javascript	39.333
Flask	Python	37.515
Django	Python	35.230
SailsJS	Javascript	19.350

Table 2: Popular web application frameworks based on GitHub

2.4 Related work

The related work has studied extensively the area of password security from various scopes, including: i) password guessing attacks in leaked databases, and, ii) analysis of password complexity. Here we present only the most recent and relevant works. Regarding the first category, which is password guessing, the main metric which is used by the related work to estimate the attack efficiency is called effectiveness. In essence, effectiveness is the fraction of passwords that will be correctly cracked after an attack. The authors in [7] have used the PCFG technique, which uses grammar theories to construct a dictionary containing passwords in a decreasing probability order and succeeded in cracking 28% - 129% more passwords in comparison to John the Ripper (JtR) [14]. In [15], the authors analyzed the Rock you [16] database to identify regular expressions that were used to create candidate passwords. The numerical results showed that the proposed method cracks 14% - 239% more passwords in comparison with JtR.

Towards this direction, the work in [17] performs an analysis of Chinese web passwords by using the PCFG and Markov-based model, which create candidate passwords phonetically relevant to the words included in a dictionary. The authors succeeded in increasing password cracking efficiency by 48% and 4.7%, respectively, for each technique. In [18], the authors proposed a tool named OMEN, which was compared in password guessing with the PCFG and the Markov-based techniques. The recorded effectiveness was higher by 20% and 40% in comparison to PCFG and Markov-based techniques respectively. Moreover, [19] performed an empirical analysis on passwords and compared the effectiveness of dictionary password guessing attacks to this of the PCFG and Markov-based techniques. The PCFG method managed to crack 40-50% of the passwords, while 61.90% of passwords were cracked using the Markov-based methodology with 850 million guesses.

The second category of the related work is password complexity analysis. More specifically, the work in [20] performs a password analysis of the RockYou leaked database consisting of cleartext passwords. The results pinpointed that most of the passwords are not secure enough to withstand password guessing attacks. In fact, 30% of the users chose passwords whose length is equal or below six characters, and 60% of the users use the limited alpha-numeric set to form their passwords, while the most commonly used password was “123456”. Reports from the Keeper password manager [21] show that, even in 2016, the users’ passwords are still predictable, since the most common recorded passwords include “123456”, “qwerty” and “111111”. In [22], the authors performed interviews with several different groups (i.e., students, ICT specialists, etc.) regarding their password habits. They discovered that 50% of the respondents use less than 4 different passwords for all their services. Moreover, in all groups more than 50% of the respondents use passwords shorter than nine characters and most of the passwords still consisted of letters and characters.

3 Password hashing schemes

A hashing scheme takes as an input a plaintext password and transforms it into a hash value considering three parameters: i) hash function; ii) iterations; iii) salt. More specifically, the core parameter of a hashing scheme is the employed hash function, such as MD5. The iterations parameter is optional and specifies the number of consecutive executions of the employed hash function to compute the hash value. For example, if a hashing scheme uses the MD5 hash function and the number of iterations is 100, then it will conduct 100 consecutive executions of MD5 to compute the password hash. The number of iterations can be adjusted so that the computation of the hash value takes an arbitrarily large amount of computing time (also known as key stretching). In this way, iterations are used to slow down password guessing attacks. Regarding the last parameter, the salt is also optional and it is a random string which together with the password are the inputs to the hash function to produce the hash value. Using random

salts, rainbow tables become ineffective. That is, an attacker won't know in advance what the salt value is and therefore he/she cannot pre-compute a rainbow table.

There are numerous functions used for password hashing including: MD5 [23], SHA1 [24], SHA256 - SHA512 [25], PBKDF2 [26], BCRYPT [27], SCRYPT [28] and Argon2 [29]. A detailed analysis of these hash functions is omitted; instead we briefly mention the most important features relevant to the scope of this paper. The first four hash functions (i.e., MD5, SHA1, SHA256, SHA512) do not require the use of a salt by default. Thus, a separate function should be used to generate a s*alt* for the hashing scheme. On the other hand, the rest of the hash functions internally generate and use a random salt during hash calculation.

As we mentioned previously, the iterations parameter specifies the number of consecutive executions of the employed hash function, increasing the computation time to compute the hash value. For this reason, PBKDF2, BCRYPT, SCRYPT and Argon2 hash functions use iterations by default. More specifically, PBKDF2 is the simplest function, since it iterates the employed hash function, usually SHA256 or SHA512. On the other hand, BCRYPT, which is based on the blowfish encryption algorithm, uses iterations only in the Blowfish key setup function using the salt and password parameters as inputs. For PBKDF2 and BCRYPT, memory usage is not tunable separately (i.e., it is fixed for a given amount of CPU time). On the other hand, SCRYPT and Argon2 belong to a special category of hash functions named as memory hard functions (MHF), which are designed to use an arbitrary large and tunable amount of memory compared to PBKDF2 and BCRYPT making the size and the cost of a hardware implementation of these hash functions much more expensive, and therefore, limiting the amount of parallelism an attacker can use. Similar to BCRYPT, both SCRYPT and Argon2 use iterations in specific parts of the algorithm. SCRYPT was one of the first proposed MHF [28] and recently in 2016, the SCRYPT algorithm was published by IETF as a standard (RFC 7914) [30]. It is important to mention that for BCRYPT and SCRYPT, the literature uses the term cost factor [27], [28] instead of iterations (specifically for SCRYPT it is called CPU/Memory cost factor). In the rest of the paper we will explicitly use the term iterations instead of cost factor. Apart from iterations, SCRYPT and Argon2 include several parameters that can be used to adjust the memory requirements for hash value computation. The analysis of these parameters is out of scope of the paper, since we will specifically focus on the iterations parameter.

Regarding the exact value of iterations for the above hash functions, NIST guidelines recommend PBKDF2 with minimum 10.000 iterations [31], while the author of SCRYPT recommends 16384 iterations [28]. On the other hand, there is no official recommendation for BCRYPT and Argon2. We have only discovered that PHP programming language by default uses BCRYPT with 1024 iterations [32].

As mentioned in section 2.2, password guessing attacks greatly benefit from multiple processing cores, especially for hashing schemes that can be executed in parallel. MD5, SHA1, SHA256, SHA512 hash functions can be executed in parallel on multi-processor systems, fact that increases significantly the efficiency of password guessing attacks. Moreover, several weaknesses of PBKDF2 [33] allow efficient implementations with very little use of RAM, which makes brute-force attacks to PBKDF2 using FPGAs relatively cheap. Also, the work in [34] achieved a great optimization in running PBKDF2 on GPU hardware.

On the other hand, BCRYPT, due to its pseudorandom access to memory makes difficult to cache data into the GPU's internal memory [35]. Subsequently, BCRYPT implementations on GPUs use the external memory, thus spending more time transferring operands to and from the GPU. Thus, compared to PBKDF2, BCRYPT is less parallelizable and more resistant to password guessing attacks [28]. However, recent works such as [36] [37] have presented BCRYPT implementations that achieve a high level of parallelization in embedded hardware devices. Finally, MHF such as SCRYPT and Argon2 are specially designed to withstand against

hardware-equipped adversaries. MHF bound the memory amount and the memory bandwidth, limiting in this way the level of parallelism that an attacker can achieve. While a practical attack for SCRYPT has not been demonstrated yet, new MHF were proposed in the password hashing competition in 2014 [38] in which Argon2 was the winner.

4 Cost analysis of password guessing attacks

In this section we analyze a cost analysis framework for password guessing attacks. The rationale is to first compute the number of hashes, that will be performed throughout password guessing attacks, and secondly to estimate their *effectiveness* (i.e., percentage of successfully guessed passwords). By utilizing these two values, the cost of password guessing attacks is defined as the average number of hashes required to successfully crack a password hash. Lastly, the cost can be transformed into the average time required to crack a password hash. It is important to mention that the aim here is not to derive new mathematical models for password cracking, which has been already done in the previous works extensively (see section 0). Instead, our aim is to formulate a simple framework that will allow us to perform a security comparison and evaluation between the various CMS and application frameworks by quantifying the cost of password cracking.

4.1 Parameters

This section elaborates on the parameters of the proposed framework for the cost estimation of password guessing attacks. These parameters are as follows:

- **Iterations (*I*)**: The iterations parameter represents the number of consecutive executions of a hash function to compute the password hash. For example, a hashing scheme of 500 SHA1 iterations requires 500 consecutive executions of SHA1 to compute the hashing result. Note that this value is relevant only for iterations of MD5, SHA1, SHA256, SHA512 hash functions. On the other hand, PBKDF2, BCRYPT, SCRYPT and Argon2 that use iterations as an internal parameter, the parameter *I* is not considered (i.e., I=1).
- **Database passwords (*D*)**: This parameter indicates the number of password hashes in the database.
- **Salt (*S*):** This parameter indicates the number of salts in the database. We will assume that each password has a unique salt, therefore the number of database passwords *D* is equal to number of salts *S*. On the other hand, if the database does not use salt, then the parameter *S* is not considered (i.e., S=1).
- **Hashrate (*Hr*)**: It is the number of calculated hash values per second.
- **Password length (*pwd_length*):** This parameter is the length of the target passwords that an attacker desires to crack in a brute force attack. We also define as $pwd\_length_{\min}$ the minimum and pwd_length_{max}, the maximum password length that the attacker aims to crack.
- **Charset (*C*):** The *charset* is the second attacking parameter of brute force password guessing attacks. The value of *charset* depicts the number of unique characters of the different sets that are used for the composition of a password (see Table 3)
- **Attempts in a dictionary attack (*attempts*):** It is the number of candidate passwords that an attacker will attempt to crack the passwords. This parameter is relevant only for a dictionary attack.
- **Effectiveness (E_{BF} *or* E_{DC})**: The effectiveness of a password guessing attack is the percentage of password hashes in a database that will be cracked after the completion of the attack. The effectiveness of the brute force attack is denoted as E_{BF}, while for the dictionary attack is noted as E_{DC}.

Type of character set	Charset (*C*) value
Numeric	10
Lowercase	26
Uppercase	26
Loweralphanumeric or Upperalphanumeric	36
Mixedcase	52
Mixedalphanumeric	62
Special	94

Table 3: Charset value for different types of character sets

4.2 Brute force attack

In this section, we elaborate on the cost estimation of brute force password guessing attacks. The first step of the cost estimation is to compute the average number of hashes that will be performed during a brute force password guessing attack, defined as $hashes_{BF}$. To achieve this, we need to calculate the number of candidate passwords, by leveraging the charset and the *pwd_length* parameters. The usage of a unique salt per password affects the $hashes_{BF}$ value, since the guessing attempts performed during a brute force attack, will be a multiplication of all the candidate passwords by the total number of *salts*. Lastly, the $hashes_{BF}$ is affected by the usage of iterations, since a guessing attempt requires *iterations* consecutive hash executions.

Based on the above, it can be deduced that the $hashes_{BF}$ value can be estimated by using equation (1). The $hashes_{BF}$ value is analogous to both the iterations I and to the number of salts (i.e. S). In addition, $hashes_{BF}$ value is analogous to the sum of all candidate passwords (i.e. C^i), considering specific charset and password length values. That is,

$$\boldsymbol{Hashes_{BF} = a \cdot I \cdot S \cdot \sum_{i=pwd\_length_{min}}^{pwd\_length_{max}} C^i} \quad \textbf{(1)}$$

Note that the parameter a is a real number, where $a \in (0,1]$. The parameter a is defined as the *attack success factor* and it is related to the probability to successfully crack all hashed passwords at the end of the attack. In the worst case scenario for the attacker, the value of a is equal to 1. In this case, the attack will cover all the candidate passwords. To better understand the role of the parameter a, we consider the following example. Assume a brute force attack in which the attacker aims to crack numeric passwords (i.e., C=10 from Table 3) of minimum length 4 and maximum length 5 (i.e., pwd_length_{min} = 4, pwd_length_{max} = 5), for a hashing scheme that uses 100 iterations (I=100). The number of the hashed passwords is D=100. This means that the salt S is also equal to 100 (i.e., one salt per password). All the candidate 4-character numeric passwords are 10^4, while the 5-character are 10^5, summing to a total number of 1.1 10^5 passwords. If we assume the worst case scenario for the attacker (i.e., a=1), then by multiplying the number of candidate passwords with the iterations and the number of salts, the value of $hashes_{BF}$ will be $1.1 \cdot 10^9$. This means that the attacker for each password (with its related salt) will cover all candidate passwords. On the other hand, in the average case we have $a = 1/2$ and in this case the attacker will cover half of candidate passwords (i.e., $Hashes_{BF} = \frac{1.1 \cdot 10^9}{2}$).

The second step of this analysis is to estimate the number of target password hashes that will be cracked by a brute force attack, defined as $cracked\_pass_{BF}$. This can be achieved by leveraging the effectiveness parameter E_{BF}, which defines the percentage of password hashes that will be successfully cracked by the attack. Therefore, using E_{BF}, we can calculate the

cracked_pass$_{BF}$ by multiplying the E_{BF} with the number of password hashes in the database *D*, as shown in equation (2).

$$\boldsymbol{cracked\_pass_{BF} = D \cdot E_{BF}} \quad \textbf{(2)}$$

Having calculated the *hashes*$_{BF}$ and the *cracked_pass*$_{BF}$, we can calculate the cost of password guessing for the brute force attack, (defined as *cost*$_{BF}$). The cost *cost*$_{BF}$ represents the average number of hashes that will be performed during the attack to crack a password hash in the database. To calculate *cost*$_{BF}$ we use the following equation.

$$\boldsymbol{cost_{BF} = \frac{hashes_{BF}}{cracked\_pass_{BF}}}$$

By replacing the *hashes*$_{BF}$ with equation (1) and *cracked_pass*$_{BF}$ with equation (2), the final form of *cost*$_{BF}$ can be derived as follows:

$$\boldsymbol{cost_{BF} = \frac{a \cdot I \cdot S}{D \cdot E_{BF}} \cdot \sum_{i=pwd\_length_{min}}^{pwd\_length_{max}} C^i} \quad \textbf{(3)}$$

Lastly, the *cost*$_{BF}$ can be translated into the average time required to crack a password hash in the database *D*, (defined as *cost_time*$_{BF}$) using the hashrate (i.e. *Hr*) parameter, as shown in equation (4).

$$\boldsymbol{cost\_time_{BF} = \frac{cost_{BF}}{Hr}} \quad \textbf{(4)}$$

4.3 Dictionary Attack

In this section, we elaborate on the cost estimation of dictionary password guessing attacks. The first step of the cost estimation is to compute the number of hashes that will be performed during an attack, defined as *hashes*$_{DC}$. The *hashes*$_{DC}$ value can be estimated by multiplying the iterations *I* with the salt *S* and with the number of guessing attempts (i.e., *attempts*). Thus, *hashes*$_{DC}$ can be estimated as follows:

$$\boldsymbol{hashes_{DC} = a \cdot I \cdot S \cdot attempts} \quad \textbf{(5)}$$

As in the brute force attack, the parameter a is the attack success factor. The next step for the cost estimation is to compute the number of password hashes that will be cracked after the completion of a dictionary password guessing attack, defined as *cracked_pass*$_{DC}$. The value of *cracked_pass*$_{DC}$ relies on the effectiveness E_{DC} of the dictionary attacks. Note that the E_{DC} value relies on the actual method of dictionary attack (e.g., pure, PCFG or Markov model). Using E_{DC}, the estimated number of the cracked passwords can be computed as follows:

$$\boldsymbol{cracked\_pass_{DC} = D \cdot E_{DC}} \quad \textbf{(6)}$$

Having calculated the *hashes*$_{DC}$, and the *cracked_pass*$_{DC}$, the last step is to estimate the average hashes that will be performed in the database *D* until a successful password crack, defined as *cost*$_{DC}$. To achieve this, we divide *hashes*$_{DC}$ by *cracked_pass*$_{DC}$.

$$\boldsymbol{cost_{DC} = \frac{hashes_{DC}}{cracked_{pass_{DC}}}}$$

Next, we can use equations (5) and (6), to derive the final form of *cost*$_{DC}$.

$$\boldsymbol{cost_{DC} = \frac{a \cdot I \cdot S \cdot attempts}{D \cdot E_{DC}}} \quad \textbf{(7)}$$

Finally, to convert *cost*$_{DC}$ into the average time required until a successful password crack in the database *D*, *cost_time*$_{DC}$, we need to divide *cost*$_{DC}$ by the hashrate (i.e. *Hr*), as shown in equation (8).

$$cost\_time_{DC} = \frac{cost_{DC}}{Hr} \quad (8)$$

5 Evaluation of default hashing schemes in CMS and web applications frameworks

This section evaluates the default hashing schemes used by CMS and web application frameworks based on the following parameters: i) hash function; ii) *iterations*; iii) usage of *salt*, and iv) minimum acceptable *pwd_length*. In total, we have examined 49 commonly used CMS and 47 popular web application frameworks. Table 4 shows the considered CMS classified into 7 categories: i) 13 CMS are included in the generic category, which represents CMS that allow the development of websites with various functionalities that focus on the content (e.g. blog, news web site), ii) 9 for forums, iii) 5 for ecommerce, iv) 7 for Enterprise Resource Planning (ERP) and Customer Relationship Management (CRM), v) 2 for coding and bug tracking, vi) 2 for project management, and viii) 11 are classified as "Other", which do not belong to any of the above categories.

Based on the results of Table 4 which depicts the default hashing schemes of the investigated CMS, we can observe that 26.53% of the CMS including osCommerce, SuiteCRM, WordPress, X3cms, SugarCRM, CMS Made simple, Mantisbt, Simple Machines, miniBB, Phorum, MyBB, Observium, and Composr use the outdated hash function MD5. MD5 is highly parallelizable and we will analyze in section 6.1.1, it is the fastest among all hash functions that can be executed in GPUs. Regarding the remaining hash functions of the CMS, GetSimple CMS, Redmine, Collabtive, PunBB, Pligg, and Omeka (i.e. 12.24%) use the SHA1 hash function, which similar to MD5 is highly parallelizable on GPUs. Drupal, EspoCRM, PhreeBooks, Odoo, ImpressCMS, Magento, Bugzilla, TYPO3 CMS, Mediawiki, and PhpList (i.e. 20.41%) use either SHA256/SHA512 or PBKDF2. These hash functions are also parallelizable, thus increasing the effectiveness of password guessing attacks. Lastly, Joomla, Zurmo, OrangeHRM, SilverStripe, Elgg, XOOPS, e107, NodeBB, Concrete5, phpBB, Vanilla Forums, Ushahidi, Lime Survey, Mahara, Mibew, vBulletin, OpenCart, PrestaShop, and Moodle (i.e. 40.82%) use the BCRYPT hash function. As we mentioned in section 3, BCRYPT is more secure than the rest of the hashing schemes, since it more difficult to be parallelized in GPU hardware. Based on the above we can conclude to the following observation:

Observation 1: *A whopping number (i.e., 59.18%) of CMS use default hashing schemes that can be highly parallelized with GPU hardware, making password guessing attacks easier. Indicatively, the popular CMS WordPress uses by default MD5. On the other hand, 40.82% of the CMS use BCRYPT by default including Joomla.*

Another observation which is related to the usage of the hashing schemes is the following:

Observation 2: *No CMS has adopted SCRYPT, Argon2 or any other MHF yet.*

Observation 2 may come as no surprise if we consider that the PHP programming language that all the CMS are based on, has no official SCRYPT implementation. This means that in case an administrator of a CMS wants to use SCRYPT, he/she should rely on a third party or custom implementation of SCYPT. However, using non-official implementations is considered an insecure practice, as they may include backdoors [39], [40] or insecure code [41]. On the other hand, Argon2 was included recently (late 2017) in PHP v7.2 and compared to SCRYPT it can be more easily adopted in a CMS. However, Argon2 is a relatively new hash function and the audits are too scarce to draw safe conclusions about its security properties. Finally, a common

CMS	Category	Hash function	Iterations	Salt	Min pwd length	CMS	Category	Hash function	Iterations	Salt	Min pwd length
Drupal 8.4.4	Generic	SHA512	65536	✓	1	OsCommerce2.3.4.1	Ecommerce	MD5	1	✓	5
Joomla 3.8.3	Generic	BCRYPT	1024	✓	4	Zen Cart 1.5.5	Ecommerce	BCRYPT	1024	✓	7
WordPress 4.9.1	Generic	MD5	8192	✓	1	SuiteCRM 7.9.9	ERP/CRM	MD5	1000	✓	1
X3cms 0.5.3	Generic	MD5	1	✗	5	Zurmo 3.2.3	ERP/CRM	BCRYPT	4096	✓	5
ImpressCMS 1.3.10	Generic	SHA512	5000	✓	5	OrangeHRM 4.0	ERP/CRM	BCRYPT	4096	✓	4
GetSimple CMS 3.3.13	Generic	SHA1	1	✗	1	SugarCRM 6.5.25	ERP/CRM	MD5	1000	✓	1
CMS Made simple	Generic	MD5	1	✓	1	EspoCRM 5.0.2	ERP/CRM	SHA512	1	✓	1
SilverStripe 4.0.1	Generic	BCRYPT	1024	✓	1	PhreeBooks 9	ERP/CRM	SHA256	1	✓	5
Elgg 2.3.5	Generic	BCRYPT	1024	✓	6	Odoo 11	ERP/CRM	$PBKDF2_{SHA512}$	12000	✓	1
XOOPS 2.5.9	Generic	BCRYPT	1024	✓	5	Mantisbt 2.10.0	Coding	MD5	1	✗	1
e107 2.1.7	Generic	BCRYPT	1024	✓	8	Bugzilla 5.1.1	Coding	SHA256	1	✓	8
TYPO3 v9	Generic	$PBKDF2_{SHA512}$	25000	✓	8	Redmine 3.4.4	Proj. Mgmt	SHA1	2	✓	8
Concrete5 8.3.1	Generic	BCRYPT	4096	✓	5	Collabtive 3.1	Proj. Mgmt	SHA1	1	✗	1
PhpBB 3.2.2	Forum	BCRYPT	1024	✓	6	Ushahidi 3	Other	BCRYPT	4096	✓	7
Vanilla Forums 2.6	Forum	BCRYPT	1024	✓	6	Pligg 1.2.2	Other	SHA1	1	✓	5
Simple Machines 2.0.15	Forum	MD5	1	✓	6	Observium 0.17.11	Other	MD5	1000	✓	1
MiniBB 3.2.2	Forum	MD5	1	✗	5	Lime Survey 2	Other	BCRYPT	1024	✓	1
Phorum 5.2.23	Forum	MD5	1	✗	1	MediaWiki 1.30.0	Other	$PBKDF2_{SHA512}$	30000	✓	1
MyBB 1.8.12	Forum	MD5	1	✓	6	Omeka 2.5	Other	SHA1	1	✓	6
PunBB 1.4.4	Forum	SHA1	1	✓	4	phpList 4	Other	SHA256	1	✗	8
vBulletin 5.3.4	Forum	BCRYPT	1024	✓	1	Mahara 17.04	Other	BCRYPT	4096	✓	6
NodeBB	Forum	BCRYPT	4096	✓	6	Mibew 3.1.3	Other	BCRYPT	256	✓	1
OpenCart 3.0.2.0	Ecommerce	BCRYPT	1024	✓	4	Composr 10	Other	MD5	1	✓	1
PrestaShop 1.7	Ecommerce	BCRYPT	1024	✓	5	Moodle 3.4	Other	BCRYPT	1024	✓	8
Magento 2.2	Ecommerce	SHA256	1	✓	7						

Table 4: The default hashing scheme parameters of the investigated open source CMS

reason that hinders the adoption of both SCRYPT and Argon2 is related to the fact that the transition to a new hashing scheme of an already deployed website can lead to downtimes or it may require once again the registration of its users with a new (or the same) password. Therefore, for backwards compatibility reasons website administrators avoid to modify hashing schemes and choose to remain with legacy hash functions. A case in point is the CMS named Phorum; it still uses the MD5 as the default hashing scheme (see Table 4), despite the fact that there is a request in the official development repository of Phorum to change MD5 to a stronger hash function [42]. After a discussion between users and the development team (see [42]), the main developer opposes to this change, because the developers of Phorum CMS are considered how existing installations are going to update to the new hash function. Thus, they decide not to proceed with any modification to the hash function leaving MD5 as the main hash function. Another similar discussion takes place for Magento CMS [43], which is an e-commerce platform and still uses SHA256.

Regarding the usage of *salt*, the most important finding is that 14,29% of the targeted CMS, and specifically X3cms, GetSimple CMS, miniBB, Phorum, MantisBT, Collabtive, and phpList do not use *salt* in their hashing scheme (see Table 4), which renders password hashes vulnerable to rainbow table attacks. The fact that salt is missing in these CMS implies that users with the same plaintext passwords will also share the same password hash. Another important finding is that 36.73% of the tested CMS do not use *iterations* in their password hashing scheme (i.e., the iterations value is 1). Also, the rest of the CMS that use iterations use an arbitrary number of iterations. For instance, for BCRYPT we observe that there are CMS that use 256, 1024, or 4096 iterations, while for PBKDF2 we observe 10000, 12000, or 30000. This variations stems from the fact that BCRYPT does not have official recommendations for its iterations, while NIST proposes a minimum of 10.000 iterations for PBKDF2. Based on the above, we can conclude to the following observation:

Observation 3: *Password hashes created by 14.29% of the CMS are vulnerable to guessing attacks based on rainbow tables, since the relevant CMS do not use salt in their hashing scheme. Also, 36.73% of the CMS do not use iterations, which makes them even more vulnerable to password guessing attacks. On the other hand, the rest of the CMS that use iterations, select the number of iterations inconsistently and arbitrarily.*

The last parameter to be analyzed is the minimum acceptable password length. Although this parameter does not affect the execution time of a hashing scheme, password hashes created from small passwords are more likely to be cracked. From the analysis of Table 4 it is observed that only 12.24% of the CMS (i.e., e107, Typo3 CMS, Bugzilla, Redmine, Phplist, and Moodle) enforce passwords of 8 characters length or greater. On the other hand, 6.12% require passwords with a minimum length of 7 characters, 14.29% of 6 characters, 20.41% of 5 characters and 8.16% of 4 characters. However, the most important remark is that 38.78% (i.e. Drupal, SuiteCRM, WordPress, SugarCRM, EspoCRM, GetSimple CMS, CMS Made simple, Odoo, Mantisbt, Collabtive, Vanilla Forums, Observium, Lime Survey, MediaWiki, Phorum, vBulletin, Mibew, and Composr) of the CMS do not check the password length during the registration process, since we were able to create single character passwords. Based on the above, we can conclude to the following observation:

Observation 4: *38.78% of the CMS do not enforce minimum password length policy, which may result in users selecting weak passwords. Notably, WordPress and Drupal belong to this category of CMS that allow a single character password. This observation, alongside with the fact that many CMS use parallelizable hash functions makes password cracking even more effective.*

Driven by the above observations, we can conclude that the majority of CMS offer weak hashing schemes in the default settings. A prime example is Phorum; it uses MD5 without

iterations and salt, while it allows even 1-character length passwords (see Table 4). Of note, the majority of the considered CMS allow modifications to the default settings. For instance, there is a plugin for WordPress that allows to easily change the default MD5 to BCRYPT for password hashing. However, CMS are characterized as "plug and play" solutions. In particular, their main goal is to allow even non-developers to easily deploy websites. This fact makes it less probable that CMS administrators will ever try to modify the default configurations. What is more, this argument is also strengthened by the fact that in general individuals tend to remain at the default assignment (also known as default effect [44]). Based on the above, a more generic observation can be extracted as follows:

Observation 5: *CMS follow an opt-in policy for security configurations. That is, by default they do not provide the most secure hashing schemes, but they allow the modification to more secure schemes. However, considering that CMS administrators may not be developers and do not have the appropriate security expertise, we argue that most CMS are deployed in the Internet with the default security settings including the hashing scheme.*

The second part of this section examines the default hashing schemes of the most commonly used web application frameworks. As we mentioned in section 2.3, a key difference between CMS and web application frameworks is that the latter require programming knowledge and they are utilized by web developers, while the former (i.e., CMS) does not require coding knowledge, since it is based on installable modules. Table 5 shows the investigated web application frameworks classified into 5 categories, based on the programming language for web application development. More specifically, we investigated i) 10 frameworks which rely on PHP, ii) 14 that are based on Python, iii) 11 that use Ruby on Rails, and iv) 11 based on Javascript. ASP.NET is the last framework we explored, and we categorized it as "Other", since it supports development in several programming languages. The default hashing schemes of the investigated web application frameworks are depicted in Table 5. An important observation that can be derived is that 48.94% of the web application frameworks do not offer a default password hashing scheme, which might lead to improper password hashing. Moreover, the Kohana PHP framework uses the same *salt* value for all stored passwords, thus they are vulnerable to rainbow table attacks. Another significant finding is that Kohana, Django, CherryPy, Bottle, ExpressJS, MeanJS, MernJS, nodeJS, AllcountJS, Cuba, and ASP.NET (i.e. 23.40%) use parallelizable hash functions (i.e., MD5, SHA1, SHA256, SHA512 and PBKDF2), while Kohana, CherryPy, Bottle, AllcountJS, Cuba, and ASP.NET (i.e. 12.77%) use only 1 *iteration* of the employed hash function. On the other hand, Laravel 5.4, Codeigniter 3.1.4, CakePHP 3.3, Zend framework3, Yii 2, Phalcon 3.0.4, Aura PHP, Lithium, MeteorJS, SailsJS, FathersJS, Derby, and Ruby on Rails, which stand for 27.66% use the BCRYPT hash function by default. Based on the above we can conclude to the following observation:

Observation 6: *23.40% of the web application frameworks opt for weak (i.e., parallelizable) hash functions, while 12.77% of them do not use iterations. What is more, only 27.66% use the BCRYPT hash function by default. Similar to CMS and observation 2, SCRYPT and Argon2 are absent from the default settings.*

Moreover, from Table 5, we can notice that:

Observation 7: *48.94% of the investigated web application frameworks do not offer a default password hashing scheme, which might lead to the selection of a weak password hashing scheme in web applications.*

The underlying assumption of observation 7 lies to the fact that developers are expected to have the knowledge of selecting appropriate hash functions and configure securely the hashing scheme of the websites they develop using salts. In a recent work [6], web developers were given the task to store passwords for authentication in a website. Among the many key insights of this work, the most important ones were: i) many developers stored the passwords in

PHP Frameworks	Hash function	Iterations	Salt
Kohana 3.3	SHA256	1	✓ (Constant)
Symfony 3.2	No default		
Laravel 5.4	BCRYPT	1024	✓
Codeigniter 3.1.4	BCRYPT	1024	✓
CakePHP 3.3	BCRYPT	1024	✓
Zend framework3	BCRYPT	16384	✓
Yii 2	BCRYPT	8192	✓
Phalcon 3.0.4	BCRYPT	256	✓
Aura PHP	BCRYPT	1024	✓
Lithium	BCRYPT	1024	✓
Python Frameworks	**Hash function**	**Iterations**	**Salt**
Django	$PBKDF2_{SHA256}$	30000	✓
CherryPy	MD5	1	✓
Flask	$PBKDF2_{SHA256}$	50000	✓
Bottle	No default		
Pyramid	SHA512	1	✗
Klein	No default		
Web2py	SHA512	1000	✗
Objectweb	No default		
Pecan	No default		
Tornado	No default		
Grok	No default		
Zope	No default		
Turbogears	No default		
Quixote	No default		

JavaScript	Hash function	Iterations	Salt
MeteorJS	BCRYPT	1024	✓
ExpressJS	$PBKDF2_{SHA512}$	10000	✓
SailsJS	BCRYPT	1024	✓
FathersJS	BCRYPT	1024	✓
Derby	BCRYPT	1024	✓
Wakanda	No default		
MeanJS	$PBKDF2_{SHA512}$	10000	✓
MernJS	$PBKDF2_{SHA512}$	10000	✓
nodeJS	$PBKDF2_{SHA512}$	10000	✓
AllcountJS	SHA1	1	✓
AngularJS	No default		
Ruby Frameworks	**Hash function**	**Iterations**	**Salt**
Ruby on Rails	BCRYPT	1024	✓
Padrino	No default		
Nyny	No default		
Grape	No default		
Nancy	No default		
Ramaze	No default		
Cuba	SHA1	1	✓
Camping	No default		
Scorched	No default		
Celluloid	No default		
Sinatra	No default		
Other Frameworks	**Hash function**	**Iterations**	**Salt**
ASP.NET	SHA256	1	✓

Table 5: The default hashing scheme parameters of the investigated web application frameworks

plaintext; ii) most of the developers focused on the functionality and only added security as an afterthought; iii) even participants who attempted to store passwords security often did it insecurely, because they used outdated methods (e.g., they used MD5 without even iterations) as security is a fast moving field; iv) different standards and security recommendations made it difficult for developers to decide what is the right course of actions. Therefore, all the above observations imply that there is a lack of adequate security knowledge even by developers, and simply assuming that they will select a secure password storage scheme is a dangerous misconception. Hence, it would be beneficial for web applications frameworks to offer secure default hashing schemes.

6 Cost analysis comparison of CMS and web applications frameworks

Based on the cost analysis framework that we derived in section 4, in this section we perform a comparison of the various CMS and web frameworks to evaluate the security of their default hashing scheme. In order to be able to perform this numerical comparison, first we have to derive the input values for our cost analysis framework. In particular, we have to derive values for: i) hashrates for various hash functions and iterations ii) effectiveness of dictionary attacks, and, iii) effectiveness of brute force attacks.

6.1 Input values

6.1.1 Hashrates

First, we derive hashrate values using a popular GPU-based password cracking tool named Hashcat [45]. Due to its' popularity, there are numerous benchmarks available on the Internet that calculate the hashrate of various GPU models. However, due to the fact that we were not able to find up to date benchmarks (i.e., the most recent ones were of 2014) we opted for our own benchmarks. To this end, we derived hashrate values (see Table 6) of various hash functions and iterations using the GeForce GTX 1070 [46], which was NVIDIA's second-best GPU model of 2016. As expected the hash functions MD5, SHA1, SHA256 and SHA512 exhibit high performance in the sense that GPUs can compute several hashes per second. PBKDF2 slows downs the computations due to the iterations used. Regarding BCRYPT and SCRYPT, we observe that BCRYPT has the slowest performance for number of iterations up to 16384 iterations, but for higher values, SCRYPT is slower than BCRYPT.

Along with GPU based hashrates, it is imperative to derive the runtime of a hash value calculation in a typical Web Server machine. The reason for this calculation is that the number of iterations should not be set too high; otherwise the calculation of a hash value can be significantly delayed, disrupting the normal operation of the website. That is, authentication delays (due to the multiple iterations for a hash calculation) can frustrate users that are trying to login, especially if they have to provide multiple times their password, because they provided an erroneous input. As mentioned in [47], [48], authentication delays higher than 1 second are not acceptable by many internet users. As a side note, for an offline environment (i.e., disk encryption), higher numbers of iterations can be used (e.g., for key generation from low entropy passwords). To this end, we have used a typical server setup, an Intel Xeon E5-2640 v2 CPU with 4 GB RAM to estimate the runtime of the hash functions for various iterations (see Table 6). We observe that in almost all considered iterations values, the runtime of the hash functions does not exceed the upper limit of one second, except for BCRYPT for 32678 and 65536 iterations, which the runtime is 2.72 sec and 5.45 seconds respectively.

Hash function (iterations)	Hashrate (H/s) (NVIDIA GTX1070)	Runtime (sec) (Intel Xeon E5-2640 v2)
MD5 (1)	21,359,700,000.00	$1.06 \cdot 10^{-6}$
SHA1 (1)	7,043,888,888.00	$1.37 \cdot 10^{-6}$
SHA256 (1)	2,536,500,000.00	$1.75 \cdot 10^{-6}$
SHA512 (1)	844,100,000.00	$1.95 \cdot 10^{-6}$
BCRYPT (1024)	358.00	$8.68 \cdot 10^{-6}$
BCRYPT (8192)	44.75	$6.85 \cdot 10^{-5}$
BCRYPT (16384)	22.00	$6.8 \cdot 10^{-1}$
BCRYPT (32768)	11.00	2.72
BCRYPT (65536)	5.00	5.45
$PBKDF2_{SHA256}$ (8192)	121,375.00	$1.09 \cdot 10^{-2}$
$PBKDF2_{SHA256}$ (16384)	60,574.00	$3.92 \cdot 10^{-2}$
$PBKDF2_{SHA256}$ (32768)	30,271.50	$7.67 \cdot 10^{-2}$
$PBKDF2_{SHA256}$ (65536)	15243.50	$1.57 \cdot 10^{-1}$
$PBKDF2_{SHA256}$ (131072)	7,587.00	$3.04\ 10^{-1}$
$PBKDF2_{SHA256}$ (262144)	3,797.00	$6.16 \cdot 10^{-1}$
$PBKDF2_{SHA512}$ (8192)	43,631.00	$2.61 \cdot 10^{-2}$
$PBKDF2_{SHA512}$ (16384)	22,174.00	$5.23 \cdot 10^{-2}$
$PBKDF2_{SHA512}$ (32768)	10,895.25	$1.03 \cdot 10^{-1}$
$PBKDF2_{SHA512}$ (65536)	5487.00	$2.06 \cdot 10^{-1}$
$PBKDF2_{SHA512}$ (131072)	2,752.00	$4.12 \cdot 10^{-1}$
$PBKDF2_{SHA512}$ (262144)	1,388.00	$8.22 \cdot 10^{-1}$
SCRYPT (8192)	122.00	$2.75 \cdot 10^{-2}$
SCRYPT (16384)	34.00	$5.24 \cdot 10^{-2}$
SCRYPT (32768)	9.00	$1.06 \cdot 10^{-1}$
SCRYPT (65536)	2.00	$2.16 \cdot 10^{-1}$
SCRYPT (131072)	0.3	$4.35 \cdot 10^{-1}$
SCRYPT (262144)	0.012	$8.71 \cdot 10^{-1}$

Table 6: Hashrates and runtime values

6.1.2 Effectiveness

6.1.2.1 Dictionary

In this section, we analyze the effectiveness E_{DC} (see section 4.1) for three types of dictionary attacks: i) pure ii) Markov model and iii) PCFG. These values are obtained from the related work. For pure dictionary attacks, we use the E_{DC} and the *attempts* parameter values from [19] (see Table 7). The authors of this work used dictionaries with English, Italian and Finish lowercase words and executed pure dictionary attacks against two databases DB1 and DB2 respectively, recording effectiveness E_{DC} values 24.79% and 26.02% respectively. Note that the DB1 included hashed passwords leaked from an Italian messaging server, while DB2 consisted of hashed passwords from Finnish speaking forums.

Dictionary	*attempts*	E_{DC} DB1	E_{DC} DB2
English, Italian and Finnish words	$1.45 \cdot 10^3$	24.79%	26.02%

Table 7: Effectiveness values for pure dictionary password guessing attacks (values were taken from [19])

Moreover, we have obtained the E_{DC} values based on Markov model and PCFG as derived from [19] (see Table 8). The E_{DC} for the PCFG model ranges from 41.50% for 1.45 million guessing attempts to 49.36% for *145 million* guessing attempts. On the other hand, the Markov model is more efficient, since its E_{DC} values are greater than the ones of PCFG. Particularly, by leveraging the Markov model, 53.49% of the passwords can be cracked with 149 million attempts, while this value can be increased to 99.70% for 10^{40} guessing attempts.

Model	*attempts*	E_{DC}
PCFG	$1.45 \cdot 10^6$	41.50%
PCFG	$41 \cdot 10^6$	46.33%
PCFG	$145 \cdot 10^6$	49.36%
Markov	$\sim 149 \cdot 10^6$	53.49%
Markov	$\sim 156 \cdot 10^6$	54.58%
Markov	$\sim 850 \cdot 10^6$	61.90%
Markov	$\sim 7 \cdot 10^{16}$	91.44%
Markov	$\sim 10^{40}$	99.70%

Table 8: Effectiveness values for dictionary password guessing using PCFG or Markov models (values were taken from [19])

6.1.2.2 Brute force

To compute the effectiveness of a brute force attack E_{BF}, we define the parameter P_{pwd_length} as the percentage of passwords that have a specific length and the parameter P_{C,pwd_length}, as the percentage of passwords to have a specific length and charset *C*. For instance, for *pwd_length*=8, then P_{pwd_length} represents the percentages of 8-character passwords, while for charset *C*=10 (see Table 3) and *pwd_length*=4, then P_{C,pwd_length} is the percentage of numerical passwords with 4 digit numbers. Recall also from section 4.1, that $pwd\_length_{\min}$ and pwd_length_{max}, is the minimum and maximum password length respectively that the attacker aims to crack. Based on the above, the E_{BF} value can be estimated as shown in equation (9).

$$E_{BF} = \sum_{pwd\_length_{min}}^{pwd\_legnth_{max}} P_{pwd\_length} \cdot P_{C,pwd\_lentgh} \quad (9)$$

To the best of our knowledge there is no work that has calculated the P_{pwd_length} and the P_{C,pwd_length} values. To this end, we perform an empirical analysis of passwords, in order to derive numerical values for P_{pwd_length} and P_{C,pwd_length}. More specifically, we have gathered a large collection of leaked password datasets from various online services across multiple years (from 2006 to 2017). The total number of collected passwords is 254.38 million passwords from 12 datasets. Note that these datasets are public and can be found in the Internet in various blogs and forums. It is also important to mention that we have collected leaked datasets that include only plaintext passwords. This is a key factor to avoid biasing results, since in this way we guarantee that all passwords are included in our statistical analysis. On the contrary, if we had used datasets that include cracked passwords, then we would have performed a statistical analysis only with passwords that have been guessed biasing the results. We verified that the considered databases are composed of plaintext passwords using a two-step procedure: i) by checking that the length of the passwords in the datasets do not match the length of a hash value (e.g., an MD5 hash has always a fixed output of 16 bytes), and ii) by performing a cross check

with a historical record of leaked passwords available as a public service [49]. Considering that the processed usernames and passwords are in plaintext form, we do not reference their source, since many of these accounts may be still active.

In Table 9, we classify the breached websites into various categories (9 in total) based on their content or service they provide. We observe that the associated user accounts of these websites are diverse in the sense that they are created from non-technical users (e.g. Mate1 was an online dating platform) to web developers (e.g. 000webhost is a web hosting platform for PHP/MySQL websites). Moreover, the breached websites offer their services globally, except for Auction-warehouse which explicitly requires their users to be US citizens. Therefore, we believe that the collected datasets represent a diverse and generic set of passwords.

Dataset #	Website	Category	Number of Passwords
1	000webhost	Web hosting	15.311.565
2	1394store	e-shop	20.649
3	Auction-warehouse	Auctions	26.616
4	Clixsense	Advertisemts	2.222.542
5	Mail.ru	email	4.664.479
6	Mate1	Social	27.403.959
7	Neopets	Gaming	68.743.269
8	Rockyou	Social	32.625.471
9	Tuscl	Adult	38.599
10	VKontakte	Social	100.544.934
11	Yahoo voices	Publishing	453.837
12	Youporn	Adult	2.325.492

Table 9: Categories and number of leaked passwords

The numerical values of the password analysis are shown in Table 10. Note that the presented values are averages of the password length and character set distributions from each one of the considered databases. For the character set distributions we classify the passwords based on the following categorization: i) ***numeric*****:** only numbers (e.g., 1234567890); ii) ***lowercase*****:** only lowercase Latin alphabet characters (e.g. password); iii) ***uppercase*****:** only uppercase Latin alphabet characters (e.g., PASSWORD); iv) ***mixedcase*****:** *uppercase* + *lowercase* (e.g., PassworD); v) ***loweralphanumeric*****:** *lowercase* + *numeric* (e.g., passw0rd); vi) ***upperalphanumeric*****:** *uppercase* + *numeric* (e.g., PASSW0RD); vii) ***mixedalphanumeric*****:** *mixedcase* + *numeric* (e.g., Passw0rD); and viii) ***special*****:** passwords that contains at least one special character (e.g., P@ssw0rD).

Table 10 can be used to derive the P_{pwd_length} and P_{C,pwd_length} values and consequently the effectiveness E_{BF} of brute force attacks. To exemplify, consider an attack targeting 7 to 8-character lowercase passwords (i.e., *pwd_length*=8 and *C*=26). In this case, P_{pwd_length} equals to 20.68%, and P_{C,pwd_length} equals to 30.36%, while pwd_length_{min}=7 and pwd_length_{max}=8. Thus, using equation (9), the effectiveness for a brute force attack E_{BF} is equal to 12.16%.

6.2 Comparative analysis

Here we use our cost analysis model that we presented in section 4 to perform a comparative analysis of the cost time between different CMS and web application frameworks. To derive numerical results for the cost time we consider the values from section 6.1 for the hashrates as well as for brute force and dictionary effectiveness. We also consider the worst case scenario for the attacker, which means that the attack success factor a is equal to 1 (see section 4.2). Table 11 summarizes the numerical results. The comparison is performed using five (5) different groups. Group 1 compares the cost time for a brute force attack (i.e., $cost\_time_{BF}$)

Password length	Password Length Distribution	Character set distributions							
		Numeric	Lowercase	Uppercase	Mixedcase	Loweralphenumeric	Upperalphanumeric	Mixedalphanumeric	Special
≤4	2,68%	38,47%	39,92%	2,06%	4,80%	3,46%	0,38%	0,08%	10,83%
5	3,60%	13,71%	57,27%	1,83%	5,19%	9,69%	0,53%	0,40%	11,39%
6	19,12%	25,25%	39,96%	1,21%	1,56%	28,40%	1,10%	1,21%	1,31%
7	15,53%	10,57%	37,88%	0,96%	1,68%	42,94%	1,38%	2,18%	2,42%
8	**20,68%**	13,51%	**30,36%**	0,61%	1,88%	44,76%	1,31%	4,78%	2,79%
9	12,26%	6,80%	30,23%	0,77%	1,59%	50,53%	1,50%	4,36%	4,22%
10	8,57%	9,96%	29,77%	0,49%	1,58%	46,18%	1,52%	5,14%	5,37%
11	4,22%	6,80%	27,46%	0,59%	2,31%	44,39%	1,47%	7,95%	9,05%
12	2,96%	3,37%	27,28%	0,52%	2,05%	45,02%	1,26%	8,44%	12,06%
13	1,48%	2,52%	20,62%	1,48%	3,24%	44,79%	2,48%	8,30%	16,56%
14	1,12%	3,23%	19,61%	1,29%	1,85%	44,27%	1,88%	9,10%	18,77%
15	0,97%	2,09%	18,66%	1,54%	2,19%	43,50%	2,12%	8,43%	21,46%
16	1,17%	3,97%	19,59%	2,24%	3,12%	34,59%	2,61%	12,65%	21,24%
≥17	5,64%	3,49%	21,25%	1,24%	1,65%	31,83%	1,87%	9,00%	29,68%

Table 10: Values for password length as a function of character set distributions

between a CMS that does not enforce a password policy by default and a CMS which applies a password policy. From the investigated CMS we identified that the majority of the CMS do not enforce a password policy by default, except for Magento CMS. To this end, in group 1 we include for the comparison a CMS named EspoCRM (which does not have a password policy) to Magento CMS (which by default uses a password policy). In particular, Magento policy accepts passwords that composed from at least 3 different charsets (i.e., numeric, lowercase, uppercase, special). Thus, for this comparison, we estimate the cost time of a brute force attack $cost\_time_{BF}$ for 8-character length mixedalphanumeric passwords for Magento (due to the password policy), and 8-character length lowercase passwords for EspoCRM (due to the absence of a password policy). Using equation (4) in section 4.2 and the input values derived in section 6.1 we calculate that for EspoCRM the $cost\_time_{BF}$ is equal to 3940 seconds, while for Magento is 8708036 seconds, which is a whopping 220.916% increase. This can be justified by the fact that password charset *C* of Magento is 62 (mixedalphanumeric – see Table 3) which greatly increases the required number of hashes for the brute force attack.

Observation 8*: A simple password policy such as the one of Magento, can have a drastic effect on the effort of the attacker to perform password guessing. Unfortunately, the majority of CMS and web application frameworks do not enforce the use of password policies, not even in the password length.*

Group 2 compares a CMS (i.e., Mibew) that uses BCRYPT with its lowest number of iterations (i.e., 2) among all CMS and web application frameworks as shown in Table 4, with a web application framework (i.e., Flask) that uses PBKDF2, which is the highest number of iterations (50.000 iterations) among all CMS and web application frameworks (see Table 4). The attack is brute force and since no password policy is enforced in these CMS, we select 8-character numeric passwords. The numerical results (see Table 11) show that even the lowest iterations of BCRYPT have significantly higher cost time (i.e., 2499488 seconds) compared to the highest iterations of PBKDF2 (i.e., 181814 seconds). This is due to the fact that BCRYPT reduces the level of parallelism [27]. As we mentioned in section 3, NIST guidelines [31] recommend PBKDF2 for hashing passwords with a minimum number of 10.000 iterations. Given our results, we argue that this recommendation is not adequate to withstand against offline passwords attacks.

Observation 9*. BCRYPT even only with 256 iterations provide significant improvements in terms of security over PBKDF2 with 50.000 iterations. Thus, we argue that not only the minimum recommended iterations of PBKDF2 by NIST is too low (i.e., 10.000), but also the recommended hash function itself (i.e., PBKDF2) is not resistant to password guessing.*

Group 3 investigates the effect of iterations for BCRYPT on the cost time in a dictionary attack. For this reason, we selected OpenCart, which uses 1024 iterations, and Zend framework, which uses the highest number of BCRYPT iterations among all CMS and web application frameworks (i.e. 16384). In this group, the derived numerical results of cost time are based on a dictionary attack. Specifically, we select a dictionary attack based on PCFG with $1.45 \cdot 10^6$ attempts and E_{DC}=41.5% (see first row of Table 8). As observed, an attacker needs 17302 seconds to guess a password for OpenCart (i.e., 1024 BCRYPT iterations), while this value increases to 276836 seconds for Zend Framework (i.e., 16384 BCRYPT iterations), which is an 1500% increase. Considering that the runtime of BCRYPT for 16384 iterations on a server is $6.8 \cdot 10^{-1}$ seconds (see Table 6), which is lower than the login delay threshold of one second (see section 6.1.1), OpenCart (and all other CMS using BCRYPT) can increase the value of iteration.

Observation 10*. Most CMS uses 1024 iterations for BCRYPT. This is attributed to the fact that the PHP programming language which all the CMS are based on, uses 1024 BCRYPT iterations*

by default. We argue that PHP can increase the default number of BCRYPT iterations (e.g., 16384) without imposing significant delays in the login procedure.

Group 4 aims at investigating the cost time of MHFs compared to BCRYPT. For this reason, we opt for phpBB which uses BCRYPT with 1024 iterations and a hypothetical website utilizing SCRYPT with 16384 iterations. Note that the recommended value of SCRYPT [28] is 16384. We select a dictionary attack based on PCFG using E_{DC}=41.5%. From numerical results we can deduce that the SCRYPT hash function increases the robustness of password hashing schemes, considering that an attacker needs 31376 seconds to crack a password. Moreover, the runtime of SCRYPT on servers is negligible, since it equals to $5.24 \cdot 10^{-2}$ seconds for 16384 iterations (see Table 6). From group 4 results, we can conclude to the following:

Observation 11. *As a long-term solution, we suggest CMS to upgrade their default hash function to a MHF, such as SCRYPT, which is resistant to password cracking and does not add login delays. Also NIST guidelines should replace PBDKF2 with a MHF. On a positive note recent 2017 NIST guidelines do suggest (but not impose) the use of MHF.*

Finally, group 5 aims at comparing the three most popular CMS namely WordPress, Joomla, and Drupal. WordPress, which is the most commonly used CMS, uses the weak MD5 hash function with 8192 iterations, while Drupal uses 65536 iterations of the highly parallelizable SHA512 hash function. On the contrary, Joomla uses BCRYPT with the PHP's default iteration value (i.e. 1024). As observed, a dictionary attack with E_{DC}=41.5% can crack a WordPress password in 2.4 seconds, while this value increases to 481 seconds for Drupal. The low level of parallelization of BCRYPT, has a significant impact on the $cost\_time_{DC}$ considering that an attacker needs 17302 seconds to crack a Joomla password hash. To conclude, the most secure CMS is Joomla, followed by Drupal, while WordPress is the most vulnerable to offline password guessing attacks despite it is the most widely used CMS.

	Attack	Target CMS	Password Policy	Hash function	Iterations	Attempts	Effectiveness	Cost time (sec)
Group 1	*BF*	Magento	✓	SHA256	1	62^8 (Pl =8, C=62)	E_{BF} =0.99%	8708036
		EspoCRM	✗	SHA512	1	26^8 (Pl =8, C=26)	E_{BF} =7.83%	3940
Group 2	*BF*	Flask	✗	$PBKDF2_{SHA256}$	50000	10^8 (Pl =8, C=10)	E_{BF} =2.79%	181814
		Mibew	✗	BCRYPT	256	10^8 (Pl =8, C=10)	E_{BF} =2.79%	2499488
Group 3	*DC*	OpenCart	✗	BCRYPT	1024	$1.45 \cdot 10^6$	E_{DC} =41.5%	17302
		Zend	✗	BCRYPT	16384	$1.45 \cdot 10^6$	E_{DC} =41.5%	276836
Group 4	*DC*	PhpBB	✗	BCRYPT	1024	$1.45 \cdot 10^6$	E_{DC} =41.5%	17302
		Hypothetical website	✗	SCRYPT	16384	$1.45 \cdot 10^6$	E_{DC} =41.5%	31376
Group 5	*DC*	WordPress	✗	MD5	8192	$1.45 \cdot 10^6$	E_{DC} =41.5%	2.4
		Drupal	✗	SHA512	65536	$1.45 \cdot 10^6$	E_{DC} =41.5%	481
		Joomla	✗	BCRYPT	1024	$1.45 \cdot 10^6$	E_{DC} =41.5%	17302

Table 11: Numerical results of the cost time for various CMS and web application frameworks.

7 Misuse of password hashing schemes for denial of service attacks

In this section we investigate whether hashing schemes can be misused to lead to denial of service attacks to web applications. The rationale behind the experiments was that resource intensive configurations of hashing schemes (e.g., high number of iterations) can deplete the CPU resources of the web server and eventually result in denial of service conditions. To this end, we deployed a custom version of the popular WordPress CMS using the Apache web

server. We implemented a plugin for WordPress with which we can easily modify and configure all the parameters of the hashing scheme, such as the hash function, the number of iterations, etc (see below for the parameter values of the hashing schemes). Finally, we wrote a script that performs multiple login requests with a registered username and random password values, forcing WordPress to hash and verify them. On the web server, we measured the CPU utilization using htop toolkit [50]. Regarding the hardware setup, we used a Intel Xeon E5-2640 v2 CPU and 4 GB memory running Ubuntu server 18.04, Apache 2.4.29 and PHP 7.2.

As shown in Table 12, the parameters of the experiment were: i) the hash function, ii) iterations, iii) password length and iv) rate (login requests per second). More specifically, we examined hash functions that are used. Particularly, we considered the following hash functions, which are the default ones for the 3 most popular CMS (i.e., WordPress, Joomla, Drupal). That is, we examined: i) MD5 as it is the default one used by WordPress, ii) SHA512 which is the default one of Drupal, and iii) BCRYPT used by Joomla. Apart from the above hash functions we also included in the experiments SCRYPT, which is a memory hard function as discussed in section 3. Moreover, the iterations value ranges from 1 to 65536 (2^{16}), while the password length ranges from 10 to 10000 characters. Lastly, the rate of the login requests per second of users varies from 1 to 30 requests per second.

Parameter	Values
Hash function	MD5, SHA512, BCRYPT, SCRYPT
Iterations (*I*)	1, 1024, 4096, 8192, 16384, 32768, 65536
Password length (pwd_length)	10, 1000, 5000, 10000
Rate (login requests per second)	1, 5, 10, 15, 20, 25, 30

Table 12: Parameters of the hashing schemes.

Figure 1 shows the CPU utilization as a function of the login rate for the MD5, SHA512, BCRYPT, and SCRYPT hash functions. In this experiment, we have used the default iteration values of the hash functions as they employed in the popular CMS. That is, we use: i) MD5 with 8192 iterations, as this is the default setting in WordPress, ii) BCRYPT with 1024 iterations, which is the default setting of Joomla iii) SHA512 with 65536 iterations, which is the default setting of Drupal. Moreover, to include also a MHF in the experiments, we use SCRYPT with 16384 iterations, as recommended in its specifications [28]. As it is observed, in all cases the increase of the CPU utilization is almost linear as the login rate increases. It is important to note that BCRYPT (i.e. Joomla), and SHA512 (i.e. Drupal) with their default settings could cause the CPU utilization to increase to 100% for rate equal to 20 and 25 requests respectively. By maintaining such CPU load, the web server cannot cope with the required login attempts, thus keeping occupied all the available Apache connections. This results in a denial of service at the application layer, since the web server cannot respond to new requests. A significant remark is that denial of service attacks realized even with 20-25 login requests per second, are not easily detectable by firewalls, if the logins are performed from different IPs (i.e., distributed denial of service). On the other hand, SCRYPT reaches 80% for rate equal to 30 requests per second. It is important to mention that during the experiments we observed that when CPU utilization reached 80%, the website was responsive but its pages were loading after a significant delay (i.e., 10-15 seconds). Therefore, although SCRYPT did not reach 100% CPU utilization, it was still capable of clogging the web server. On the other hand, Figure 1 suggests that MD5 cannot deplete the CPU resources as its increase rate is very slow and does not exceed 30% CPU utilization. Based on the above, we can conclude to the following observation:

Observation 12: *Slow rate denial of service attacks against websites that use hash functions with iterations are feasible (except for MD5). BCRYPT with 1024 iterations can reach 100%*

CPU utilization, even for login rate equal to 20 requests per second. This result is alarming considering that distributed denial of service attacks originated by botnets can far exceed the rates of our experiments. As mentioned in [51] *the majority of the distributed denial of service attacks in 2017 was performed using 100 to 1000 requests per second.*

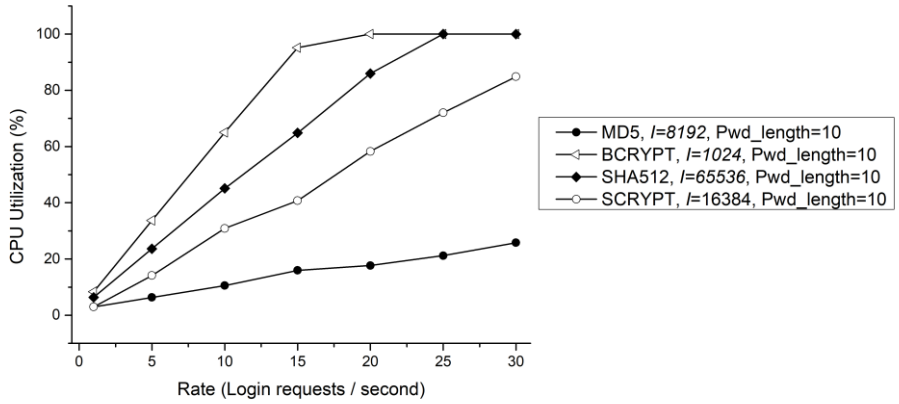

Figure 1: CPU utilization vs login rate

Although slow rate denial of service attacks are not easily detectable by intrusion detection systems and next generation firewalls [52], the nature of our considered denial of service based on password hashing has a weak point that defenders can take advantage of, to withstand websites against this attack. In particular, by using a mechanism called rate-limit (aka throttle), a website can block the usernames related to the incorrect logins, for a specific time period when a predefined threshold of failed consecutive attempts is reached. In this way, attackers cannot continue performing the denial of service for a long time period, since eventually all the usernames under the possession of the attacker will be blocked and the related login attempts will be discarded. Another beneficial characteristic of this solution lies to the fact that the rate limit can be applied at the application layer. As a matter of fact, there are many ready to use free CMS plugins, (such as [53] for WordPress) or a middleware for web application frameworks (such as [54] for CakePHP) that an administrator/developer can consider to use.

Observation 13: *It is imperative to employ rate-limit in websites to mitigate denial of service attacks based on concurrent login attempts. The rate limit of login attempts is an effective and easy to deploy security mechanism available in many CMS and web applications frameworks. NIST guidelines consider as highly important to enforce rate limits and recommend maximum 100 failures account* [31].

In the next two experiments we will investigate if password length and iterations can cause denial of service attacks even for very slow rates. More specifically, Figure 2 shows the CPU utilization versus the password length for the same hash functions and iterations number as in the previous experiment. The rate of attempts is equal to 1 request per second. The first and most important finding is that SHA512 with 65536 iterations (i.e., Drupal default settings) is vulnerable to denial of service attacks, since the CPU utilization reaches 100% for password length equal to 6000. MD5 has also an increasing behavior but reaches almost 15% CPU utilization for password length equal to 10.000. This happens because MD5 and SHA512 do not have a maximum acceptable password length. On the contrary, BCRYPT has a constant CPU utilization independent from the password length, because the maximum password length for BCRYPT is 72 characters. Lastly, although SCRYPT does not have a password length limitation, its' CPU utilization does not change significantly, possibly due to its fast runtime on CPUs (see Table 6). Based on the above results, we infer that CMS and application frameworks should set by default a maximum acceptable password length policy to avoid denial of service

with very large passwords. We discovered that WordPress by default limits to 4096 characters, while Drupal limits even more the password length to 128 characters.

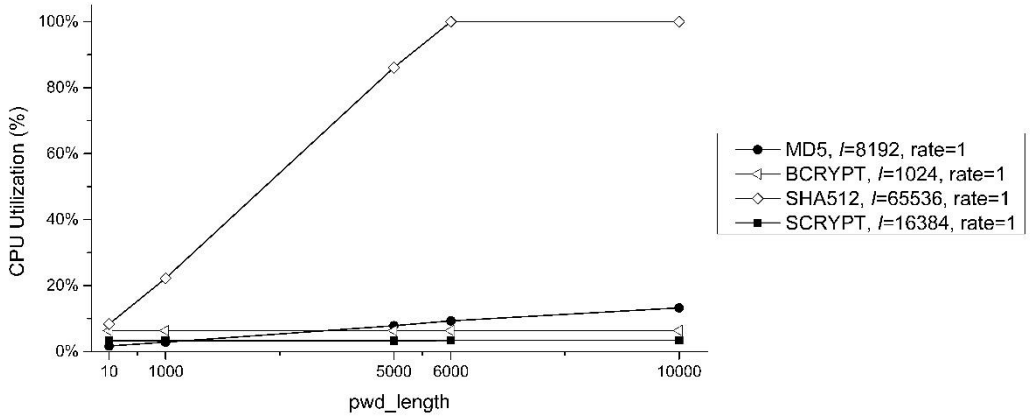

Figure 2: CPU utilization vs password length

Observation 14: *All websites that use SHA1, SHA256,SHA512 or PBKDF2 with very high number of iterations should accordingly limit the maximum password length similarly to WordPress and Drupal to avoid falling victim of denial of service. On the other hand, BCRYPT and SCRYPT are not susceptible to denial of service with large passwords.*

Finally, Figure 3 shows the CPU utilization as a function of iterations. In this experiment, we use a small password length and slow login rate, equal to 10-character and 1 request/sec respectively. From Figure 3 we can observe that in all cases the CPU utilization increases with iterations. However, increasing iterations we also increase the resistance of passwords against guessing attacks. In other words, the iterations regulate an inherent tradeoff between security and performance. In particular, as the number of iterations increases, on the one hand the password hashes are more resistant to guessing attacks (security), but on the other hand CPU utilization is increased (performance). Figure 3 depicts also that BCRYPT is vulnerable to denial of service, since it reaches 100% CPU utilization with 32768 iterations, while SCRYPT reaches only 25% CPU utilization for 65536 iterations. At the same time, the runtime for SCRYPT is lower than 1 second in typical server machine (see Table 6), which makes it suitable for interactive logins, due to its small authentication delay. Subsequently, we can conclude to the following observation:

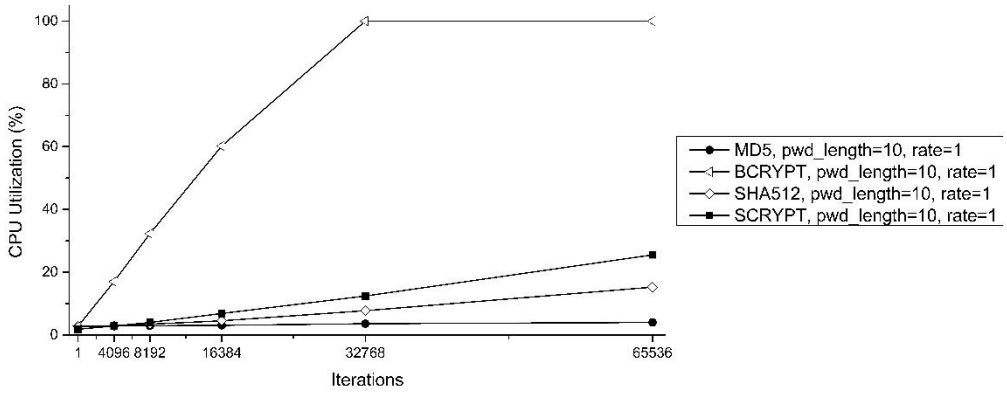

Figure 3: CPU utilization vs iterations

Observation 15: *Compared to BCRYPT, SCRYPT is more scalable in the sense that the number of iterations can be increased for password security without introducing denial of service*

conditions and login delays provided that the web server has enough physical memory (>4 GB).

8 Discussion

In light of our analysis, this section provides recommendations and alternative solutions to enhance robustness of passwords against guessing attacks.

Update NIST recommendations. As mentioned previously, NIST recommends the use of PBKDF2 with 10.000 iterations minimum. Based our observations, we believe that NIST guidelines should be updated to replace PBKDF2 with a MHF, which is adequately audited and proved that it is robust against attacks.

Use of secure default settings. One of the most influential insights from the behavioral sciences is that whatever is in the "default" position generally persist. Thus, CMS developers should shift from an "opt-in" to an "opt-out" policy with stronger security configurations. Web application frameworks should also follow this practice and avoid making the assumption that developers are able to select secure and appropriate hashing schemes (e.g., use of salt, password policy, etc.).

Upgrade legacy hash functions. Regarding legacy hash functions, it is a fact that many websites have remained with outdated hash functions such as MD5 or SHA1. The problem that hinders adoption of a new hash function is the possible frustration to the users of the website, because they will be forced to register once again to provide a new password for the new hash function [55]. We argue that there are two possible ways to upgrade a hash function without the need of a new registration. The first solution is to keep two tables side by side one with the old hash function (e.g., MD5) and another table for the new hash function. When a user logs in for the first time after the addition of the new hash function, the website will first verify the legacy hash (e.g., MD5) and then store the new hash (derived from the new hash function). When all the new hashes have been calculated by all users, then the website can delete the old table with the MD5 hashes. This solution is feasible only for a small number of users, otherwise it could take an extremely long time to achieve the migration to the new hash function. The second solution is called layered hashing scheme and it has been adopted by Facebook [56] (see Figure 4). The idea is to use multiple hashes one after the other. That is, the output of a hash function becomes input for another hash function. In this way, a website can update a hash function at any time simply by adding a new layer of a hash function, eliminating the need to maintain two separate tables and wait the users to log in first. In the case of Facebook, the layered hashing scheme is as follows:

1. H = md5(pwd) (the legacy hash function)
2. H = $hmac_{sha1}$(H, K1, salt) (K1 is a secret
3. H = Cryptoservice::hmac(H, K2) (K2 is a secret key stored in the cryptoservice)
4. H = scrypt(H, salt) (the new key hash function. Depending on the implementation SCRYPT output length can be several bytes)
5. H = $hmac_{sha256}$(H, K3, salt) (this hash function is used to limit the output length to 256 bits)

Figure 4: Layered Hashing scheme of Facebook

Note that in step 3, the Cryptoservice::hmac(H, K) refers to the computation of a hash value by an external service (see below for analysis) using a keyed HMAC function (this is known as distributed hashing – see below). In the example of Facebook, the output of the legacy MD5 (i.e., step 1) is being used as an input to multiple hash function including a $HMAC_{SHA1}$ in step2, another HMAC value (with unknown hash function) in a remote cryptoservice (i.e., step 3), an SCRYPT (i.e., step 4), and finally a $HMAC_{sha256}$ (i.e., step 5). Therefore, using this layered

approach, a hash function can be updated without causing disruptions to the normal operation of the website.

Distributed hashing. A solution which is orthogonal to the actual hash function that a website uses and can substantially protect against offline password guessing attacks is named distributed hashing. The main idea of this solution lies in the delegation of the hash value computation to an external service. More specifically, a hashing scheme which is composed of multiple hash functions as the one presented previously in Figure 4 can offload the computation of an intermediate hash calculation to a remote crypto service (aka crypto as a service) and send back the hashed value back to the web application to continue the calculation of the hash value. Note that the hash calculation in the cryptoservice is based on a keyed HMAC function, using a secret key, which is stored in the cryptoservice (see step 3 in Figure 4). In this way, even if an attacker is able to compromise the database of a web platform, in order to perform the guesses, he should necessarily request the cryptoservice to obtain the intermediate hash value, since the attacker does not possess the secret key for the HMAC function. In this way, the offline guessing attack becomes an online attack, which means that the cryptoservice can detect anomalies (i.e., a spike due to attempts of the attacker) and throttle appropriately the traffic (thus reducing the number of attempts an attacker can perform). Of note, recently a new research area has emerged [57] [58] [59] where the aim is to enhance the cryptographic primitives used in distributed hashing schemes to eliminate possible attacks against crypto services.

Federation and FIDO. Moreover, websites can opt for federated authentication solution using OpenID Connect protocol. In this way, there is no need for websites to maintain a user database including passwords, due to the delegation of authentication to established services such as Google and Facebook. On the users' side, good security practices for selecting passwords are still relevant. Users should select high entropy long passwords and avoid reusing passwords across multiple websites. What is more, passwords managers and two-factor authentication are traditional yet effective measures to resist against password cracking. Also, the emerging FIDO protocol [60], which is based on device-centric authentication, aims to eliminate the use of passwords using public key cryptography.

Server relief. Regarding denial of service attacks that take advantage of intensive hash functions to overload web servers, these can be mitigated by the use of a relatively new mechanism named server relief. As a matter of fact, Argon2 has adopted this solution to facilitate web servers to withstand against denial of service attacks. The rationale of server relief mechanism is to allow the server to carry out the majority of computational burden on the client. That is, instead of doing the entirety of the computation on the server, the client does the most demanding - in terms of computation - parts and then the client sends the intermediate values to the server, which calculates the final hash value. Evidently, all intermediate values on the client side should not leak any information for the actual password. An overview of various server relief solutions highlighting advantages and drawbacks can be found in [61].

9 Conclusions

This paper has evaluated the security of hashing schemes for popular CMS and web application frameworks. We proposed a framework that allows us to quantify the cost time of password guessing both for brute force and dictionary attacks. Next, we identified the default hashing schemes of various CMS and web applications frameworks and based on our findings we derived a set of critical observations. We concluded that the majority of CMS and web applications frameworks do not offer secure default settings. We observed usage of outdated hash functions, arbitrary number of iterations, lack of password policies and salt. Next, we applied our cost analysis framework to the default settings, in order to perform a comparative security analysis between the various CMS and web applications frameworks. We also

investigated whether hashing schemes can be misused to lead to denial of service attacks. Finally, we provided a set of best practices and alternative solutions to enhance the security of password storage. In general, we believe that the security status of the hashing schemes calls for changes with new recommendations and updates. Based on our analysis we advocate that password hashing standards should be updated to require and not merely suggest the use of MHF. It would be also beneficial, policy makers to audit and penalize organizations that fail to follow appropriate standards for password hashing.

10 Acknowledgement

This work was supported in part by the FutureTPM project of Horizon H2020 Framework Programme of the European Union under GA number 779391, and by the H2020-MSCA-RISE-2017 SealedGRID project under GA number 777996.

References

[1] "World's Biggest Data Breaches," [Online]. Available: http://www.informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/. [Accessed May 2018].

[2] G. Vindu and N. Perlorth, "Yahoo Says 1 Billion User Accounts Were Hacked," New York Times, 14 December 2016. [Online]. Available: https://www.nytimes.com/2016/12/14/technology/yahoo-hack.html. [Accessed April 2018].

[3] A. Ghoshal, "Yahoo's billion-user database reportedly sold on the Dark Web for just $300,000," The next web, January 2017. [Online]. Available: https://thenextweb.com/security/2016/12/16/yahoos-billion-user-database-reportedly-sold-on-the-dark-web-for-just-300000/#.tnw_7j4OqioP. [Accessed April 2018].

[4] "GEFORCE NVidia TITAN V," NVIDIA, [Online]. Available: https://www.nvidia.com/en-us/titan/titan-v/. [Accessed 8 May 2018].

[5] "Google," [Online]. Available: https://cloud.google.com/gpu/. [Accessed 7 May 2018].

[6] A. Naiakshina, A. Danilova, C. Tiefenau, M. Herzog, S. Dechand and M. Smith, "Why Do Developers Get Password Storage Wrong?: A Qualitative Usability Study," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[7] M. Weir, S. Aggrawal and B. d. Medeiros, "Password Cracking Using Probabilistic Context-Free Grammars," in *30th IEEE Symposium on Security and Privacy*, 2009.

[8] A. Narayanan and V. Shmatikov, "Fast Dictionary Attacks on Passwords Using Time-Space Tradeoff," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, Virgina, 2005.

[9] S. Marechal, "Automatic mangling rules generation," December 2012. [Online]. Available: http://www.openwall.com/presentations/Passwords12-Mangling-Rules-Generation/Passwords12-Mangling-Rules-Generation.pdf. [Accessed 8 May 2018].

[10] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk and W.-M. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU

using CUDA," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, New York, 2008.

[11] T. Murakami, R. Kasahara and T. Saito, " An implementation and its evaluation of password cracking tool parallelized on GPGPU," in *10th International Symposium on Communications and Information Technologies*, Tokyo, 2010.

[12] "Usage of content management systems for websites," W3Techs, [Online]. Available: https://w3techs.com/technologies/overview/content_management/all. [Accessed July 2018].

[13] "Github: Web application frameworks," [Online]. Available: https://github.com/showcases/web-application-frameworks?s=stars. [Accessed July 2018].

[14] "http://www.openwall.com/john/," Openwall, [Online]. Available: http://www.openwall.com/john/. [Accessed April 2018].

[15] E. I. Tatli, "Cracking More Password Hashes With Patterns," *IEEE Transactions on Information Forensics and Security,* vol. 10, no. 8, pp. 1656-1665, 2015.

[16] "Passwords," Skullsecurity, [Online]. Available: https://wiki.skullsecurity.org/Passwords. [Accessed April 2018].

[17] W. Han, Z. Li, L. Yuan and W. Xu, "Regional Patterns and Vulnerability Analysis," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 2, pp. 258-272, 2016.

[18] M. Dürmuth, F. Angelstorf, C. Castelluccia, D. Perito and A. Chaabane, "OMEN: Faster Password Guessing Using an Ordered Markov Enumerator," *International Symposium on Engineering Secure Software and Systems*, pp. 119-132, 2015.

[19] M. D. Amico, P. Michiardi and Y. Roudier, "Password Strength: An Empirical Analysis," in *Proceedings of the 29th conference on Information communications (INFOCOM 2010)*, 2010.

[20] "The Imperva Application Defense Center (ADC) - Consumer Pasword Worst Practices," [Online]. Available: https://www.imperva.com/docs/gated/WP_Consumer_Password_Worst_Practices.pdf. [Accessed Apr 2018].

[21] C. McGoogan, "The world's most common passwords revealed: Are you using them?," The Telegraph, January 2017. [Online]. Available: http://www.telegraph.co.uk/technology/2017/01/16/worlds-common-passwords-revealed-using/. [Accessed May 2018].

[22] B. Lorenz, K. Kikkas and A. Klooster, "The Four Most-Used Passwords Are Love, Sex, Secret, and God": Password Security and Training in Different User Groups," in *Human Aspects of Information Security, Privacy, and Trust: First International Conference*, Las Vegas, Springer Berlin Heidelberg, 2013, pp. 276-283.

[23] R. Rivest, "The MD5 Message-Digest Algorithm," Apr. 1992. [Online]. Available: https://www.ietf.org/rfc/rfc1321.txt. [Accessed June 2018].

[24] D. Eastlake, "US Secure Hash Algorithm 1 (SHA1)," Sept 2001. [Online]. Available: https://tools.ietf.org/html/rfc3174. [Accessed June 2018].

[25] D. Eastlake, "US Secure Hash Algorithms (SHA and HMAC-SHA)," Jul 2006. [Online]. Available: https://tools.ietf.org/html/rfc4634. [Accessed 2 Sept 2007].

[26] B. Kaliski, "PKCS #5: Password-Based Cryptography Specification Version 2.0," RSA Laboratories, Sept 2000. [Online]. Available: https://tools.ietf.org/html/rfc2898. [Accessed 3 Sept 217].

[27] N. Provos and D. Mazières, "A Future-Adaptable Password Scheme," in *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*, 1999.

[28] C. Percival, "Stronger Key Derivation via Sequential Memory-Hard Functions," 2009. [Online]. Available: https://www.tarsnap.com/scrypt/scrypt.pdf. [Accessed April 2018].

[29] A. Biryukov, D. Dinu and D. Khovratovich, "Technical Report: Argon and argon2: password hashing scheme," 2015. [Online]. Available: https://password-hashing.net/submissions/specs/Argon-v2.pdf..

[30] I. E. T. F. (IETF), "RFC 7914: The scrypt Password-Based Key Derivation Function," August 2016. [Online]. Available: https://tools.ietf.org/html/rfc7914.

[31] "NIST Special Publication 800-63B: Digital Identity Guidelines Authentication and Lifecycle Management," June 2017. [Online]. [Accessed July 2018].

[32] "PHP - password_hash()," [Online]. Available: http://php.net/manual/en/function.password-hash.php. [Accessed July 2018].

[33] A. Visconti, S. Bossi, H. Ragab and A. Calò, "On the weaknesses of PBKDF2," in *International Conference on Cryptology and Network Security (CANS 2015)*, Marrakesh, Morocco, 2015.

[34] A. Ruddick and J. Yan, "Acceleration Attacks on PBKDF2: Or, What Is inside the Black-Box of oclHashcat?," in *10th USENIX Workshop on Offensive Technologies*, 2016.

[35] "bcrypt on GPU," Openwall community wiki, [Online]. Available: http://openwall.info/wiki/john/GPU/bcrypt. [Accessed May 2018].

[36] F. Wiemer and R. Zimmermann, "High-speed implementation of bcrypt password search using special-purpose hardware," in *International Conference on ReConFigurable Computing and FPGAs*, 2014.

[37] K. Malvoni, S. Designer and J. Knezovic, "Are Your Passwords Safe: Energy-Efficient Bcrypt Cracking with Low-Cost Parallel Hardware," in *8th USENIX Workshop on Offensive Technologies*, 2014.

[38] "Password Hashing Competition," [Online]. Available: https://password-hashing.net.

[39] P. Pierluigi, "Lenovo spotted and fixed a backdoor in RackSwitch and BladeCenter networking switches," SecurityAffairs.co, [Online]. Available: https://securityaffairs.co/wordpress/67729/hacking/lenovo-backdoor-networking-switches.html. [Accessed July 2018].

[40] L. Armasu , "Backdoors Keep Appearing In Cisco's Routers," Tom's Hardware, [Online]. Available: https://www.tomshardware.com/news/cisco-backdoor-hardcoded-accounts-software,37480.html. [Accessed July 2018].

[41] T. McLean, "Critical vulnerabilities in JSON Web Token libraries," Auth0.com, [Online]. Available: https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/. [Accessed July 2018].

[42] "Phorum - Improving md5 password storage security," [Online]. Available: https://www.phorum.org/phorum5/read.php?14,155691,155691. [Accessed June 2018].

[43] "Magento - Use native PHP Password API," [Online]. Available: https://github.com/magento/magento2/issues/992. [Accessed July 2018].

[44] B. P. Knijnenburg, A. Kobsa and H. Jin, "Counteracting the Negative Effect of Form Auto-completion on the Privacy Calculus," in *AIS Electronic Library (AISeL)*, 2013.

[45] "Hashcat," [Online]. Available: https://hashcat.net/hashcat. [Accessed June 2018].

[46] "GeForce 1070 / 1070 Ti," [Online]. Available: https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1070-ti/.

[47] J. Blocki, B. Harsha and S. Zhou, "On the Economics of Offline Password Cracking," in *IEEE Symposium on Security and Privacy (SP)*, 2018.

[48] G. Rempel, ""Defining Standards for Web Page Performance in Business Applications," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering ICPE '15*, 2015.

[49] "Vigilante.pw," [Online]. Available: https://vigilante.pw/.

[50] "march 2018 Web server Survey," Netcraft, [Online]. Available: https://news.netcraft.com/archives/2018/03/27/march-2018-web-server-survey.html.

[51] "Global DDOS Threat Landscape Q4 2017," Incapsula, [Online]. Available: https://www.incapsula.com/ddos-report/ddos-report-q4-2017.html. [Accessed July 2018].

[52] K. Ronen, "Why Low & Slow DDoS Application Attacks are Difficult to Mitigate," [Online]. Available: https://blog.radware.com/security/2013/06/why-low-slow-ddosattacks-are-difficult-to-mitigate/. [Accessed July 2018].

[53] Arshid, "WP Limit Login Attempts," [Online]. Available: https://wordpress.org/plugins/wp-limit-login-attempts/. [Accessed June 2018].

[54] "(API) Rate limiting requests in CakePHP 3," Github, [Online]. Available: https://github.com/UseMuffin/Throttle. [Accessed May 2018].

[55] B. Schneier, "Schneier on Security: Changing Passwords," [Online]. Available: https://www.schneier.com/blog/archives/2010/11/changing_passwo.html.

[56] A. Muffett, "Facebook: Password Hashing & Authentication," in *Real World Crypto* , 2015.

[57] J. Camenisch, A. Lysyanskaya and G. Neven, "Practical yet universally composable two-server password-authenticated secret sharing," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012.

[58] A. Everspaugh, R. Chatterjee, S. Scott, A. Juels and T. Ristenpart, "The pythia PRF service," in *SEC'15 Proceedings of the 24th USENIX Conference on Security Symposium*, 2015.

[59] R. F. Lai, C. Egger, D. Schröder and S. S. M. Chow, "Phoenix: Rebirth of a Cryptographic Password-Hardening Service," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[60] " FIDO Alliance," [Online]. Available: https://fidoalliance.org/. [Accessed July 2018].

[61] S. Contini, "Online report: Method to Protect Passwords in Databases for Web," [Online]. Available: https://eprint.iacr.org/2015/387.pdf.

[62] A. S. brown, E. Bracken, S. Zoccoli and K. Douglas, "Generating and remembering passwords," *Applied Cognitive Phychology*, vol. 18, no. 6, pp. 641-651, 2004.

[63] M. J. Yang, W. Luo and N. Li, "A study of probabilistic password models," in *IEEE Symposium on Security and Privacy*, 2014.