ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

UNIVERSITY OF PIRAEUS

User authentication and detection of malicious actions

A dissertation submitted to the
Department of Digital Systems, of
University of Piraeus in complete
fulfillment of the requirements for
the degree of Doctor of Philosophy

by

Stefanos Malliaros

B.Sc. School of Information and
Communication Technologies, Department of
Digital Systems, University of Piraeus

M.Sc. School of Information and
Communication Technologies, Department of
Digital Systems, University of Piraeus

Piraeus 2018

Advisory Committee

Christos Xenakis – Associate Professor, School of Information and Communication Technologies, Department of Digital Systems, University of Piraeus (supervisor)

Konstantinos Lambrinoudakis – Professor, School of Information and Communication Technologies, Department of Digital Systems, University of Piraeus

---

Sokratis Katsikas – Professor, School of Information and Communication Technologies, Department of Digital Systems, University of Piraeus

Examination Committee

Christos Xenakis – Associate Professor, School of Information and Communication Technologies, Department of Digital Systems, University of Piraeus (supervisor)

Konstantinos Lambrinoudakis – Professor, School of Information and Communication Technologies, Department of Digital Systems, University of Piraeus

---

Sokratis Katsikas – Professor, School of Information and Communication Technologies, Department of Digital Systems, University of Piraeus

---

Stefanos Gritzalis – Professor, School of Engineering, Department of Information & Communication Systems Engineering, University of the Aegean

---

Vassilis Chrissikopoulos – Professor, Department of Informatics, Ionian University

---

Emmanouil Magkos – Associate Professor, Department of Informatics, Ionian University

---

Panagiotis Rizomiliotis – Associate Professor, School of Digital Technology, Department of Informatics and Telematics, Harokopio University

Acknowledgements

I would like to thank my supervisor, Dr. Christos Xenakis, for his guidance and support throughout the years of being a PhD candidate. Moreover, I would like to thank him for selecting me as one of his core colleagues. I would also like to thank Dr. Costas Lambrinoudakis and Dr. Sokratis Katsikas for all their constructive comments and productive discussions not only during my PhD, but also during my master course.

Very special thanks to Dr. Christoforos Dadoyan for proving his knowledge, patience, and guidance during my PhD period. I would also like to thank the members of the PhD review committee for agreeing to serve on my dissertation committee.

A big “Thank You!” to Eleni Veroni for being an excellent friend, and colleague. You provided valuable support these years.

Finally, I would like to thank my wife Chara because she has always supported my decisions and has been with me throughout these years both in bad and good situations. I cannot leave out my family and sister. I greatly express my gratitude for believing in me and for all the sacrifices you made for me.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου Δρ. Χρήστο Ξενάκη για την καθοδήγηση και την υποστήριξή του καθ' όλη τη διάρκεια της ιδιότητας μου ως υποψήφιος διδάκτορας. Επιπλέον δε θα μπορούσα να μην τον ευχαριστήσω για την επιλογή μου ως έναν από τους βασικούς συνεργάτες του. Θα ήθελα επίσης να ευχαριστήσω τους καθηγητές Δρ. Κώστα Λαμπρινουδάκη και Δρ. Σωκράτη Κατσικά για όλες τις εποικοδομητικές παρατηρήσεις τους και παραγωγικές συζητήσεις, τόσο κατά τη διάρκεια του διδακτορικού μου όσο και κατά τη διάρκεια του μεταπτυχιακού προγράμματος σπουδών.

Ιδιαίτερες ευχαριστίες στον Δρ. Χριστόφορο Νταντογιάν για τις γνώσεις που μου προσέφερε, την υπομονή και την καθοδήγησή του κατά τη διάρκεια των διδακτορικών μου σπουδών. Θα ήθελα, επίσης, να ευχαριστήσω ιδιαιτέρως τα μέλη της εξεταστικής επιτροπής για την πολύτιμη βοήθειά τους στην ολοκλήρωση αυτού του κύκλου σπουδών.

Ένα μεγάλο "Ευχαριστώ!" για την Ελένη Βερώνη η οποία στάθηκε εξαιρετική φίλη και συνάδελφος.

Θα ήθελα, επιπλέον, να ευχαριστήσω τη σύζυγό μου, Χαρά, για τη συνεχή στήριξη της στις αποφάσεις μου καθ' όλη τη διάρκεια των ετών. Στέκεσαι συνεχώς δίπλα μου τόσο σε κακές όσο και σε καλές καταστάσεις. Τέλος, δεν θα μπορούσα να μην αναφέρω την οικογένειά μου και την αδελφή μου. Είμαι, λοιπόν, ευγνώμων για την υποστήριξή τους προς μένα και για όλες τις θυσίες που κάνατε για την πραγμάτωση των δικών μου στόχων.

Abstract

Modern devices can carry out potentially dangerous actions, such as storing corporate and personal data, performing electronic transactions, accessing health data, and many more. All these actions introduce the ability to securely access increasingly personal information, which, in fact, raises the problem of user authentication. The usage of passwords introduces critical security issues due to their predictability, while tokens are not resistant to malware attacks, such as key loggers and memory scrapers.

These issues can only be addressed by holistically investigating the problem of user authentication. The security of online accounts is drastically affected by the password predictability, as well as the parameters for password storage. Therefore, we propose a mathematical model, based on the parameters that influence password security. The main goal is to discover the cost of password guessing. Moreover, an extended survey of the default password storage parameters indicates that a significant percentage of websites use insecure password hashing. We have proved that the cost of password guessing can be a measure of defense to password guessing attacks.

Apart from password storage, the security of user accounts relies on the protocols used for authentication, as well as the feasibility of obtaining the user credentials via malware. As a result, we explore the security of FIDO authentication framework, which replaces passwords with biometric modalities. The result of the analysis is a list of vulnerabilities that may be exploited by an attacker to compromise the authenticity, privacy, availability, and integrity of the FIDO. Moreover, as recent research has shown, authentication credentials and cryptographic keys remain in the volatile memory and can be easily extracted by malware. Therefore, we present safeguards that can be applied to the software level, either from the operating system or the applications, to erase data in the volatile memory from running and terminated applications.

Lastly, with continuous authentication, users are continually authenticated via a "score", which measures the certainty that the account owner is using a service or application. Therefore, we propose gaithashing, which is a secure two-factor authentication scheme based on the gait modality. The proposed scheme eliminates the noise and distortions caused by different silhouette types and achieves to authenticate a user independently of his/her silhouette. Lastly, this thesis proposes a novel technique to detect malicious actions using machine learning. This has been applied in the context of Ad hoc networks, where a new critical attack parameter has been identified. This parameter can be used to quantify the relation between AODV's sequence number parameter and the performance of blackhole attacks.

Περίληψη

Οι σύγχρονες συσκευές μπορούν να πραγματοποιούν δυνητικά επικίνδυνες ενέργειες, όπως η αποθήκευση εταιρικών και προσωπικών δεδομένων, η εκτέλεση ηλεκτρονικών συναλλαγών, η πρόσβαση σε δεδομένα υγείας και πολλά άλλα. Όλες αυτές οι ενέργειες επιτρέπουν την ασφαλή πρόσβαση σε ευαίσθητες πληροφορίες, γεγονός που διεγείρει το πρόβλημα επαλήθευσης χρήστη. Η χρήση των κωδικών πρόσβασης εισάγει κρίσιμα ζητήματα ασφαλείας, ενώ η αυθεντικοποίηση δύο παραγόντων δεν είναι ανθεκτική σε επιθέσεις κακόβουλου λογισμικού.

Αυτά τα προβλήματα μπορούν να αντιμετωπιστούν μόνο με ολιστική διερεύνηση του προβλήματος της πιστοποίησης ταυτότητας χρήστη. Η ασφάλεια των online λογαριασμών επηρεάζεται δραστικά από την προβλεψιμότητα των κωδικών πρόσβασης, καθώς και από τον τρόπο αποθήκευσης τους. Ως εκ τούτου, προτείνουμε ένα μαθηματικό μοντέλο βασισμένο στις παραμέτρους που επηρεάζουν την ασφάλεια των κωδικών πρόσβασης. Ο σκοπός είναι ο υπολογισμός του κόστους επιθέσεων password guessing. Επιπλέον, πραγματοποιούμε μια ενδελεχή έρευνα των παραμέτρων προεπιλεγμένης αποθήκευσης κωδικού πρόσβασης που δείχνει ότι ένα σημαντικό ποσοστό των ιστότοπων χρησιμοποιούν μη ασφαλείς τρόπους αποθήκευσης κωδικών πρόσβασης. Έχουμε αποδείξει πως το κόστος των password guessing επιθέσεων μπορεί να είναι ένας τρόπος άμυνας απέναντι σε αυτές.

Εκτός από την αποθήκευση των κωδικών πρόσβασης, η ασφάλεια των λογαριασμών χρηστών βασίζεται στα πρωτόκολλα που χρησιμοποιούνται για τον έλεγχο ταυτότητας, καθώς και στη δυνατότητα διαρροής τους μέσω κακόβουλου λογισμικού. Επομένως, εξετάζουμε την ασφάλεια του πλαισίου ελέγχου ταυτότητας FIDO, το οποίο αντικαθιστά τους κωδικούς πρόσβασης με βιομετρικά χαρακτηριστικά. Το αποτέλεσμα της ανάλυσης είναι μια λίστα ευπαθειών που μπορεί να εκμεταλλευτεί ένας εισβολέας για να θέσει σε κίνδυνο την αυθεντικότητα, την ιδιωτικότητα, τη διαθεσιμότητα και την ακεραιότητα του FIDO. Επιπλέον, καθώς πρόσφατες έρευνες έχουν δείξει ότι τα πιστοποιητικά ελέγχου ταυτότητας και τα κρυπτογραφικά κλειδιά παραμένουν και μπορούν να διαρρεύσουν μέσω της πτητικής μνήμης, παρουσιάζουμε τεχνικές που μπορούν να εφαρμοστούν σε επίπεδο λογισμικού, είτε από το λειτουργικό σύστημα είτε από τις εφαρμογές, με σκοπό την διαγραφή των κωδικών πρόσβασης από την πτητική μνήμη.

Τέλος, με την χρήση της συνεχούς αυθεντικοποίησης, οι χρήστες επαληθεύονται συνεχώς μέσω μίας μέτρησης, η οποία μετρά τη βεβαιότητα ότι ο κάτοχος του λογαριασμού χρησιμοποιεί είτε την υπηρεσία ή την εφαρμογή. Ως εκ τούτου, προτείνουμε ένα ασφαλές σύστημα ταυτοποίησης δύο φάσεων, το οποίο εφεξής θα αναφέρετε ως gaithashing, βασισμένο στη μέθοδο βάδισης. Το προτεινόμενο σχήμα εξαλείφει το θόρυβο και τις παραμορφώσεις που προκαλούνται από διαφορετικούς τύπους σιλουέτας του χρήστη και επιτυγχάνει την εξακρίβωση της ταυτότητας του ανεξάρτητα από τη σιλουέτα του. Τέλος, αυτή η διατριβή προτείνει μια νέα τεχνική για την ανίχνευση κακόβουλων ενεργειών που βασίζεται στην μηχανική μάθηση. Αυτή εφαρμόστηκε στο πλαίσιο των δικτύων ad hoc, όπου ορίστηκε μια νέα παράμετρος, η οποία ποσοτικοποιεί τη σχέση μεταξύ των αριθμών ακολουθίας του AODV και της απόδοσης επιθέσεων τύπου blackhole.

Table of Contents

List of Figures

List of Tables

1. Introduction

Modern devices can carry out potentially dangerous actions, such as storing corporate and personal data, performing electronic transactions, accessing health data, etc. All these actions introduce the ability to securely access increasingly personal information, which in fact raises the problem of user authentication. Currently, user authentication and access control are mainly carried out based on the usage of passwords or tokens. However, these mechanisms present fundamental limitations in terms of both security and usability. On the one hand, short length passwords are usually of low entropy, which means that an attacker may guess them, while lengthy passwords are difficult to remember. This results in the reuse of a password or the creation similar passwords for each service, which increases significantly the risk of a password to be broken and the associated services to be compromised. On the other hand, tokens can be easily stolen, while they are not resistant to malware attacks, such as key loggers and memory scrapers.

User authentication is the process of determining whether someone is, in fact, who he declares to be. This is usually performed by checking if a user's credentials match the credentials in a database of authorized users. Several corporates [1] have become victims of security breaches, resulting in the disclosure of billions of stored user 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.

Apart from password guessing attacks, that target passwords originating from an online database, users are also threatened from malware that can steal their authentication credentials in real time. Recent research has shown that authentication credentials and cryptographic keys remain in the volatile memory and can be easily extracted [2]. For this reason, the volatile memory has become a prime target for malicious software. As a matter of fact, a new malware category has emerged named as

memory scrapers, which specifically target the volatile memory, to steal sensitive information, such as credit card numbers [5]. To achieve this, memory scrapers use regular expression matches, to harvest credit card data from the volatile memory, and then the collected data are sent to a malicious server. The first known memory scraper, named StarDust targeted point of sale terminals and compromised nearly 20.000 credit cards in the US [7].

1.1. Research Contribution and structure

The first part of this work (see section 2) focuses on studying the security of the currently used methodologies for user authentication. This is performed by proposing a mathematical model, based on the parameters that influence password security, for estimating the cost of brute force and dictionary password guessing attacks. By performing an extended survey on the hashing performance of graphics processing units, we applied the proposed model to the most commonly used CMSs and web application frameworks to investigate whether they offer secure password hashing. Although, the first observations of the first part showed that a significant percentage of websites use insecure password hashing, we proved that the cost of password guessing can be a measure of defense to password guessing attacks.

The second part of this work (see section 3) investigates already existing solutions that offer advantages over traditional authentication mechanisms. Therefore, we explore the FIDO UAF protocol by comprehensively analyzing the client-side operation, including any associated security measures proposed by the UAF protocol specifications. The critical functionality of the UAF protocol typically operates in a consumer platform such as a mobile device, which is susceptible to a variety of attacks such as malware and viruses. Based on a comprehensive security analysis, we have identified several vulnerabilities that may be exploited by an attacker to compromise the authenticity, privacy, availability, and integrity of the UAF protocol. Although FIDO increases the users' security by abolishing the use of passwords, disclosure attacks can also target the users' personal computer, Thus, we investigate safeguards that can be applied at the software level, either from the operating system or the applications, to zeroize data in the volatile memory. Experimental results showed that Windows kernel zeroizes data after a process termination, while the Linux kernel does not. Moreover, by comparing software functions in C/C++ programming language and built in Windows functions,

we have concluded that only Windows provides a specific function, named SecureZeroMemory, that can reliably zeroize volatile memory data.

Lastly, the third part of this work (see section 4) focuses on proposing novel solutions and methodologies for continuous authentication and detection of malicious actions. The first solution, named gaithashing, is a two-factor authentication that interpolates between the security features of biohash and the recognition capabilities of gait features to provide a high accuracy and secure authentication system. A novel characteristic of gaithashing is that it enrolls three different human silhouettes types. By selecting appropriate weight values, the proposed scheme eliminates the noise and distortions caused by different silhouette types and achieves to authenticate a user independently of his/her silhouette. The second solution focuses on the detection of malicious actions. This has been performed in the context of Ad hoc networks, and one of the simplest yet effective attack that targets the AODV routing protocol. Particularly, a comprehensive analysis of the blackhole attack is conducted focusing not only on the effects of the attack, but also on the exploitation of the route discovery process. As a result, a new critical attack parameter is identified (i.e., blackhole intensity), which quantifies the relation between AODV's sequence number parameter and the performance of blackhole attacks.

2. Password based authentication: A deficient approach

2.1. Background

2.1.1. 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 attack categories: i) Online and ii) Offline. In online attacks, an attacker can try to login to a website by selecting frequently used passwords. After several 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. Subsequently, in this thesis 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 [6], iii) Markov model based [7] and iv) mangling rules [8]. 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 thesis, 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.

2.1.2. Hardware based 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 [9], 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 [10], 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 thesis, we will consider GPUs as the hardware platform of password guessing attacks.

2.1.3. CMS and web application frameworks

Nowadays, the majority of websites originate either from CMS or 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 [11], 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 [11]. 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 [12]. 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

2.1.4. 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 [6] 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) [13]. In [14], the authors analyzed the Rock you [15] 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 [16] 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 [17], 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, [18] 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 [19] 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 [20] show that, even in 2016, the users’ passwords are still predictable, since the most common recorded passwords include “123456”, “qwerty” and “111111”. In [21], 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.

2.2.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 [22], SHA1 [23], SHA256 - SHA512 [24], PBKDF2 [25], BCRYPT [26], SCRYPT [27] and Argon2 [28]. 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 [27] and recently in 2016, the SCRYPT algorithm was published by IETF as a standard (RFC 7914) [29]. It is important to mention that for BCRYPT and SCRYPT, the literature uses the term cost factor [26], [27] instead of iterations (specifically for SCRYPT it is called CPU/Memory cost factor). In the rest of this thesis 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. 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 [30], while the author of SCRYPT recommends 16384 iterations [27]. 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 [31].

As mentioned in section 2.1.2, 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 [32] allow efficient implementations with very little use of RAM, which makes brute-force attacks to PBKDF2 using FPGAs relatively cheap. Also, the work in [33] 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 [34]. 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 [27]. However,

recent works such as [35] [36] 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 [37] in which Argon2 was the winner.

2.3. A mathematical model for cost estimation of password guessing attacks.

In this section we propose 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 2.1.4). 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.

2.3.1. Mathematical 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

2.3.2. Effectiveness: Brute Force password guessing attacks

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 2.3.1, 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 [38]. 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 4, 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 4: Categories and number of leaked passwords

The numerical values of the password analysis are shown in Table 5. 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 5 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%.

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

2.3.3. Cost analysis: Brute Force password guessing attacks

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 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 $\alpha = 1/2$ 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} (see section 2.3.2), 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).

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

2.3.4. Effectiveness: Dictionary password guessing attacks

In this section, we analyze the effectiveness E_{DC} (see section 2.3.1) 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 [18] (see Table 6). 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 6: Effectiveness values for pure dictionary password guessing attacks (values were taken from [18])

Moreover, we have obtained the E_{DC} values based on Markov model and PCFG as derived from [18] (see Table 7). 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 7: Effectiveness values for dictionary password guessing using PCFG or Markov models (values were taken from [18])

2.3.5. Cost analysis: Dictionary password guessing attacks

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 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).

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

2.4. Password hashing scheme evaluation

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 8 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 vii) 11 are classified as "Other", which do not belong to any of the above categories.

Based on the results of Table 8 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 2.5.1, 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 2.2, 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 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 8), 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.

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

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 8), 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. These variations stem 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 8 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 (seeTable 8). 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 2.1.3, 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 9 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 9. 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 9, 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 [45], 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 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.

PHP Frameworks	Hash function	Iterations	Salt	JavaScript	Hash function	Iterations	Salt
Kohana 3.3	SHA256	1	✓ (Constant)	MeteorJS	BCRYPT	1024	✓
Symfony 3.2	No default			ExpressJS	$PBKDF2_{SHA512}$	10000	✓
Laravel 5.4	BCRYPT	1024	✓	SailsJS	BCRYPT	1024	✓
Codeigniter 3.1.4	BCRYPT	1024	✓	FathersJS	BCRYPT	1024	✓
CakePHP 3.3	BCRYPT	1024	✓	Derby	BCRYPT	1024	✓
Zend framework3	BCRYPT	16384	✓	Wakanda	No default		
Yii 2	BCRYPT	8192	✓	MeanJS	$PBKDF2_{SHA512}$	10000	✓
Phalcon 3.0.4	BCRYPT	256	✓	MernJS	$PBKDF2_{SHA512}$	10000	✓
Aura PHP	BCRYPT	1024	✓	nodeJS	$PBKDF2_{SHA512}$	10000	✓
Lithium	BCRYPT	1024	✓	AllcountJS	SHA1	1	✓
Python Frameworks	**Hash function**	**Iterations**	**Salt**	AngularJS	No default		
Django	$PBKDF2_{SHA256}$	30000	✓	**Ruby Frameworks**	**Hash function**	**Iterations**	**Salt**
CherryPy	MD5	1	✓	Ruby on Rails	BCRYPT	1024	✓
Flask	$PBKDF2_{SHA256}$	50000	✓	Padrino	No default		
Bottle	No default			Nyny	No default		
Pyramid	SHA512	1	✗	Grape	No default		
Klein	No default			Nancy	No default		
Web2py	SHA512	1000	✗	Ramaze	No default		
Objectweb	No default			Cuba	SHA1	1	✓
Pecan	No default			Camping	No default		
Tornado	No default			Scorched	No default		
Grok	No default			Celluloid	No default		
Zope	No default			Sinatra	No default		
Turbogears	No default			**Other Frameworks**	**Hash function**	**Iterations**	**Salt**
Quixote	No default			ASP.NET	SHA256	1	✓

Table 9: The default hashing scheme parameters of the investigated web application frameworks

2.5. Cost of password cracking

2.5.1. Hashrates

First, we derive hashrate values using a popular GPU-based password cracking tool named Hashcat [46]. 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 10) of various hash functions and iterations using the GeForce GTX 1070 [47], 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 [48], [49], 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 10). 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.

2.5.2. Comparative analysis

Here we use our cost analysis model that we presented in section 2.3 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 2.5.1 for the hashrates, as well as sections 2.3.2 and 2.3.4 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 2.3.3). 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., $cost\_time_{BF}$) 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

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 10: Hashrates and runtime values

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 are 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 2.3.3 and the input values derived in section 2.3.2 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 8, 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. 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 [26]. As we mentioned in section 2.2, NIST guidelines [30] 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 7). 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 10),

which is lower than the login delay threshold of one second (see section 2.5.1), 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 [27] 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 10). 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.

2.6. 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 an 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 2.2. 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 [27]. 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.*

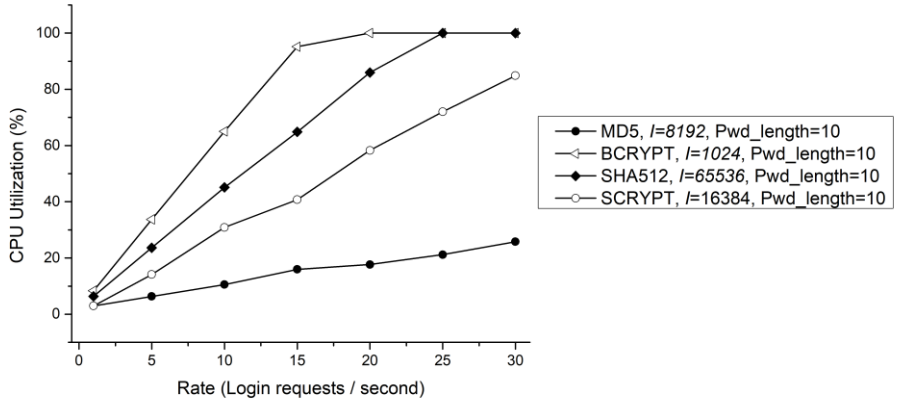

Figure 1: CPU utilization vs login rate

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* [30].

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 10). 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.

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.*

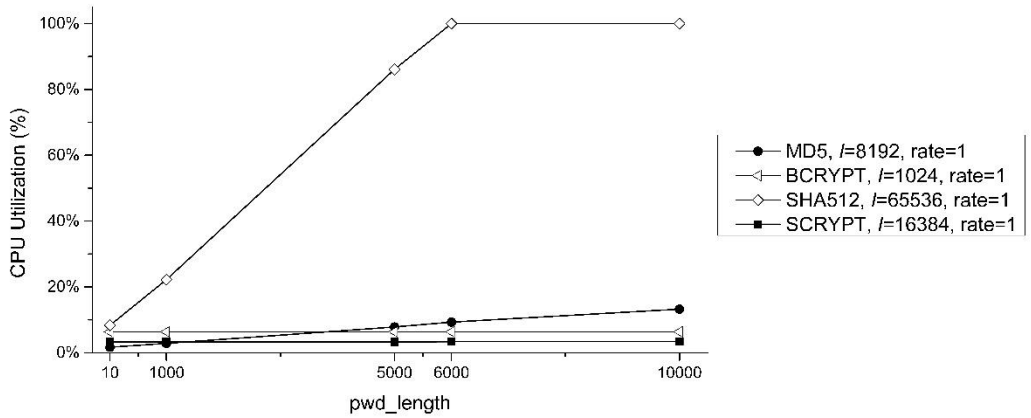

Figure 2: CPU utilization vs password length

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 10), which makes it suitable for interactive logins, due to its small authentication delay. Subsequently, we can conclude to the following observation:

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).*

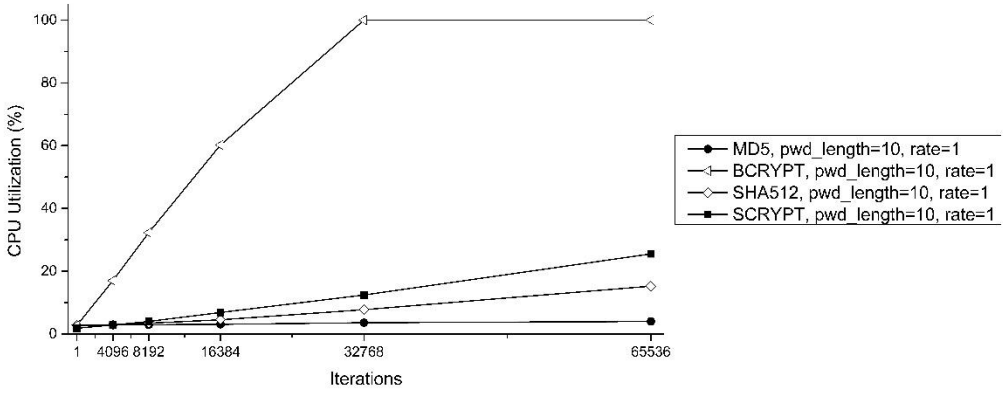

Figure 3: CPU utilization vs iterations

2.7.Recommendations on Password hashing

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 assuming 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].

3. Overcoming the limitation of passwords

3.1. Strong authentication with Fast IDentity Online

3.1.1. Background

3.1.1.1. Related Work

The FIDO security reference [60] outlines a list of assets that must be protected against malicious behavior and provides a limited set of security requirements with the goal of protecting these assets. It is important to point out that these requirements are optional and vendors receiving FIDO certification are not obliged to implement them. A variety of vendors such as Samsung, LG, Qualcomm, and Huawei [62] have already received FIDO certification, however, their implementations are proprietary, and, therefore, not open to 3rd party evaluation. Per FIDO specifications, the critical assets of the UAF protocol are the private key of the authentication key pair, the private key of the UAF authenticator attestation key pair, and the UAF authenticator attestation authority private key [63]. Furthermore, the UAF protocol specifications incorporate the following (optional) security requirements: the authentication keys must be securely stored within a UAF authenticator and thus protected against any misuse, users must authenticate themselves to the UAF authenticator before the authentication keys are accessed, the UAF authenticators may support authenticator attestation using a shared attestation certificate, and a UAF authenticator may implement a secure display mechanism (also referred as transaction confirmation mechanism), which can be used by the UAF client for displaying transaction data to the user. Therefore, the UAF specifications do not incorporate any mechanisms that safeguard the cryptographic material stored in the UAF authenticators or protect against attacks that may target the UAF client. Instead, the responsibility for the design and implementation of any security measures that protect these critical entities is passed on to the vendors.

One solution to address the security requirements of the UAF specifications and provide a secure operational environment for the UAF authenticators, is the incorporation of trusted computing platform technologies [64]. The trusted computing platform constitutes of specialized hardware that provides a variety of services, such as secure input/output, device authentication, integrity measurement, sealed storage, remote attestation, cryptographic acceleration, protected execution, root of trust, and digital rights management. Two prevalent platforms for trusted computing currently exist [64], the Trusted Platform Module (TPM) [65], which is based on the specifications created

by the Trusted Computing Group, and the TrustZone (TZ) platform [66], created by the ARM corporation. The TPM is a co-processor, which provides basic cryptographic capabilities like random number generation, hashing, protected storage of sensitive data (e.g. secret keys), asymmetric encryption, as well as generation of signatures. The TPM platform presents some significant limitations [64]: (i) the need for a separate module increases the cost of a device; (ii) it cannot be deployed on legacy devices; (iii) it does not protect against runtime attacks; (iv) it relies on the assumption that a TPM cannot be tampered; (v) the physical size and energy consumption requirements make it an unsuitable solution for mobile and embedded devices; (vi) in case of a TPM compromise, the hardware module must be physically replaced; and (vii) the supported cryptographic algorithms have been found to pose security concerns (i.e., SHA-1), and are not well suited for resource restricted devices (i.e., RSA).

The TrustZone platform, is part of ARM's processor cores and system on chip (SoC) reference architecture. The associated hardware is part of the SoC silicon, and thus, it does not require any additional hardware. The primary objective of TrustZone is to establish a hardware-enforced security environment providing code isolation, that is, a clear separation between trusted software, which is granted access to sensitive data like secret keys, and other parts of the embedded software. To achieve this, the TrustZone platform provides two virtual processing cores with different privileges and a strictly controlled communication interface, enabling the creation of two distinct execution environments, encapsulated by hardware. Nevertheless, to the best of our knowledge, Samsung is the only certified vendor that implements a UAF authenticator using the TrustZone platform [67]. Furthermore, this approach only protects the UAF authenticator, while the UAF client is still susceptible to a variety of attacks. Finally, extensive literature has shown that the TrustZone platform itself is not immune to weakness and vulnerabilities [68] [69] [70] [71].

3.1.1.2. FIDO UAF protocol operations

The UAF protocol (see Figure 5) encompasses three major operations, namely, registration, authentication, and deregistration. During the registration operation, the UAF protocol allows a user to register to a relying party using one or more UAF authenticators. Once registration is complete, the user can then invoke the authentication operation, in which the relying party prompts for a user authentication using the UAF authenticator previously used during the registration operation. Finally,

in the deregistration operation, the relying party can trigger the deletion of the authentication key material and remove the user from its list of authenticated users.

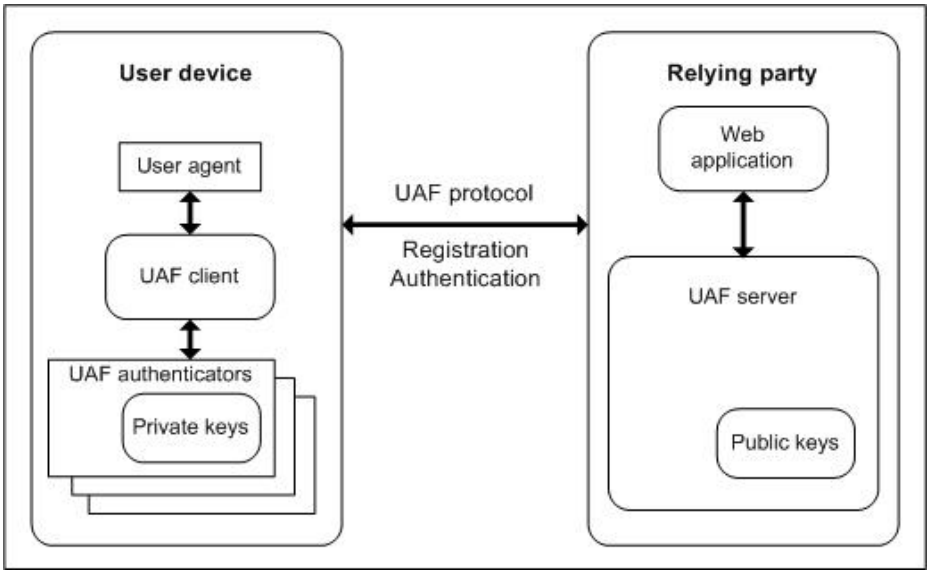

Figure 5: The FIDO UAF protocol

The UAF registration operation. The registration operation is initiated when a user requests a registration to a relying party, either through a compatible application or through a browser. The relying party replies to the registration request by transmitting a registration message with the following parameters: the AppID, the authenticator policy, the server generated challenge, and the username to the UAF client residing in the user's device (illustrated in Figure 6). The AppID parameter is used by the UAF client to determine if the calling application (or website) is authorized to use the UAF protocol and it is associated with a key pair by the UAF authenticator (during key generation), so that access to the generated key pair is limited to its respective application. The authenticator policy lists the type of UAF authenticators required by the relying party, while the server generated challenge is a random nonce value used to protect against replay attacks. Finally, the username parameter is used by the UAF authenticator to distinguish key pairs that belong to the same application (or website), but to different users.

Once the UAF client receives the registration message from the relying party, it first identifies the calling app (or website) and then determines (based on the AppID parameter) whether the associated application is trusted and allowed to proceed with a registration request. To accomplish this, the UAF client queries the relying party for the trusted facet list (i.e., a list of all the approved entities related to the calling app)

and, based on this list, decides whether registration will proceed or not. For example, if the registration request was initiated by an application, then the trusted facet list will contain a signature of the calling application that the UAF client can use to verify the app. If, on the other hand, the registration was initiated by a website, then the trusted facet list will contain all the associated and approved domain names. Subsequently, the UAF client will check the authenticator policy parameter and generate a key registration request to the set of UAF authenticator(s) mandatory by the policy. If the required UAF authenticators are not present in the user's device, then the registration operation will be canceled.

The UAF client communicates with the UAF authenticator(s) using the authenticator specific module (ASM), a software associated with a UAF authenticator that provides a uniform interface between the hardware and the UAF client software. At this stage, the UAF client performs the following operations: it first calls the UAF authenticator in order to compute the final challenge parameter (FCP), which is a hash of the AppID and the server challenge. Then, it generates the KHAccessToken, which is an access control mechanism for protecting an authenticator's UAF credentials from unauthorized use. It is created by ASM by mixing various sources of information together. Typically, KHAccessToken contains the following four data items: AppID, PersonaID, ASMToken and CallerID. The AppID is provided by the relying party and it is contained within every UAF message. The PersonaID is obtained by ASM from the operating system, and, typically, a different PersonaID is assigned to every user account. The ASMToken is a random generated secret which is maintained and protected by ASM. In a typical implementation ASM will randomly generate an ASMToken when it is first executed and will store this secret until it is uninstalled. CallerID is the calling UAF client's platform assigned ID. Once the FCP and the KHAccessToken are computed, the UAF client will send the key registration request to the UAF authenticator including the FCP, the KHAccessToken, and the username parameter.

Following the reception of a key registration request by a UAF authenticator, the later will first prompt the user for authentication, and, then, generate a new key pair (Uauth.pub, Uauth.priv), store it on its secure storage, and associate it with the received username and KHAccessToken. Subsequently, the UAF authenticator will create the key registration data (KRD) object containing the FCP, the newly generated user public

key (Uauth.pub), and the authenticator's attestation ID (AAID), which is a unique identifier assigned to a model, class or batch of UAF authenticators, and it is used by the relying party to identify a UAF authenticator and attest its legitimacy. Once the KRD is generated, the UAF authenticator will sign it using its attestation private key and return to the UAF client a key registration reply (which the later forwards to the relying party) that encompasses: the signed KRD, the AAID, Uauth.pub, and its attestation certificate (Certattest). Upon the reception of the key registration reply by the relying party, the later cryptographically verifies the KRD object, uses the AAID to identify if the UAF authenticator is a legitimate authenticator with a valid (i.e., unrevoked) attestation certificate, and, finally, stores the Uauth.pub key in a database for the purposes of user authentication in any subsequent authentication requests.

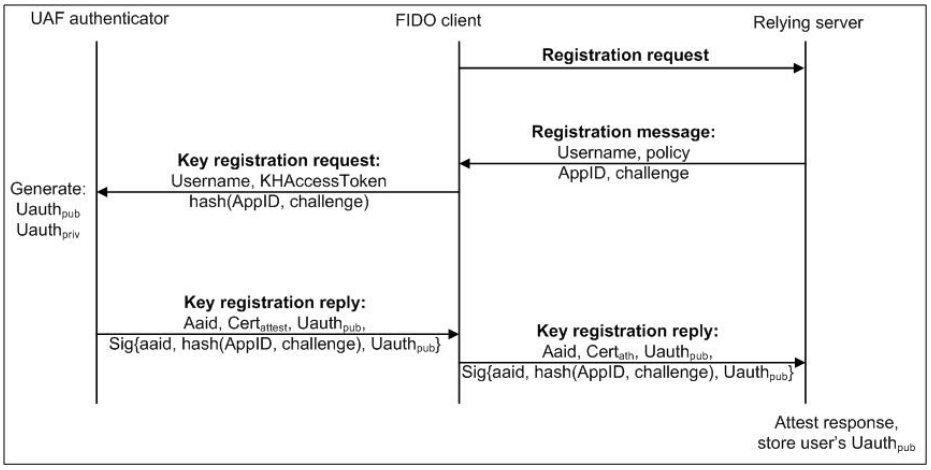

Figure 6: The UAF registration operation

The UAF authentication operation. The authentication operation (illustrated in Figure 7) is initiated when a user requests a service that requires authentication to a relying party, either through a compatible application or through a browser (in a similar fashion with the registration operation outlined above). The relying party replies to the authentication request by transmitting an authentication message with the following parameters: the AppID, the authenticator policy, and a server generated challenge, to the UAF client residing in the user's device. The UAF client receiving the authentication request, first identifies the calling app (or website) and then determines (based on the AppID parameter) whether the associated application is trusted and allowed to proceed with the authentication request. Subsequently, the UAF client checks the authenticator policy parameter and sends a key authentication request to the set of UAF authenticator(s) mandatory by the policy. If the required UAF authenticators

are not present in the user's device, then the authentication operation will be canceled. Using ASM, the UAF client performs the following operations: it first calls the UAF authenticator in order to compute the FCP, which is a hash of the AppID and the server challenge. Then, it retrieves the KHAccessToken, and finally, sends the key authentication request to the UAF authenticator(s) including the FCP and the KHAccessToken.

Following the reception of a key authentication request by a UAF authenticator, the later will first check if the UAF client is authorized to request an authentication for that particular user key, based on KHAccessToken. If the UAF client is authorized, then the UAF authenticator will prompt the user for authentication, and, then, retrieve the associated Uauth.priv from its secure key storage. Subsequently, the UAF authenticator will create the SignedData object containing the FCP, a newly generated nonce, and a Sign Counter (cntr). The cntr variable is a monotonically increasing counter, incremented on every sign request performed by the UAF authenticator for a particular user key pair. This value is then used by the relaying party to detect cloned authenticators. Once the SignedData object is generated, the UAF authenticator will sign it using the Uauth.priv key and return to the UAF client a key authentication reply (which the later forwards to the relying party) that encompasses: the signed object SignedData, the FCP, the nonce n, and the counter cntr. Finally, upon the reception of the key authentication reply by the relying party, the later first retrieves Uauth.pub from its database, cryptographically verifies the signedData object, and stores the value of the cntr counter. If the verification of the SignedData object succeeds, then the user is successfully authenticated.

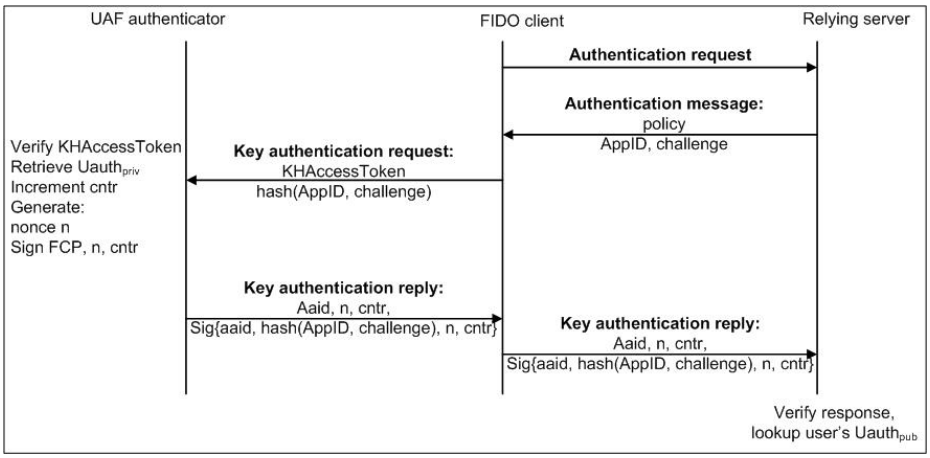

Figure 7: The UAF authentication operation

3.1.2. Security analysis

UAF authenticator vulnerabilities. The first and most apparent attack vector of the UAF protocol is the authentication keys. Therefore, an attacker may attempt to (directly or indirectly) gain unprivileged access to these keys. As we previously mentioned, the responsibility of storing the authentication keys lies with the UAF authenticator and based on the UAF protocol security requirements, the UAF authenticator utilises some form of secure/privileged storage. However, it has been shown in the literature that such types of key storage solutions can still be compromised [72]. UAF authenticators typically rely on trusted computing platforms for the storage of cryptographic material. Cooijmans et al [69] have shown that on several widely adopted trusted computing platforms, an attacker with privileged rights can gain the ability of using encrypted credentials by moving them to a different directory, which designates a malicious application as the owner of the credentials. Finally, an attacker may also attempt to indirectly gain access to the authentication keys, by fully compromising the UAF authenticator(s). Based on the literature, an attacker can gain full access to a trusted computing platform by performing an integrated circuit attack (i.e., ICA) [68]. One limitation of this attack is the requirement to have physical access to the user's device. However, once the attack is performed, the attacker can then create a cloned UAF authenticator, alleviating any further need for the original user's device.

When utilizing a cloned UAF authenticator, an attacker must then evade the security mechanisms of the UAF protocol, implemented on the purpose of identifying such malicious behavior. Recall that the UAF protocol incorporates two security mechanisms that safeguard the operation of the UAF authenticator: (i) an attestation mechanism, in which the UAF authenticator must prove its legitimacy by providing an attestation signature during the registration process and (ii) a sign counter (cntr) mechanism, which is a monotonically increasing counter, incremented on every sign request performed by the UAF authenticator for a particular user key pair and used by the relaying party to detect cloned UAF authenticators.

Regarding the attestation mechanism, we have identified three approaches that can be used by an attacker to circumvent detection. In the first method, an attacker may utilize the extracted attestation key from the compromised UAF authenticator and perform registration requests to relying parties, impersonating the legitimate user. Since the attestation keys for each UAF authenticator are not unique (i.e., a group of UAF

authenticators share the same attestation key pair), the malicious behavior cannot be easily detected by the relying party. If, however, the attestation keys are revoked by the device's vendor, then there is a risk of detection by the relying party. A second method that can be used by an attacker when employing a cloned authenticator is to avoid the attestation mechanism all together. This can be achieved by exploiting a limitation in the attestation process. Recall that the attestation process takes place only during the registration operation. Therefore, an attacker may allow the legitimate UAF authenticator to perform the registration process, and, subsequently, without the users' knowledge, use the cloned authenticator to authenticate itself to the relying party, masquerading as the legitimate user. Finally, an attacker may use the cloned UAF authenticator temporarily to collect personal information related to the legitimate user, and, then, register at other relying parties using a different, non-cloned UAF authenticator. Subsequently, since the attestation procedure takes place at a non-cloned authenticator, there is no risk of revocation, while the attacker retains the ability to impersonate the legitimate user to any relying party.

On the other hand, the second security measure proposed by the UAF specifications (i.e., sign counter), can be circumvented by an attacker, if the later actively attempts to perform an authentication operation immediately after the completion of cloning a UAF authenticator. Recall from that during the authentication operation, a relying party will assume a UAF authenticator is legitimate if the sign counter encapsulated in the key authentication reply is equal to the sign counter maintained by the relying party incremented by one. Therefore, a race condition evolves between the legitimate and the cloned UAF authenticator, since only the UAF authenticator that manages to perform an authentication request first, will be considered legitimate by the relying party (while the second authenticator will attempt to authenticate using an older value of the sign counter). Thus, an attacker can circumvent this security measure by performing an authentication request to the relying party as soon as the UAF authenticator is cloned, maximizing his chances of winning the race condition.

UAF client vulnerabilities. The second critical entity of the UAF protocol that resides at a user's device is the UAF client. Recall that the UAF client acts as an intermediator between the relying party on one hand and the UAF authenticator on the other and it is responsible for most of UAF's protocol operations, short of generating the encryption keys or performing cryptographic operations. Furthermore, the UAF client is

implemented entirely in software, making it an ideal candidate for software attacks. Even more importantly, the UAF protocol does not incorporate any security measures that safeguard the UAF client from attacks or verifies that a user's device operates a legitimate version of the client. The UAF protocol specifications propose the execution of the UAF client in a "privileged" environment, however, since the client is typically embedded within a browser either fully or as a plug-in, it is de-facto implemented as a normal application.

The simplest method of delivering a malicious UAF client to a user's device is by deceiving the user to install the application voluntarily. Common delivery methods include attachments in e-mails or browsing a malicious website that installs software after the user clicks on a pop-up. Other methods of compromising a UAF client is through malicious software residing at the user's device (such as a virus, worm. trojan, or root kit) or by exploiting an operating system vulnerability. The latter enables the execution of a plethora of attacks such as spoofing of inter-process communication, privilege escalation, return-oriented programming, or code injection attacks. For example, in a variety of sources such as [73] [74] [75], the authors demonstrate methodologies for accomplishing privilege escalation in the android operating system, one of the most widely used platforms, which includes a variety of privilege protection mechanisms, such as application specific sandboxing and Mandatory Access Control (MAC) policies. Furthermore, in the most recent versions of android, privilege escalation is typically achieved using system less root [74], which is the process of gaining escalated privileges without any modification to the system partition, thus evading detection by any security mechanisms that validate an operation system through a checksum of its system partition (i.e., a common security mechanism used by most of the trusted computing platforms).

3.1.3. Threat analysis

Critical assets related to the UAF protocols' secure operation. The UAF specifications [76] provide a limited list of assets that must be protected in an implementation of the UAF protocol. These assets include the private key of the authentication key pair, the private key of the UAF authenticator attestation key pair, and the UAF authenticator attestation authority private key. However, an attacker may also target several other assets that are either part of the UAF protocol, or they are integral in its secure operation. In particular, an attacker may either target the UAF

authenticator(s) or the UAF client that are present in a legitimate users' device. Furthermore, an attacker may indirectly compromise the secure operation of the UAF protocol by exploiting existing vulnerabilities (i) at the underlying operating system in which the UAF protocol is executed, or (ii) at the trusted computing platform (typically the TrustZone platform), used for the hardware-assisted protection of the encryption keys and the operation of the UAF authenticator(s).

Threat evaluation. Based on the security analysis, the private keys stored in the UAF authenticator, namely the attestation private key and the authentication private keys pose a critical attack vector of the UAF protocol. Recall from that these keys are used by the UAF authenticator to sign registration and authentication replies, respectively. On the other hand, the relying party uses these signed replies to authenticate the UAF authenticator and verify its legitimacy. Therefore, if an attacker compromises the attestation private key, he would then be capable of impersonating the legitimate user by registering to other relying parties on the users' behalf, without the latter's consent (including fraudulent relaying parties). In order to have access to the authentication keys associated with the malicious registrations and to avoid detection by the user, the attacker will have to import the attestation private key to a cloned and silent authenticator, i.e., an authenticator that appears to have been manufactured by the same vendor as the legitimate one and does not prompt the user for any action during the registration and authentication operations of the UAF protocol. On the other hand, if the attacker compromises one or more authentication private keys, he would then be capable of impersonating the legitimate user by authenticating as the user to relying parties. The attacker is limited, however, to relying parties that the legitimate user has already registered. Nevertheless, once authenticated, the attacker can then collect personal data related to the legitimate user and stored at the relying party, as well as perform transactions with the relaying party without the users' consent.

An attacker may also attempt to indirectly gain access to the attestation and authentication keys, by fully compromising the UAF authenticator(s) residing at the device of a legitimate user. This can be accomplished in the following ways: the user unwillingly installs a malicious authenticator to his/her device, the attacker compromises the UAF authenticator by targeting the UAF authenticators' underlying trusted computing platform, and, the attacker gains physical access to the device and either installs a malicious authenticator, or tampers with the legitimate UAF

authenticator(s) installed on the device. As a result, any subsequent registration and authentication requests will be captured by the malicious authenticator, enabling the attacker to impersonate the legitimate user, collect personal data, and perform transactions on the users' behalf, similarly to the cloned authenticator threat we analyzed previously. Furthermore, the attacker can also extract the attestation and authentication keys, to create a cloned authenticator that resides outside the device of the user.

The UAF client signifies another critical attack vector identified in the security evaluation. An attacker may attempt to compromise the UAF client by exploiting one or more of the following vulnerabilities: gaining physical access to the user's device and manually installing a malicious client, deceiving the user to install the malicious client voluntarily, using other malicious software residing at the user's device (such as a virus, worm. trojan, or root kit) to install the malicious client, or by exploiting an operating system vulnerability. Having successfully compromised the UAF client, an attacker is then capable of launching several additional attacks against the UAF protocol, such as: allowing itself or other malicious applications to perform registration/authentication operations without the user's consent, enforce the use of the weakest/less secure UAF authenticator during a legitimate registration process, direct a user to a fake or malicious relying party, and defeat the user consent, transaction confirmation, and trusted facet list security measures of the UAF protocol. During the registration operation, the UAF client is responsible for initiating registration requests, determining if applications (or websites) are authorized to use the UAF protocol, present a UI to the user, and directing the relying party challenge to the UAF authenticator based on the authenticator policy transmitted by the relying party (i.e., based on the trusted facet list). Since the UAF client is the only entity responsible for assessing the trusted facet list, it can allow the registration operation for any website, or from any application, regardless of what is enforced by the trusted facet list security measure. Therefore, the user may unwillingly be redirected to a malicious relying party masqueraded as a legitimate one, so that personal/valuable information can be phished by an attacker. Furthermore, as we mentioned previously, it is the UAF client's responsibility for presenting a UI to the user, and, therefore, even if the user's device incorporates a transaction confirmation security mechanism, the confirmation will always be true, since the mechanism validates if the information provided to the user is

tampered/modified/spoofed after leaving the UAF client, and not if the later modified the displayed content. Finally, a malicious UAF client may forward a relying party challenge to the weakest UAF authenticator (preferably one with a low entropy secret). Subsequently, during authentication, the attacker could attempt to discover the secret and access the user's account without the legitimate users' consent.

Asset	Threat	Consequences
Attestation private key	Attacker gains access to the attestation keys	Impersonate user, create a clone authenticator
Authentication private key	Attacker gains access to the authentication keys	Impersonate user, capture user data
UAF authenticator	User installs a malicious authenticator	Impersonate user, capture user data, register the user to a fraudulent replaying party
TrustZone, UAF authenticator	Attacker compromises the trusted computing platform	Create cloned authenticator, impersonate user, compromise the UAF authenticator
UAF client, UAF authenticator, TrustZone	Attacker gains physical access to a user's device	Create cloned authenticator, impersonate user, compromise the UAF authenticator, install malicious UAF client
UAF authenticator	Attacker employs a cloned authenticator	Impersonate user, capture user data, register the user to a fraudulent relaying party
UAF client	User installs a malicious client	Register to a fraudulent relaying party, phishing – lead to malicious websites, downgrade authentication policy, capture user data, circumvent transaction confirmation security mechanism, allow malicious apps to register/impersonate the user
Operating system	Attacker can execute privileged code at the user's device	Compromise the UAF client

Table 13: Threats related to the UAF protocol and their associated consequences

3.1.4. Results and discussion

The UAF protocol provides several important advantages over traditional authentication mechanisms, such as strong authentication and a simplified registration and authentication procedure. However, the UAF protocol also transfers user authentication operations from the server-side to the client-side. Therefore, the critical functionality of the UAF protocol typically operates in a consumer platform such as a mobile device, which is susceptible to a variety of attacks such as malware and viruses, its users deploy unsupervised software, and the deployed operating systems may be susceptible to several vulnerabilities. As a part of this thesis, we have provided a comprehensive security analysis of the UAF protocol and have identified several

vulnerabilities that may be exploited by an attacker to compromise the authenticity, privacy, availability, and integrity of the UAF protocol. More specifically, we have investigated methods of attacking the two entities of the UAF protocol residing at a user's device, namely, the UAF authenticator and the UAF client, including the ability of an attacker to gain unprivileged access to the cryptographic material stored within the UAF authenticator and highjack either the of these two entities. Furthermore, we have investigated and identified how an attacker can circumvent the security measures provided by the UAF protocol, including the authenticator attestation mechanism, the transaction confirmation mechanism, the trusted facet list, and the sign counter.

3.2. Real-time protection of user authentication credentials

3.2.1. Related work

Regarding the retrieval of sensitive information in the volatile memory, Darren et al. tried to recover data remnants from cloud storage applications including Dropbox [77], Skydrive [78], and Google Drive [79]. Similarly, in [80] the authors investigate the volatile memory of cloud services applications, such as Amazon S3, Dropbox, Google Docs and Evernote. In all the aforementioned publications, several artifacts were recovered such as authentication credentials, visited URLs, filenames and hashes. Apart from personal computers, sensitive information was also recovered from the volatile memory of Android devices using two different methods. More specifically, in the first method [81] the authors used the Linux Memory Extractor (LiME) kernel module [82] and a physical Samsung i9000 phone to dump the Android memory, whereas in the second technique [83] the Android emulator was used alongside with Dalvik Debug Monitor Server (DDMS) to acquire the memory data. In both cases, critical and secure applications, such as mobile banking and password managers, were examined and authentication credentials were recovered in plain text from the dumped memory.

Regarding memory encryption, the proposed solutions can be further classified into two categories: software- based and hardware-based. For software-based solutions, in [84], the authors propose a modified secure memory bus controlled by the OS, in which the encryption key is generated each time the system boots up. Peterson, in [85], modified the virtual memory manager of the Linux 2.6.24 kernel and partitioned the volatile memory into a plaintext and an encrypted segment. However, [86] shows that the memory maps, should be maintained in the plaintext segment; thus pointing

the addresses to where the encrypted volatile data are stored. The second category of the proposed solutions for memory encryption is based on hardware modifications. In particular, several publications [87] [88] [89] [90] [91] [92] [93] for single processor systems propose the addition of an encryption unit to cipher and decipher data from and to the volatile memory. Moreover, for multi-processor systems, [94] proposes a shared bus, containing a crypto engine, to coordinate and secure traffic between processors, while [95] [96] proposed the use of sequence numbers for the coordination between different processors. Lastly, in [97], the authors propose SecBus, a cryptographic coprocessor between the volatile memory and the main processor.

The main limitation of the proposed memory encryption solutions has to do with the fact that hardware-based solutions require extensive changes in the current computer architecture, while the software-based solutions require modifications at the OS kernel. In contrast to the relevant works, in this thesis we investigate if the latest OS versions (Windows and Linux) provide built-in data zeroization methods as well as whether C/C++ developers can use existing software libraries and methods in order to perform data zeroization in their applications.

3.2.2. Software level protection

3.2.2.1. Operating System level protection

Memory management is the procedure of administering the volatile memory at the system level. This is performed by the kernel of the Operating System (OS) with the support of a part of the central processing unit, named memory management unit. Allocation and deallocation requests are used in order to grant or revoke memory blocks to applications. Allocation is the procedure in which memory blocks are granted to applications and are then used by them for handling the necessary data for their functionalities. On the other hand, deallocation is the procedure in which the applications free the memory blocks they do not longer need, making them available for other running or starting applications. It is important to note that the OS does not modify the allocated memory blocks, since this action could cause the running applications to crash. Subsequently, during the applications' runtime, only the applications themselves are accountable of modifying their allocated memory blocks.

In order to find out whether the OS performs data zeroization, we developed a testing application written in C programming language (see Figure 8), that holds a secret value in a variable named as *password*. The aim of the experiments was to investigate

how many instances of the *password* variable can be extracted from the volatile memory. More specifically, as shown in Figure 8, the testing application defines the *password* variable at line 3, which is an array of type *char* and size *length*. Moreover, the *stdin* (e.g. keyboard input) is used to fill in the array of the *password* variable. For the experiments, three types of memory dumps were considered which are: A) **Process:** This memory dump includes only the memory blocks that are allocated to the executable of Figure 8. B) **All- Processes:** This memory dump includes memory blocks allocated to all running user-mode processes in the OS. In this way we can find out whether the password variable of Figure 8 can be extracted from other user-mode running processes; C) **System:** This memory dump contains the entire volatile memory including memory allocated not only to user-mode processes but also to the OS kernel, drivers, unallocated blocks. The technical methodology that we followed in order to obtain the memory dumps is as follows. To perform a **Process** dump in Linux, the GNU debugger (i.e. GDB) was used to dump the memory blocks of a process based on its PID. Similarly, the **All-processes** dump was performed using a script that feeds GDB with all the running PIDs. The same methodology was followed in Windows. In particular, we used the Windows Powershell in order to list all the running PIDs and feed them to ProcDump [98] (i.e., a Windows utility which performs memory dumps of running processes). It is important to note that all the aforementioned memory dumps, were executed using root privileges both in Linux and Windows. To perform **System** dump, we used virtual machines, in order to dump the entire volatile memory of the system in an easy manner.

First testing application

```
01: void main() {
02:   static int length;
03:   char password[length];
04:   fgets(password, length * sizeof(char), stdin);
05:   sleep(120);
06: } //suspend for 120 seconds
```

Figure 8: First testing application used to discover the total number of instances of the password variable in the volatile memory

Moreover, two scenarios were considered. In the first scenario named as "***Running process***" we performed memory dumps (all three types) while the process of the executable was running. This was achieved during the sleep function (see line 5 of Figure 8), where the execution of the process was suspended, and we were able to

recover the memory dump. In the second scenario named as "***After termination***", we performed the

Memory Dump	Operating System			
	Ubuntu Linux		***Windows 7/10***	
	Running Process	***After termination***	***Running Process***	***After termination***
Process	1	Not Applicable	3	Not Applicable
All-Processes	1	1	3	0
System	9	2	5	0

Table 14: Number of instances of the password variable

memory dumps immediately after the termination of the executable. Evidently, in this scenario, we performed only **All- processes** and **System** memory dumps, since **Process** dump cannot be performed after the termination of the executable. The experiments were conducted in Windows 7 and 10 and Ubuntu Linux 14.04, fully updated as of 15th of April 2016. In both versions of Windows, the compiler of Microsoft Visual Studio 2015 suite was used, while in Ubuntu Linux we used the latest version of the GCC compiler (i.e., v5.3).

The results of the experiments are summarized in Table 14. We can observe that in the "***Running process***" scenario in all three memory dump types for both Linux and Windows OS we were able to recover the value of the *password* variable. It is interesting to notice that in the **All-processes** memory dump type, the number of the instances of the *password* variable were the same as in the **Process** memory dump type (i.e., 1 time in Linux and 3 times in Windows). This means that apart from the process itself of the testing application (see Figure 8), the other processes running in the system did not use the *password* variable. We can also observe that in the **System** memory dump, the number of recovered *password* instances increased (i.e., 9 times in Linux and 5 times in Windows). This result means that i) apart from the process of the testing application itself, the OS kernel stores also the value of the *password* variable and ii) the OS kernels stores in multiple memory regions the value of the *password*.

Regarding the results of the "***After termination***" scenario, we can observe an interesting outcome: for both ***All-processes*** and ***System*** dumps in Linux we were able of recovering the *password* variable (1 and 2 times respectively). On the other hand, in Windows we were not able to recover it. This result means that Windows kernel

zeroize the deallocated blocks of a process immediately after its termination. On the other hand, the Linux kernel follows a different approach. That is, instead of zeroizing the deallocated memory blocks of a terminating process, it zeroizes the memory blocks right before their allocation [99]. Thus, in Linux, a malicious software that has access to the entire system memory can extract potentially sensitive information (such as authentication credentials) even from applications that were terminated, in case the related deallocated blocks have not been allocated to a new process. On the contrary, in Windows, a malicious software can extract information only from the memory blocks of running applications.

The above observation implies that Windows is more secure than Linux to memory disclosure attacks. To overcome this issue, we have identified that there is a Linux kernel patch, named as GRsecurity, which provide several security enhancements for the Linux kernel [100]. One of these enhancements enables the Linux kernel to zeroize the deallocated memory blocks after process termination by compiling the Linux kernel with the *PAX_MEMORY_SANITIZE* option that the GRsecurity provides. To this end, we repeated the experiments (using the testing application of Figure 8) in Ubuntu 14.04 compiled with a kernel that has GRSecurity installed and the *PAX_MEMORY_SANITIZE* option enabled. We observed that this time we were not able of recovering instances of the *password* variable after the process termination. Based on the above discussion, we *propose the use of GRsecurity (with the PAX_MEMORY_SANITIZE option enabled),* in order to minimize information disclosure in volatile memory.

Despite the fact that GRsecurity may enable the kernel to perform data zeroization, it is not widely adopted in Linux Distributions. Even those that offer a GRsecurity patched kernel by default, many of them have not enabled the PAX_MEMORY_SANITIZE option. In total, we found six Linux distributions [101] [102] [103] [104] [105] [106] that come with a GRsecuity patched kernel and only three of them have the *PAX_MEMORY_SANITIZE* option enabled.

3.2.2.2. Source code level protection

The previous results show that OS zeroize data only after the termination of the running process which means that during the runtime of a process, sensitive information can be extracted in its allocated memory blocks. In this section, we investigate functions and methods that developers can use in order to zeroize memory

blocks during the runtime of their applications. We focus on C/C++ programming language, since it provides low-level memory manipulation. All experiments carried out in this section perform **Process** dump in a "**Running-process**" scenario.

First, we investigate for Windows OS, if there are special functions that can be used in order to zeroize data. More specifically, by including the *windows.h* header file in a C/C++ source code, a developer has the ability of using the macro *SecureZeroMemory*, which calls the function *RtlSecureZeroMemory* that guarantees to zeroize memory blocks, even if it is not subsequently written or accessed by the code [107]. We repeated the experiments performed in the previous section (i.e., as mentioned previously only **Process** dump in the "**Running process**" scenario) using the same testing application with the difference that at the end of the code we called the SecureZeroMemory macro. We observed that indeed the macro *SecureZeroMemoy* replaced the contents of the password variable with zeroes. Thus, in Windows, developers should use the macro SecureZeroMemory to ensure that the memory blocks of their applications are zeroized.

On the other hand, for Linux OS, there is no similar C function that can be used to zeroize data in the volatile memory. To this end, we have used the function *memset* of the C programming language to manually try to zeroize memory blocks allocated to a process. In particular, we have used the testing application of Figure 9, which is identical to the code of Figure 8, with the difference that Figure 9 includes in line 5, the command *memset(password, '0', length).* This command writes in the memory block, which is allocated for the value of the *password* variable, the 0 character as many times as indicated by the value of the *length* variable. This will result in the zeroization of the data of the array *password.* We repeated the experiments of the previous section and we observed that the *memset* function was not operating as we expected, since the value of the password variable was detected in the process dumps. After investigation, we identified that the *memset* function was not being called due to code optimization. The latter is the process in which a compiler tries to improve the generated executable code by making it consume fewer resources, such as CPU and Memory. This is performed by several techniques. One of these methods is to avoid compiling specific code which is not necessary for the execution flow. For this reason, in our experiments, the compiler skipped the calling of the *memset* function, because the new value of the *password* variable (i.e., the zeroized data) is not used

after the *memset* function. Note that although the executable of Figure 9 was compiled using GCC without optimization flags, the GCC compiler did perform optimization and did not include the *memset* function in the executable.

Second testing application

```
01: void main() {
02:   static int length;
03:   char password[length];
04:   fgets(password, length * sizeof(char), stdin);
05:   memset(password, '0', length);
06:   sleep(120); //suspend for 120 seconds
07: }
```

Figure 9: Second testing application used to discover the total number of instances of the passwrod variable ni the volatile memory

The above results raise the following question: "is it feasible to avoid optimization caused by the GCC compiler, in order to ensure that the *memset* function will be executed"? To answer this question we tried two different methods. In the first method we used the function *memset_s*. The latter has the same functionality as *memset*. The main difference between those two functions is that the *memset_s* cannot be optimized out by the compilers [108]. However, *memset_s* is included only in the currently last version of the standard of the C programming language (i.e., C11 [109]) in Annex K. Unfortunately, Annex K is not mandatory in C11, while GCC compiler (i.e., v5.3) has not implemented the Annex K, and thus the developers have no way to use the *memset_s* function.

The second method that we attempted in order to avoid bypassing optimization was to write a testing application similar to the one described in [110] (see Figure 10), which uses a function pointer of type volatile named *memset_volatile*, as defined at line 1. The declaration of a variable as volatile instructs the compiler not to optimize out functions that access the variable. This is due to the fact the volatile type is used mainly for buffers in communication with hardware devices or other applications. Based on this observation, we defined the function pointer named *memset_volatile* pointing to the function *memset* at line 1. At line 4, a pointer named *password_heap* is defined, which points to a block of memory of size *length*sizeof(char)*. This block of memory is allocated using the *malloc* function, which is used for dynamic memory allocation during the application execution. In line 5, the user enters his password, and in line 6, the memory block allocated at line 4 is freed with the *free* command. It should be noted that the *free* command does not zeroize the data of the memory

block it deallocates. Consequently, we used the *memset_volatile* function pointer to indirectly call the *memset* function. We repeated the experiments once again, using all the available optimization flags of the GCC compiler. In all cases we observed that the GCC compiler did not optimize the call to the memset function. Although the experiments showed that the data type volatile in C/C++ programing language prevents the optimization caused by the compilers, it should be noted that GCC compiler can arbitrary perform optimization even in volatile data types as mentioned in [111]. In any case, volatile function pointers can be used to increase the chances that the *memset* function will not be optimized out during compilation.

Second testing application

```
01: void *(*volatile memset_volatile)(void *, int, size_t) = memset;
02: void sensitive_function() {
03:   static int length;
04:   char *password_heap = malloc(length * sizeof(char));
05:   fgets(password_heap, n, stdin);
06:   memset_volatile(password_heap, 0, n * sizeof(char));
07:   free(password)
08:   sleep(120); //suspend for 120 seconds
09: }
```

Figure 10: Third testing application used to discover the total number of instances of the passwrod variable in the volatile memory

3.2.3. Results and discussion

The real-time user security is significant, as authentication credentials can be stolen in real-time. Therefore, this thesis investigates security measures that can be applied at the OS and the source code level to protect sensitive information in volatile memory from disclosure attacks. Based on the experimental analysis, it was observed that Windows delete the data from deallocated memory blocks, while Linux does not. This can be solved using the GRsecurity Linux kernel patch that enables the zeroization of deallocated memory blocks, using the *PAX_MEMORY_SANITIZE* option during the kernel compilation. At the source code level, the Windows developers may use the SecureZeroMemory function for manually modifying volatile memory data without facing any optimization issues. In Linux, we propose the use of *volatile* function pointers to ensure that the call to *memset* will not be optimized out. Lastly, the experiments performed in web browsers show that in most cases it was feasible to recover user authentication credentials from all the web browsers except when the user has closed the tab that used to access the website.

4. Continuous authentication and detection of malicious actions

4.1. Continuous authentication using biometric modalities

4.1.1. Security and performance of Biometric based authentication

Protection schemes for biometric templates can be categorized as follows: a) biometric cryptosystems, and b) cancelable biometrics. Biometric cryptosystems are designed to securely bind a key to a biometric feature or generate a key from a biometric feature. On the other hand, cancelable biometrics consists of intentional, repeatable distortions of biometric features, based on one-way transforms, where the comparison of biometric templates takes place in the transformed domain. A comprehensive overview of biometric template protection schemes is presented in [112]. One of the most widely used cancellable biometrics algorithm is biohash and its variations [113], [114]. The one-way transformation of biohash is based on random projections [115]. The mathematical properties of random projections ensure the security of the protected template, while at the same time the authentication performance is not deteriorated. For this reason, the proposed scheme adopts a simple variation of biohash to secure the extracted gait features.

As mentioned previously, biometric systems include two procedures: a) enrollment and b) authentication. During enrollment, biometric features are extracted from a user of the system to form its biometric template, which is stored in a database or token. During authentication, the system extracts the considered biometric features of a user and creates a new biometric template, which is compared against the enrolled one for user's acceptance or rejection. Due to the intrinsic noise of biometric features, the authentication and enrollment template cannot perfectly match. For this reason, biometrics systems compare the distance ((i.e., Euclidean, Hamming, or any other metric) between the enrolled and authentication template of a user against a predetermined threshold. If the distance is lower than the threshold value, then the user is successfully authenticated; otherwise he/she is rejected.

The performance of a biometric system can be estimated and quantified using the following two metrics: i) false acceptance rate (FAR) and ii) false rejection rate (FRR). FAR represents the probability that an authentication system will incorrectly accept an authentication attempt by an impostor (i.e., a non-valid user that does not have an

enrolled biometric template in the system); whereas FRR represents the probability that the system will incorrectly reject an authentication attempt by a genuine user (i.e., a valid and registered user of the system with an enrolled biometric template). As we analyze below, the exact value of FAR and FRR depend on the predetermined threshold value of the system. Another important metric that can be used to evaluate the authentication performance of a biometric system, is the Equal Error Rate (EER). The latter is the rate at which both acceptance and rejection errors are equal (i.e., EER=FAR=FRR). It is evident that the lower the value of EER is, the higher the accuracy of the biometric system.

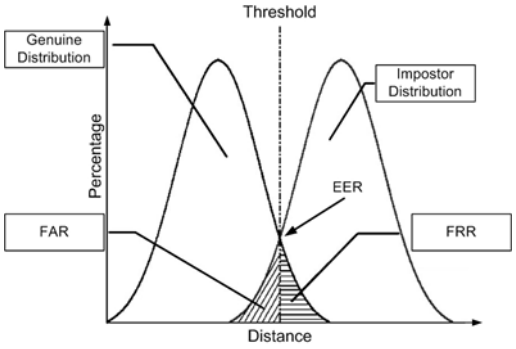

Figure 11: Genuine and impostor distributions as a function of distance between enrollment and authentication templates

To gain better understanding of the FAR, FRR and EER metrics, Figure 11 plots genuine and impostor distributions of a generic biometric system as a function of the distance between the enrolled and authentication templates. As expected, genuine users have small distances, while impostors have high distances. We can also observe that the two distribution curves have an overlapping area. This means that in this overlapping area the system cannot distinguish genuine users from impostors. Moreover, as shown in Figure 11, the threshold value is set at the intersection point of the two curves. The threshold value divides the overlapping area into two sub-areas. The left sub-area represents the FAR, while the right sub-area represents the FRR. The intersection point of the two curves defines the EER value (see Figure 11), since at this point the FAR and FRR are equal (i.e., EER=FAR=FRR). Moreover, it is evident that a biometric system presents optimum results (i.e., FAR and FRR equal to 0) when the genuine and impostor curves do not overall at all. On the other hand, as the overlapping area between the genuine and impostor curves increases, then the authentication performance is deteriorated.

4.1.2. Related Work

Over the last years, several studies have been performed to consider gait signatures, by using shape analysis and extracting features from the silhouette of the human body. Here, we provide a brief overview of the most recent works in this area. In [116], the authors pinpoint that temporal information is critical to the performance of gait recognition. To address this, they propose a novel temporal template, named chrono-gait image (CGI) in order to retain temporal information in a gait sequence. Moreover, the authors of [117] argue that the change of viewing angle of the sensor causes significant distortion to the extracted features. Based on this observation, they formulate a new patch distribution feature (PDF) to address this issue. The same viewing angle problem is addressed in [118]. The authors propose a transformation framework of the walking silhouettes to normalize gaits from arbitrary views. In [119], the proposed method is based on the idea that the problem of human gait recognition can be transformed from the spatiotemporal into the spatial domain, specifically, the 2D image domain. This is achieved by representing a sample of a human gait as a still image.

Towards this direction, [120] argues that variations of walking speed may lead to significant changes of human walking patterns. Based on this observation, a differential composition model (DCM) is proposed that differentiates the effects caused by walking speed changes on various human body parts; while at the same time it balances the different discriminabilities of each body part on the overall gait similarity measurements. In [121], the concept of the gait energy image (GEI) is extended from 2D to 3D images, creating gait energy volume (GEV). The obtained numerical results show that the GEV performance is improved, compared to the GEI baseline and fused multi-view GEI approaches. Next, in [122] the authors instead of using human silhouette images from moving picture, they apply 3D point clouds data of human body obtained from stereo camera, which has the scale-invariant property. In this way, they achieve significant performance improvement in terms of gait recognition. In [123], the authors propose a multi-view, multi-stance gait identification method, using unified multi-view population hidden Markov models, in which all the models share the same transition probabilities. Hence, the gait dynamics in each view can be normalized into fixed-length stances by Viterbi decoding. [124] provides an extensive overview of the methods used for accelerometer-based gait analysis, using mobile devices. In [125], the

extraction of distinguishable gait features is proposed using the radial integration transform (RIT), the circular integration transform (CIT), and the weighted Krawtchouk moments. In our proposed scheme, we use the CIT and RIT transformations for gait feature extraction, due to their excellent recognition capabilities

On the other hand, the related work in protection schemes for gait features is rather limited. In [126], the authors propose an authentication system that protects gait features using biometric cryptosystems. Gait features are extracted using an accelerometer attached to the user's body. Experimental results show that the proposed scheme achieves small EER values, only, for small key sizes. Thus, high accuracy is achieved without providing an adequate level of security. Finally, in [127], the authors propose a template protection scheme for gait features, based on channel coding (i.e., LDPC codes). Their approach achieves EER=6% for straight silhouette types, but 20% and 30% for bag and coat types respectively.

A common limitation of the majority of the literature is that it focuses, only, on the extraction and not on the protection of the gait features. On the contrary, as a part of this thesis we propose and integrate feature extraction and protection into one system, providing a complete solution for biometric authentication based on gait features. Moreover, the previous works [127] and [126] that attempt to secure gait features, fail to achieve an optimum tradeoff between security and performance. On the hand, by interpolating between the security of biohash and the recognition capabilities of gait features, we achieve to outperform existing solutions, without undermining the provided security. Finally, it is important to mention that biohash has been successfully applied to various biometric features including fingerprints [113] [128], face [129] [130], signatures [114], palmprints and palm veins [131] [132], but to the best of our knowledge it has not been applied to gait features.

4.1.3. Continuous authentication using the gait modality

4.1.3.1. Feature Extraction

For the extraction of gait features, this part considers three different types of human silhouettes: 1) straight (i.e., the user wears trousers, blouse and shoes), 2) coat (similar to straight silhouette, but the user also wears a coat), and, 3) bag (similar to straight silhouette, but the user carries also a briefcase). It is worth noting that although the current work considers only the above three types of silhouettes, the proposed authentication system can be easily extended to take into account other types of

silhouettes (e.g., the user wears a hat) or various combinations (e.g., a user wearing a coat and a hat).

The extraction of gait features is based on two feature-based algorithms: the RIT and CIT transformations. These algorithms are selected due to their capability to represent important shape characteristics [131]. That is, during human movement, there is a considerably large diversity in the angles of lower parts of the body (e.g. arms, legs), which vary among individuals. Both RIT and CIT transformations ensure that the important dynamics of human shape are captured, thus enabling the correct classification of individuals. Moreover, these algorithms are less sensitive to the presence of noise on the silhouette image, compared to other schemes [131].

At this point, we provide a brief presentation of these transformations, where additional details can be found in [125]. The first step in gait analysis is the extraction of the walking subject's silhouette from the input image sequence. The normalized silhouettes are defined as $\tilde{S}_G(x, y)$ where transformations are applied. More specifically, the RIT transform of a function $f(.,.)$ is defined as the integral of $f(.,.)$ along a line starting from the center of the silhouette(x_0, y_0), which forms angle θ with the horizontal axis. The discrete form of RIT, which computes the transform in steps of $\Delta\theta$ is given by:

$$RIT(t\Delta\theta) = \frac{1}{J}\sum_{j=1}^{J}(\tilde{S}_G(x_0 + j\Delta u * \cos(t\Delta\theta), y_0 + j\Delta u * \sin(t\Delta\theta))),$$

where $\tau = 1, \dots, T, \Delta u$ and $\Delta\theta$ are constant step sizes of distance u and angle θ, J is the number of silhouette pixels that coincides with the line that has orientation θ and are positioned between the center of the silhouette and the end of the silhouette in that direction, and $T = 360^\circ/\Delta\theta$.

In a similar manner, CIT is defined as the integral of a function $f(.,.)$ along a circle curve $h(\rho)$ with center (x_0, y_0) and radius ρ. The discrete form of the CIT transform is given by:

$$CIT(k\Delta\rho) = \frac{1}{T}\sum_{t=1}^{T}(\tilde{S}_G(x_0 + k\Delta\rho * \cos(t\Delta\theta), y_0 + k\Delta\rho * \sin(t\Delta\theta))),$$

where $k = 1, \dots K, \Delta\rho$ and $\Delta\theta$ are the constant step sizes of the radius and angle variables, $k\Delta\rho$ is the radius of the smallest circle that encloses the binary silhouette

image $\tilde{S}_G$, and $T = 360^{\circ}/\Delta\theta$. The output of the CIT and RIT transformations are the fixed-length vectors Γ_{CIT} and Γ_{RIT} of size $n_1 = 80$ and $n_2 = 120$ respectively.

4.1.3.2. Biohashing

After the extraction of the gait features (using the CIT and RIT transformations), the biohash algorithm is applied to secure them. The biohash algorithm *is a two-factor authentication scheme* that identifies a user based on what he/she is (i.e., biometrics) and what he/she has under his/her possession (i.e., token). In the context of our proposed scheme, the biohash algorithm converts the gait feature vectors Γ_{CIT} and Γ_{RIT} to non-invertible bitstreams, using a token that the user possesses. Since the application of biohash is similar to both CIT and RIT vectors, here we present the biohash algorithm in a generic way. More specifically, we present the application of biohash to a vector Γ of size n, which is converted to a bitstream B. Biohash includes the following phases [115]:

1. The token of the user generates a set of orthonormal pseudorandom vectors

$$\{r_i \in R^n | i = 1, \dots, n\},$$

2. A vector Z of size n with elements z_i is computed such as:

$$z_i = \langle \Gamma | r_i \rangle \in R, i = \{1, \dots, n\},$$

where $\langle . | . \rangle$ indicates the inner product operation. This procedure is also known as random projection.

3. The mean value μ and standard deviation σ of z_i are computed.

4. The final step is the binarization of z_i. As shown in Table 15, first it divides the real-space of z_i into 8 segments. Next, each segment is mapped to a three bit digit value $b_i \in \{0,1\}^3$, so that two successive segments have only one bit difference between them (see Table 16). In this way, it transforms the elements of vector Z into a bitstream $B = \{b_1 b_2 \dots b_n\}$ of $3n$ bits length.

Segment	z_i	b_i
1	$-\infty \leq z_i < \mu - 3\sigma$	000
2	$\mu - 3\sigma \leq z_i < \mu - 2\sigma$	001
3	$\mu - 2\sigma \leq z_i < \mu - \sigma$	011
4	$\mu - \sigma \leq z_i < \mu$	010
5	$\mu \leq z_i < \mu + \sigma$	110
6	$\mu + \sigma \leq z_i < \mu + 2\sigma$	111
7	$\mu + 2\sigma \leq z_i < \mu + 3\sigma$	101
8	$\mu + 3\sigma \leq z_i < +\infty$	100

Table 15: Conversion of z_i to b_is

4.1.4. Initial experiments and observations

In this section we propose and evaluate experimentally two initial enrollment and authentication schemes. As we analyze below, despite the fact that these two schemes proved inadequate, due to their poor authentication performance, they provided useful observations and insights that allowed us to fine-tune and design and optimal enrollment and authentication scheme that is presented in section 4.1.5.

As we mentioned in section 4.1.3.1, we consider three types of gait features that are extracted from three types of human silhouettes: i) straight $G_{straight}$, ii) coat G_{coat}, and, iii) bag G_{bag}. Thus, an important question that arises here is: *Which one of the three considered gait features the authentication system should enroll?* To answer this question, we consider the following two enrollment and authentication schemes each of which encompasses a different technical approach:

1st scheme: Enrollment of one of the three considered gait feature vectors. The selection of the specific silhouette type that will be used for enrollment is arbitrary.

2nd scheme: First, a feature-level fusion of all three gait feature vectors is performed. Next, we enroll the single vector generated from the fusion.

In the sections below, we present and evaluate through experiments the two above mentioned enrollment and authentication schemes.

4.1.4.1. 1st scheme

In the first scheme, we enroll gait features that are extracted only from one of the three considered types of human silhouettes. The specific gait feature that will be used for

enrollment is selected arbitrary. In this analysis, we consider gait features from a straight human silhouette to be used for enrollment (note that the same procedure is followed, if another type of human silhouette is selected for enrollment). In this case, the CIT and RIT transformations are applied to extract the gait features from a straight silhouette $G_{straight}$. That is,

$$GaitVector_{(cit,straight)} = CIT\_Transformation(G_{straight}),$$
$$GaitVector_{(rit,straight)} = RIT\_Transformation(G_{straight}).$$

Next, the biohash algorithm is applied to the two feature vectors (i.e., one for CIT and one for RIT), in order to generate two different enrollment bitstreams, denoted $Ebits_{(cit, straight)}$ and $Ebits_{(rit, straight)}$, respectively, which are stored in the enrollment database. That is:

$$Ebits_{(cit,straight)} = Biohash(GaitVector_{(cit,straight)}, Token),$$
$$Ebits_{(rit,straight)} = Biohash(GaitVector_{(rit,straight)}, Token).$$

In the authentication procedure, the silhouette G of the user can be one of the three types (i.e., straight, coat, bag). First, the CIT and RIT transformation are applied to extract two gait feature vectors (i.e., one from CIT and one from RIT) as follows:

$$GaitVector_{(cit)} = CIT\_Transformation(G),$$

$$GaitVector_{(rit)} = RIT\_Transformation(G).$$

Next, using the user's token and the extracted feature vectors, biohash is applied to generate two different authentication bitstreams $Abits_{(cit)}$ and $Abits_{(rit)}$. That is:

$$Abits_{(cit)} = Biohash(GaitVector_{(cit)}, Token),$$

$$Abits_{(rit)} = Biohash(GaitVector_{(rit)}, Token).$$

At this point, the hamming distance between the authentication and the enrollment bitstreams is computed, separately for each transformation. Finally, the sum of the two hamming distances is computed as follows:

$$FinalResult = HDistance(Ebits_{(cit,straight)}, Abits_{(cit)}) +$$
$$HDistance(Ebits_{(rit,straight)}, Abits_{(rit)})$$

Finally, a user is accepted if FinalResult is less than a predetermined threshold, otherwise he/she is rejected.

4.1.4.2. 2nd scheme

In the second scheme, we apply feature-level fusion [133], in order to enroll gait features from all the three considered human silhouettes. In particular, the CIT and RIT transformations are applied to extract the gait features from the three considered human silhouettes: i) straight, ii) coat, and, iii) bag. Next, we fuse the extracted feature vectors to create two mean feature vectors $GaitVector_{(cit,fused)}$ and $GaitVector_{(rit,fused)}$as follows:

$$GaitVector_{(cit,fused)}=\frac{GaitVector_{(cit,straight)}+GaitVector_{(cit,bag)}+GaitVector_{(cit,coat)}}{3},$$

$$GaitVector_{(rit,fused)}=\frac{GaitVector_{(rit,straight)}+GaitVector_{(rit,bag)}+GaitVector_{(rit,coat)}}{3}.$$

Subsequently, biohash is applied to the two mean feature vectors, in order to generate two different enrollment bitstreams denoted $Ebits_{(cit,\ fusion)}$ and $Ebits_{(rit,\ fusion)}$, respectively, which are stored in the enrollment database. The computation of the enrollment bitstreams is performed as follows:

$$Ebits_{(cit,fusion)} = BioHash(GaitVector_{(cit,fused)}),$$

$$Ebits_{(rit,fusion)} = BioHash\left(GaitVector_{(rit,fused)}\right).$$

Similarly to the first scheme, in the authentication procedure, the silhouette G of the user can be one of the three types that were captured in the enrollment procedure (i.e., straight, coat, bag). First, the CIT and RIT transformations are applied to extract two gait feature vectors (i.e., one from CIT and one from RIT). As previously, using the user's token and the gait features vectors, biohash is applied to generate two different authentication bitstreams $Abits_{(cit)}$ and $Abits_{(rit)}$. Next, the hamming distance between the authentication and the enrollment bitstreams is computed, separately, for each transformation. After that, the final score named FinalResult is computed, which is the sum of the two previously computed hamming distances. That is:

$$FinalResult = HDistance\left(Ebits_{(cit,fusion)},\ Abits_{(cit)}\right) +$$

$$HDistance\left(Ebits_{(rit,fusion)},\ Abits_{(rit)}\right)$$

4.1.4.3. Experiments and numerical results

In this section, we evaluate the authentication performance of the two enrollment and authentication schemes. To this end, we have implemented in C++ programming language the following software modules: i) the CIT and RIT transformation

algorithms, ii) the biohash algorithm, and iii) the above two enrollment and authentication schemes. In the carried out experiments, we captured silhouettes of 75 subjects (i.e., users). Three different human silhouette categories were considered: a) straight, b) coat, and, c) bag. The relative position of the camera and the subject was vertical. Thus, the angle of the direction of the camera and the face of the subject was 90 degrees.

The evaluation of the two schemes is performed by computing the genuine and impostor distributions. More specifically, to investigate the authentication performance of the proposed scheme, we classify the users as: a) genuine and b) impostors. Let user A be a genuine user with a token denoted as TRN_A, while his/her biometric data is denoted as $GAIT_A$. Assume now that an impostor has his/her own biometric data $GAIT_{impostor}$ and his/her own token $TRN_{impostor}$. The goal of the impostor is to be authenticated as user A. We identify three different attack scenarios for the impostor: i) a type 1 impostor uses his own biometric data $GAIT_{impostor}$ and his own $TRN_{impostor}$; ii) a type 2 impostor has stolen and uses user's A token TRN_A but uses his/her own biometric data $GAIT_{impostor}$; and iii) a type 3 impostor has stolen and uses the biometric data of user A $GAIT_A$ and uses his/her own $TRN_{impostor}$. Impostors of type 1 are weaker (in terms of probability of successful authentication as genuine users) than impostors of type 2 and 3, since they do not possess any authentication credential (token or gait features). It is evident that in case that an impostor possesses both gait features and the token of a valid user, then he/she can be successfully authenticated as a genuine user.

Figure 12 shows the genuine and impostor distributions for the first scheme (recall that the straight silhouette has been selected to enroll gait features). Note that since the genuine bag and coat distributions had exactly the same curves they are presented as one curve named genuine bag/coat. The same applies also for type 1 and 3 impostors distributions and, therefore, their curves are represented by a single one named type 1/3. Figure 12 shows that the type 1/3 impostors are clearly separated (i.e., no overlap) from the genuine distributions, which means that the 1st scheme achieves EER=FAR=FFR=0%. We also observe that the genuine straight distributions have a very small overlap with type 2 impostors. We have estimated that the EER value for type 2 impostors and genuine straight is equal to 9%. However, it can be deduced from Figure 12 that genuine bag/coat distributions overlap greatly with type 2 impostor distribution, which means that the system cannot distinguish them. As a matter of fact,

we have derived the EER value equal to 34% for type 2 impostors and genuine bag/coat, which is considerably high and unacceptable.

It is worth noting that we repeated the experiments using this time gait features extracted from a bag silhouette as enrollment. Again, the same distribution behavior was observed with the difference that this time genuine bag distributions had a small overlap with type 2 impostors, while straight/coat curves overlapped greatly with type 2 impostors. In this case, the Type 2 EER value was derived equal to 33%. Note that similar results we observed using a coat silhouette as enrollment. From the above analysis, we deduce the following observation:

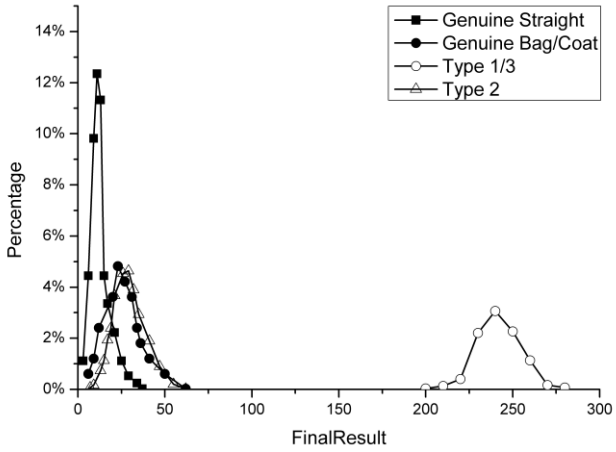

Figure 12: Distributions of the FinalResult values of the first scheme for genuine users and impostors.

Observation 16: *Gait features that are extracted from the same user are similar only when they are extracted from the same silhouette type. On the contrary, gait features that are extracted from different silhouette types of the same user have great differences.*

The above observation indicates that if, for example, we use enrollment templates generated from a straight silhouette type, then a valid user may be rejected if his/her authentication templates are generated from bag or coat types. Similarly, if we use gait features extracted from bag silhouette as enrollment template, then a valid user may be rejected, if the silhouette type for authentication is straight or coat. This happens because when the enrollment and authentication templates (i.e., gait features) are generated from different silhouette types, the extracted gait vectors differ significantly,

due to distortions that are caused by the different captured silhouette type. The above leads to the more generic observation:

Observation 17: *If we use enrollment templates only from one silhouette type, then the authentication performance is significantly deteriorated.*

Figure 13 shows the genuine and impostor distributions for the second enrollment and authentication scheme. First, we observed that all three genuine silhouette types had exactly the same distribution curve. For this reason, Figure 13 shows one genuine distribution curve that represents all silhouette types. It is observed again that the type 1/3 and genuine distributions are clearly separated and thus EER=FAR=FFR=0% is achieved for these types of impostors. On the other hand, the type 2 impostor distribution overlaps almost entirely with the genuine one, resulting in a very high EER value equal to 45% for type 2 impostors. This means that if we use feature fusion at the enrollment phase, the authentication performance is worse than the first scheme for all silhouette types.

4.1.5. User registration and authentication using the gait modality.

In this section, we describe the final enrollment and authentication scheme called *gaithashing* that yields the best numerical results. Unlike the previous two schemes that enroll only one feature gait vector (i.e., from a specific type of silhouette or fused), gaithashing enrolls separately gait feature vectors from all the three considered human silhouette types. Moreover, in the authentication process of gaithashing, the new extracted gait features are fused with each one of the enrollment templates, using weighted sums. By selecting appropriate weight values, gaithashing performs comparison between gait features of the same silhouette type, in order to increase the authentication performance and avoid the pitfalls of the previously mentioned schemes.

From the above analysis, we deduce the following observation:

Observation 18: *Feature-level fusion has adverse impact on the authentication performance.*

More specifically, as shown in Figure 14, the first step of the enrollment procedure in gaithashing is to capture the aforementioned three distinct silhouettes of the user: a) straight $G_{straight}$, b) coat G_{coat}, and, iii) bag G_{bag}. Next, the CIT and RIT transformations are applied, separately, to each one of the three silhouettes of the user to extract the gait

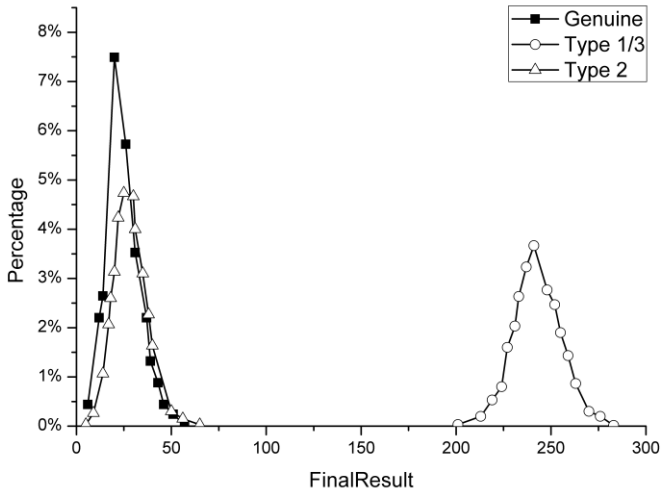

Figure 13: Distributions of the FinalResult values of the second scheme for genuine users and impostors.

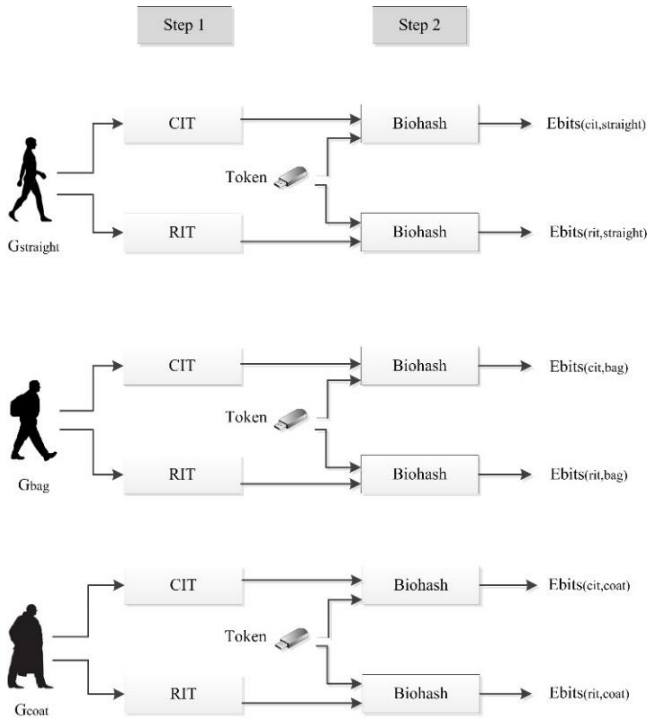

Figure 14: Gaithashing enrollment procedure

Algorithm 1: Enrollment Algorithm

Input: Three gait silhouettes **(**$G_{straight}$**,** G_{bag}**,** G_{coat}**),** Token
Output: Six enrollment Bitstreams **(**$Ebits_{(cit,straight)}$**,** $Ebits_{(cit,bag)}$**,** $Ebits_{(cit,coat)}$**,** $Ebits_{(rit,straight)}$**,** $Ebits_{(rit,bag)}$ **,** $Ebits_{(rit,coat)}$**)**

1. Categories**={**straight**,**bag**,**coat**}**
2. **for** i in Categories **do**
3. $GaitVector_{(cit,i)} = CIT\_Transformation(G_{(i)})$**;**
4. $GaitVector_{(rit,i)} = RIT\_Transformation(G_{(i)})$;
5. $Ebits_{(cit,i)} = Biohash(GaitVector_{(cit,i)}, Token)$;
6. $Ebits_{(rit,i)} = Biohash(GaitVector_{(rit,i)}, Token)$;
7. **end**

Figure 15: Gaithashing enrollment algorithm

features. In this way, in total, six different gait features are extracted: three from the CIT transformation and three from RIT. In the second step, biohash is applied to each one of the six gait features using the token of the user, generating six different enrollment bitstreams. That is, three enrollment bitstreams for the CIT transformation $Ebits_{(cit,straight)}$, $Ebits_{(cit,bag)}$, $Ebits_{(cit,coat)}$, and three enrollment bitstreams for RIT $Ebits_{(rit,straight)}$, $Ebits_{(rit,bag)}$, $Ebits_{(rit,coat)}$, which are stored in the enrollment database. The algorithm of the enrollment procedure is presented in Figure 15.

The authentication procedure includes four distinct steps. Note that in the authentication procedure, the silhouette G of the user can *be one of the three types that were captured in the enrollment procedure (i.e., straight, coat, bag)*. In the first step, the CIT and RIT transformation are applied to extract two different gait features (i.e., one from CIT and one from RIT). In the second step, using the user's token and the extracted features, biohash is applied to generate two different authentication bitstreams $Abits_{(cit)}$ and $Abits_{(rit)}$. During the third step, the authentication and the enrollment bitstreams are compared and fused, separately, for each transformation to produce the intermediate scores CitSum and RitSum (i.e., first-level fusion as shown in Figure 16). Finally, in the fourth step, the CitSum and RitSum are fused (i.e., second-level fusion as shown in Figure 16) to generate the final score named as FinalResult. At this point, the user is accepted if FinalResult is less than a predetermined threshold; otherwise he/she is rejected. As mentioned below, the first and second level fusions are based on weighted sums. The exact values of the employed weights as well as the predetermined threshold are derived experimentally (see section 4.1.6), maximizing the authentication performance.

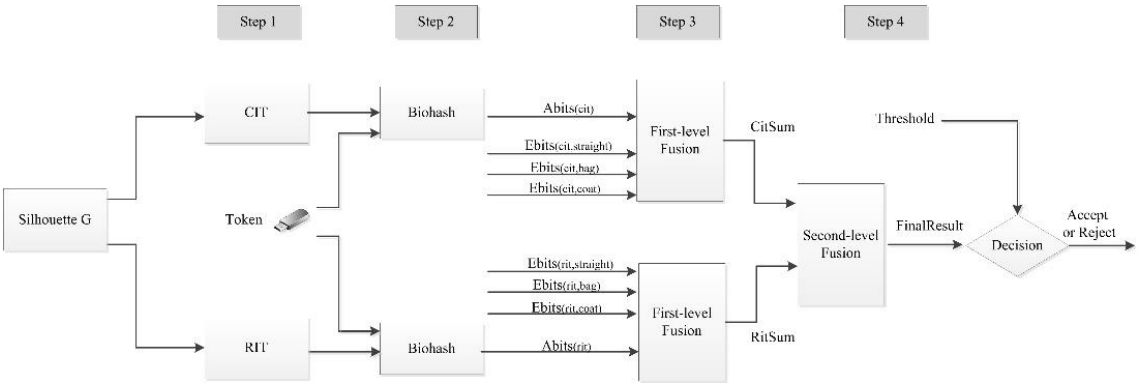

Figure 16: Gaithashing authentication procedure

First-level fusion

The first-level fusion module is invoked in the authentication procedure, right after the generation of the authentication bitstreams. This module calculates the hamming distances between each authentication and enrollment bitstream of the user. Note that the hamming distance represents the number of different bits between two bitstreams. In total, three hamming distances are computed for each transformation (CIT and RIT) as follows:

$$Score_{(cit,straight)} = HDistance\left(Ebits_{(cit,straight)},\ Abits_{(cit)}\right),$$

$$Score_{(cit,bag)} = HDistance\left(Ebits_{(cit,bag)},\ Abits_{(cit)}\right),$$

$$Score_{(cit,coat)} = HDistance\left(Ebits_{(cit,coat)},\ Abits_{(cit)}\right).$$

and

$$Score_{(rit,straight)} = HDistance\left(Ebits_{(rit,straight)},\ Abits_{(rit)}\right),$$

$$Score_{(rit,bag)} = HDistance\left(Ebits_{(rit,bag)},\ Abits_{(rit)}\right),$$

$$Score_{(rit,coat)} = HDistance\left(Ebits_{(rit,coat)},\ Abits_{(rit)}\right).$$

A small hamming distance value between the authentication and enrollment bitstreams means that the compared bitstreams are similar. On the contrary, a high hamming distance value means that the compared bitstreams are different and they do not share similarities.

Since the user's silhouette type should match with one of the three enrollment types, it is evident that one of the previously generated scores from the RIT transformation and one from CIT have small hamming distance values (see observation 16), while the

remaining scores have high hamming distance. Let X_1 be the minimum between the three scores of CIT, that is,

$$X_1 = Min(Score_{(cit,straight)}, Score_{(cit,bag)}, Score_{(cit,coat)}),$$

and X_2, X_3 the remaining two scores. Similarly, we assign Y_1 the minimum between the three scores of RIT:

$$Y_1 = Min\big(Score_{(rit,straight)}, Score_{(rit,bag)}, Score_{(rit,coat)}\big),$$

and Y_2, Y_3 the remaining two scores. In essence, X_1 and Y_1 represent the hamming distance between authentication and enrollment bitstreams of the same silhouette type, while X_2, X_3 and Y_2,Y_3 represent the hamming distance between authentication and enrollment bitstreams of different silhouette types. In other words, the values of X_2, X_3 and Y_2,Y_3 are considered to be noise. At this point, the first-level fusion module fuses the hamming distances of each transformation using weighted sums and generates two intermediate scores, CitSum and RitSum such as:

$$CitSum = \alpha_1 * X_1 + \alpha_2 * X_2 + \alpha_3 * X_3,$$

$$RitSum = b_1 * Y_1 + b_2 * Y_2 + b_3 * Y_3,$$

where $\alpha_1, \alpha_2, \alpha_3$ and b_1, b_2, b_3 are weight values such as $\alpha_1 > \alpha_2, \alpha_3$ and $b_1 > b_2, b_3$,while it is $\alpha_1 + \alpha_2 + \alpha_3 = 1$ and $b_1 + b_2 + b_3 = 1$. Note that the impact of X_1 and Y_1 on the value of CitSum and RitSum respectively is greater than the other scores. This happens because their corresponding weight values (i.e., α_1 and b_1) are greater than the other weight values. In this way, the noise introduced by X_2, X_3 and Y_2,Y_3 do not affect, significantly, the value of CitSum and RitSum.

Second-level fusion and decision

In this step, first a final score (denoted as FinalResult) is computed by fusing the CitSum and RitSum values, using weighted sums such as:

$$FinalResult = w_1 * CitSum + w_2 * RitSum,$$

where $w_1$and w_2 are weights such as $w_1 + w_2 = 1$. Finally, the user is accepted or rejected based on the following simple rule: If FinalResult is less than a predetermined threshold, then the user is authenticated successfully; otherwise the user is rejected. The algorithm of the authentication procedure is presented in Figure 17.

4.1.6. Performance evaluation

To evaluate the authentication performance of the proposed scheme, we have implemented the two-level fusion and decision algorithm of gaithashing. The parameters of the carried out experiments are the same as in section 4.3. That is, three different human silhouette categories were considered: a) straight, b) coat, and, c) bag. Moreover, we classify the users as: a) genuine and b) impostors. We identify three different attack scenarios for the impostor: i) a type 1 impostor uses his own biometric data and his/her own token; ii) a type 2 impostor has stolen and uses a valid token of a genuine user but uses his/her own biometric data; and iii) a type 3 impostor has stolen and uses the biometric data of a genuine user but uses his/her own token.

Algorithm 2: Authentication Algorithm

Input: An authentication gait silhouette **(**G**),** Six Enrollment Bitstreams**,** Token**,** Threshold

Output: Acceptance or rejection of the user

1**:** Categories**={**straight**,** bag**,** coat**}**

2**:** $GaitVector_{(cit)} = CIT\_Transformation(G)$**;**

3**:** $GaitVector_{(rit)} = RIT\_Transformation(G)$**;**

4**:** $Abits_{(cit)} = Biohash(GaitVector_{(cit)}, Token)$**;**

5**:** $Abits_{(rit)} = Biohash(GaitVector_{(rit)}, Token)$**;**

6**: for** i in Categories **do**

7**:** $Score_{(cit,i)} = HDistance(Ebits_{(cit,i)}, Abits_{(cit)})$**;**

8**:** $Score_{(rit,i)} = HDistance(Ebits_{(rit,i)}, Abits_{(rit)})$**;**

9**: end**

10**:** $X_1 = Min(Score_{(cit,straight)}, Score_{(cit,bag)}, Score_{(cit,coat)})$ and X2**,** X3 the remaining two scores**;**

11**:** $Y_1 = Min(Score_{(rit,straight)}, Score_{(rit,bag)}, Score_{(rit,coat)})$ and Y2**,** Y3 the remaining two scores**;**

12**:** $CitSum = \alpha_1 * X_1 + \alpha_2 * X_2 + \alpha_3 * X_3$**;**

13**:** $RitSum = b_1 * Y_1 + b_2 * Y_2 + b_3 * Y_3$**;**

14**:** $FinalResult = w_1 * CitSum + w_2 * RitSum$**;**

15**: if** $FinalResult < Threshold$ **then**

16**:** User is accepted**;**

17**: else**

18**:** User is rejected**;**

19**: end**

Figure 17: Gaithashing authentication algorithm

We have conducted two set of experiments. The aim of the first set is to derive the distributions of the FinalResult values for both genuine users and impostors (all three types). The FinalResult is the most important parameter in the proposed scheme, since the authentication of a user is based on its value. By investigating the distribution of FinalResult values, we gain insights for the behavior of the gaithashing scheme and whether it can distinguish impostors from genuine users. In the second set of experiments, the goal is to estimate the FAR, FRR and EER values. As mentioned

previously (see section 4.1.1), FAR represents the probability that the authentication system will incorrectly accept an authentication attempt by an impostor, whereas FRR represents the probability that the authentication system will incorrectly reject an authentication attempt by a genuine user. This experiment allows us to estimate an appropriate threshold value that can minimize both FAR and FRR, at the same time.

In the carried out experiments, the values of weights were set as follows:$\alpha_1 = b_1 = 0.5, \alpha_2 = b_2 = 0.25, \alpha_3 = b_3 = 0.25$(first-level fusion) and $w_1 = 0.4, w_2 = 0.6$ (second-level fusion). As we analyze below, these values were selected after trying various combinations and experiments, in order to achieve the best authentication performance (i.e., minimize the EER value).

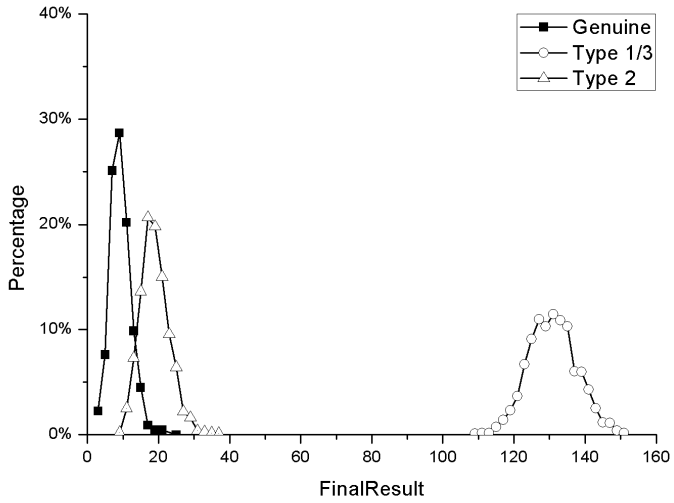

Figure 18: Distributions of the FinalResult values of gaithashing for genuine users and three impostor types

Figure 18 shows the distribution of the FinalResult values for both impostors 1, 2, 3 and genuine users. Note that the distributions of impostors type 1 and 3 were identical and are presented in one curve. It is observed that the FinalResult values of type 1 and type 3 impostors is considerably higher than the genuine. In fact, the highest value of FinalResult for genuine users is 25, while the values of FinalResult for impostors type 1/3 begins at 110. As a result, the distribution curves of the genuine users and type 1/3 impostors do not overlap at all. This means that gaithashing can always distinguish between impostors type 1/3 and genuine users. In other words, an impostor of type 1 and 3 cannot be authenticated as genuine user. For example, if we set the threshold value equal to 60, then the FinalResult value for all genuine users is less than the

threshold value, while all impostors of type 1 and 3 have FinalResult value higher than the threshold, which means that they will be rejected. On the other hand, we observe that the type 2 impostor distribution marginally overlaps with the genuine one. The intersection area of the two curves (i.e., genuine and impostor type 2 distribution) begins for FinalResult equal to 10 and ends for FinalResult equal to 25. In this area, gaithashing cannot distinguish between genuine users and type 2 impostors, since they share the same FinalResult values. The above results indicate that depending on the value of the selected threshold, an impostor type 2 may be authenticated, successfully, as a genuine user or a genuine user may be rejected, incorrectly. For example, if we set threshold equal to 10, then as shown in Figure 18, no impostor of type 2 will be accepted. However, a small percentage of genuine users will be rejected, because their FinalResult value is greater than the threshold.

To quantify and investigate further the authentication performance of gaithashing, we have estimated the FAR and FRR values, as a function of threshold values (see Figure 19). As expected, the value of FRR decreases, as the threshold increases. On the other hand, the values of FAR for the three impostors types increases as the threshold increases. Thus, the value of the threshold regulates a tradeoff between FAR and FRR. A small threshold value may minimize FAR, but the FRR may be very high. On the contrary, a high threshold value may minimize FRR, but the value of FAR can be very high. For this reason, we have to estimate the EER value (see section 4.1.1), where the FAR and FRR are equal (i.e., EER=FAR=FRR). Evidently, the value of EER should be as low as possible, since a low value of EER entails a low value of FAR and FRR. This value can be easily estimated, since it is the intersection point of the FAR and FRR curves. Thus, as shown in Figure 19, for impostors of type 2, the EER equals to 10.8% which is obtained for threshold value equal to 14. This means that if we set the threshold equal to 14, then for 100 authentication attempts, the proposed scheme presents in total 10 false rejections of a genuine user or false acceptance of a type 2 impostor. Moreover, the EER for impostors of type 1/3 is equal to 0%, since the FRR and FAR curves do not intersect. This means that gaithashing is able to always detect type 1/3 impostors. Thus, we can deduce that the proposed scheme attains very high performance for all impostor scenarios, while false alarms are kept to minimal.

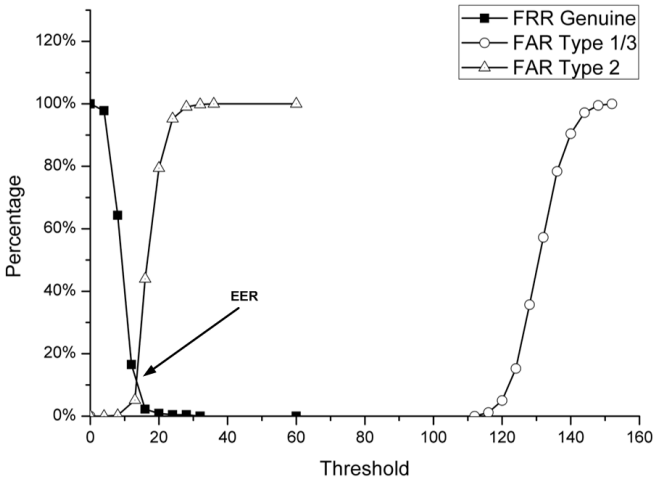

Figure 19: Gaithashing FRR-FAR values as functions of the threshold value

It is important to mention that the employed weight values for the first and second level fusion play a key role in the performance of gaithashing. These were derived after a fine tuning procedure in which we performed several trials in order to minimize the EER value. More specifically, Table 16 shows various weight values that we tested and the corresponding EER value for impostors of type 2 (note that the EER value for impostors type 1/3 was equal to 0% independently of weight values). Recall that $\alpha_1 > \alpha_2, \alpha_3$ and $b_1 > b_2, b_3$, while it is $\alpha_1 + \alpha_2 + \alpha_3 = 1$, $b_1 + b_2 + b_3 = 1$ and w_1+$w_2 = 1$. First, we randomly selected weights values for the first-level fusion, while the weights for the second level fusion were constant and equal to $w_1 = w_2 = 0.5$. Initially, we tested the following weight values: $\alpha_1 = 0.5$, $\alpha_2 = \alpha_3 = 0.25$ and $b_1 = 0.5$, $b_2 = b_3 = 0.25$, (1st trial). Numerical results showed that gaithashing achieved EER=11.4%. Next, in the 2nd trial we increased the values of α_1 (i.e., $\alpha_1 = 0.6$) and b_1 (i.e., $b_1 = 0.6$) and we observed that the EER value increased (i.e., EER=13.2%), which was not acceptable. In the third trial we increased only the value of α_1 (i.e., $\alpha_1 = 0.6$), while b_1 was equal to its initial value (i.e., $b_1 = 0.5$). Again, we observed that the value of EER was higher compared to the first trial (i.e., EER=12.5%). In the fourth trial, we reduced α_1 (i.e., $\alpha_1 = 0.4$) and b_1 (i.e., $b_1 = 0.4$). We observed that the value of EER did not modified, significantly, but it was higher than the first trial (i.e., EER=13.2%).

Trials	α_1	α_2, α_3	b_1	b_2, b_3	w_1	w_2	EER
1	0.5	0.25	0.5	0.25	0.5	0.5	11.4%
2	0.6	0.2	0.6	0.2	0.5	0.5	13.2%
3	0.6	0.2	0.5	0.25	0.5	0.5	12.5%
4	0.4	0.3	0.4	0.3	0.5	0.5	13.2%
5	0.5	0.25	0.5	0.25	0.6	0.4	11.6%
6	0.5	0.25	0.5	0.25	0.4	0.6	**10.8**%

Table 16: Gaithashing tested weight values and corresponding EER of type 2 impostors

Next, we modified the weight values of the second level fusion w_1 and w_2, while the weight values of the first-level fusion are constant and equal to the first trial. As shown in Table 16, in the 5th trial we assigned $w_1 = 0.6$ and $w_2 = 0.4$ and observed that the value of EER was not significantly modified, compared to the first trial (i.e., EER=11.6%). In the 6th trial, we selected $w_1 = 0.4$ and $w_2 = 0.6$. This time we observed that the value of EER was decreased, compared to the first trial and it was equal to 10.8%. Although we performed several other trials, the value of EER was not reduced further. Thus, we concluded that the weight values of the sixth trial should be selected in order to achieve the minimum EER value (i.e., EER=10.8%).

Apart from the aforementioned experiments, it is important to mention that we tried to further improve the EER value of gaithashing for type 2 impostors, using decision based fusion. In particular, we have implemented a scheme that performs two-level fusion. The first-level fusion is identical with gaithashing. That is, the hamming distances between each authentication and enrollment bitstreams of the subject are calculated and the CitSum and RitSum are derived using weights. In the second-level fusion, the CitSum and RitSum values are compared to two pre-defined thresholds (i.e., $\text{Threshold}_{\text{cit}}$ and $\text{Threshold}_{\text{rit}}$ respectively) to derive a binary decision (i.e., TRUE or FALSE). That is:

$$CitAuth = \begin{cases} TRUE, if\ CitSum < Threshold_{cit} \\ FALSE, if\ CitSum \geq Threhold_{cit} \end{cases}$$

$$RitAuth = \begin{cases} TRUE, if\ RitSum < Threshold_{rit} \\ FALSE, if\ RitSum \geq Threhold_{rit} \end{cases}$$

The final result denoted as FinalAuth is calculated by performing a decision-level fusion using the AND or OR logical rules. In particular, using the OR logical rule, a user is successfully authenticated if either the CitAuth or RitAuth value is TRUE,

whereas using the AND rule, both CitAuth and RitAuth values should be TRUE. To obtain numerical results (i.e., EER), we tested various values for the $\text{Threshold}_{\text{cit}}$ and $\text{Threshold}_{\text{rit}}$. The lowest EER values that we achieved for type 2 impostors were equal to 48% and 19% for the OR and rules respectively. On the other hand, as we mentioned previously gaitashing achieved EER =10.8%. Thus, it is evident that the decision based fusion approach does not improve the EER of gaithashing and as a matter of fact, it deteriorates the authentication performance [134].

To summarize, the EER values of the three proposed schemes are shown in Table 17. We conclude that all schemes achieve 0% EER for both Type 1 and 3 impostors. However, for type 2 impostors, we obtained EER = 34% for straight silhouette enrollment, as well as 27% and 32% for coat and bag enrollment respectively. Moreover, in the second scheme the EER was equal to 45%. However, the third scheme achieves EER = 10.8%, which is a significant improvement over the previous two schemes. This result means that for every 100 authentication attempts, the third scheme has in average 10 false acceptances of type 2 impostors and 10 false rejections of genuine users.

Apart from the fusion techniques, there are some other methods that could possibly improve the authentication performance of the system. In particular:

a) Use of multiple feature extraction algorithms: Apart from CIT and RIT transformation algorithms, we can extract gait features using other feature extraction algorithms proposed

Impostors type	**1st scheme**	**2nd scheme**	**3rd scheme (Gaithashing)**
Type 1	0%	0%	0%
Type 2	34% straight enrollment 27% coat enrollment 32% bag enrollment	45%	10.8%
Type 3	0%	0%	0%

Table 17: EER values of the three proposed schemes

in the literature (such as the ones presented in [120] and [119]). As a matter of fact, we can use multiple extraction algorithms to extract multiple gait features for the same user. Since different algorithms capture different characteristics of a human silhouette, we can enroll all extracted features and perform a feature-level fusion, in order to

improve the authentication performance. The negative side effect of this approach is that it increases the overall complexity as well as the processing and storage overhead, due to the extraction and enrollment of several gait features for each user.

b) Use of multi-modal biometrics: The ISO/IEC standards propose the use of multiple biometric features (i.e., also named as multi-modal biometrics), in order to overcome the limitations imposed by uni-modal biometric systems [134]. In general, multi-modal biometric systems are considered to be more reliable and robust to attacks [135], since an impostor should compromise two or more biometric features of a genuine user. In the proposed gaithashing system, gait features can be combined with face or iris or any other biometric modality to create a feature vector for the user. The downside of this approach is that the proposed system will inherit the usability issues of the other biometric modalities. That is, gait is the only biometric modality that provides unconstructive access control and authentication at-a-distance. All other biometric modalities (including fingerprints, iris, face) have several usability issues (see section 4.1.6). Therefore, on the one, hand multimodal biometrics may improve the EER results, but on the other hand it will reduce the usability of the system.

c) Use of multiple sensors: Another improvement in the authentication performance may be achieved by using multiple sensors. That is, we can use different cameras to capture the human silhouette of a user and obtain multiple gait features (each one derived from a different camera) that can be used for enrollment. However, we have to notice that the use of multiple cameras may cause deployment issues and increase the overall cost.

4.1.7. Results and discussion

Section 4.1 of this thesis proposed gaithashing, a two-factor authentication scheme that secures gait features in an efficient manner. The proposed scheme combines the security features of biohash and the recognition capabilities of gait features to provide a high accuracy authentication system. In gaithashing, a user is authenticated only if he/she possesses a valid token and a valid gait feature. The performance of the gaithashing scheme is evaluated by carrying out two sets of experiments. The obtained numerical results and the carried out evaluation allow us to derive the following generic observations:

- Gaithashing achieves EER=0% for type 1 and 3 impostors (i.e., type 1 impostor uses his/her own gait features and his/her own token, while type 3 impostors use compromised gait features and they own token for authentication). This means that the proposed scheme always detects type 1 and 3 impostors.
- It achieves very high accuracy (EER=10.8%) for type 2 impostors (i.e., an impostor that uses a compromised token and his/her own gait features for authentication).
- Gaithashing addresses the distortions caused when the subject wears a coat or holds a bag, by enrolling three different types of human silhouettes (i.e., straight, coat, bag). The proposed scheme can be easily extended to take into account other types of human silhouettes (e.g., a user wearing a hat).
- The proposed scheme secures gait features by converting them to non-invertible bitstreams using the biohash algorithm and a user's token.
- Gaithashing provides unlinkability and easy revocability of the gait templates, simply by replacing the user's token with a new one.

4.2. Detection of malicious actions using machine learning

4.2.1. Background

4.2.1.1. Routing in mesh networks

AODV is an on demand routing protocol, which maintains routes as long they are needed by source nodes. It is scalable and offers low processing, memory, and communication overheads to the underlying network. It utilizes three control messages to achieve route discovery: route request (RREQ), route reply (RREP), and route error (RERR). It also provides an optional fourth control message (i.e., Hello message), which is used for preserving connectivity between neighboring nodes. Each node maintains a list of previously established routing paths in a routing table. Each entry in this table stores routing information to a destination node in the network. The most essential fields of a routing table entry are:

- Destination IP address (dst): the IP address of the destination node.
- Destination SQN (denoted as $SQN_{dst\_node\_entry}$): this is the latest SQN of the destination node of the entry. This field can be updated during the route discovery process. The destination SQN is a measure of the freshness of the routing information in the related entry.

- Hop count (hop_count): represents the current distance to the destination node of the entry.
- Next hop node (next_hop): all packets sent to the destination node of the entry should be forwarded through this node.

When a source node S wishes to transmit a data packet to some destination D for which it does not possess a route, it initiates a route discovery process by first incrementing its own SQN by one, and, subsequently, broadcasting a RREQ message that includes the: source IP address, source SQN, destination IP address, destination SQN, RREQ id, and hop count field. The value of the destination SQN in the RREQ message (the values of destination SQNs in the AODV messages are denoted as SQN_{dst_node}) is taken from the related routing table entry of the source node for the specific destination that wishes to discover a route. The intermediate node that receive the RREQ first create a routing table entry for the source node S. Then, it checks the routing table for a route to the destination node D. If it possesses a fresh route to the destination (i.e., the $SQN_{dst\_node\_entry}$ in its corresponding routing table entry is greater than or equal to the SQN_{dst_node} included in the RREQ message), then it responds to the source node with a route reply (RREP) that includes: the hop count to the destination, the destination IP address, the destination SQN, and the source IP address (i.e., the address of the node that initiated the route request). The value of the destination SQN (i.e., SQN_{dst_node}) is taken from the stored in the intermediate nodes' routing table. Otherwise, (i.e., if the $SQN_{dst\_node\_entry}$ in the intermediate nodes' routing table entry is less than the SQN_{dst_node} included in the RREQ message or there is no route to the destination at all), then the intermediate node increments the hop count field by one and forwards the RREQ to its neighbors.

If none of the intermediate nodes possesses a fresh route to the destination, then the RREQ eventually reaches the destination node. In this case, the destination node increases its own SQN by one (if the incremented value equals the value in the RREQ message) and then sends a RREP message to the source node S that contains the: source IP address, destination IP address, destination SQN, and hop count field. The destination SQN (i.e., SQN_{dst_node}) in the RREP message is equal to the value of the destination node's own SQN. Intermediate nodes receiving the RREP update their routing tables, only, if the destination SQN_{dst_node} in the message is higher from the stored value in their routing tables (i.e., $SQN_{dst\_node\_entry}$), or the destination SQNs are

equal, but the hop count field in the RREP is smaller than the stored value. If multiple RREP messages reach the source node (i.e., this may occur when several intermediate nodes have a routing path to the destination node), it accepts the RREP with the highest destination SQN value or, in case these values are equal, the RREP with the smallest number of hops to the destination. If a link breaks, an intermediate node initiates a local repair mechanism attempting to discover a new route to the destination, by transmitting a RREQ message. If the repair mechanism fails to discover a route, the node generates a RERR message that includes the IP addresses and the last known destination SQNs of the unreachable destinations, informing the receiving nodes that they should restart the routing discovery process, if they want to communicate with them.

4.2.1.2. Blackhole attack: Acting as a sinkhole for all network traffic

The blackhole attack is a type of denial-of-service attack in which a malicious node falsely claims to possess a fresh route to the destination, in order to attract network traffic, and, subsequently, drops all data packets that are forwarded to it. In a more advanced variation of the attack, the malicious node may even selectively drop a percentage of packets (instead of all), in order to avoid detection. This variation is often referred as greyhole attack [136]. The implementation of the attack can be achieved in two ways, which we refer as "reactive" and "proactive". In the "reactive" version of the attack, a malicious node awaits for RREQ messages. When it receives an RREQ, then it responds to the source node with a spurious RREP message that includes a fake destination SQN (i.e., $SQN_{malicious}$) of an arbitrarily high value. Upon receiving the fake RREP message, the source node compares the $SQN_{malicious}$ value with the SQN values of any other received RREP messages, and, since $SQN_{malicious}$ has the highest value; the source node selects the malicious node as its path to the destination. Subsequently, the source node begins the transmission of data through the malicious node.

In the "proactive" version of the attack, a malicious node actively generates fake RREQ messages, masquerading as an intermediate node forwarding a RREQ message. First, it selects a random source and destination address and then, it generates and transmits a RREQ message that includes a fake source SQN of arbitrarily high value. Upon receiving the fake RREQ message, intermediate nodes add the malicious node as a path to the destination. Subsequently, when they have data to transmit to the destination, they select the malicious node as a path to the destination. The "proactive" version of the attack can yield more captured traffic for the malicious node, since: (i) the later does

not have to wait for RREQ messages in order to advertise its spurious path to the destination; and (ii) it enables the malicious node to actively advertise a path to any destination, contrary to the "reactive" version of the attack, where the malicious node is limited to the destinations from which a RREQ message is received.

On the other hand, detecting the "proactive" version of the attack can be implemented using a simple mechanism that takes advantage of the AODV operation. This detection mechanism should run in every node and simply check if a received RREQ message was actually generated and transmitted by the host node itself. In particular, according to the AODV protocol specifications, when a node on the network receives a RREQ message, it compares the source IP address and RREQ id with any values stored in its buffer, in order to avoid processing RREQ messages that have already been processed or that have been transmitted by itself [137]. If no matching values are found (i.e., the RREQ message is new to the host node) then the detection mechanism checks if the source IP address on the RREQ message matches the IP address of the host node. If the two IP address values match, then the RREQ message has been generated by a malicious node (even though the host node is listed as the source in the RREQ message's header) and, thus, a “proactive” blackhole attack has been detected. Consequently, as we have shown, the “proactive” version of the attack can be detected by intelligently performing only one additional comparison by the detection mechanism, thus inducing insignificant computational overhead to the host node. For this reason, throughout the remainder of this section, we focus on the "reactive" version of the blackhole attack.

To better understand the functionality of a "reactive" blackhole attack, we provide a numerical example that presents all of the steps taken by a malicious node. Figure 20 shows a network of six nodes. Node S denotes the source node, node D the destination node, nodes I_1, I_2, I_3 are intermediate nodes; while node M is the malicious node performing a blackhole attack. When node S wants to transmit data to the destination, it first checks for a valid route to its routing table. Since no such route exists, node S generates a RREQ message (with parameters dst = D, $SQN_{dst\_node} = 0$) and transmits it to its neighboring nodes I_1 and I_2, (see Figure 20, step a). These nodes do not possess a route to the destination yet either, so the RREQ message is subsequently forwarded, and, finally, it reaches both the malicious node M and the destination node D.

Upon the reception of the RREQ message, the malicious node M, generates a RREP message (even though it does not possess a route to the destination node D), using as a

destination SQN_{dst_node} (which is denoted as $SQN_{malicious}$) an arbitrarily high value, 1000 in our example, as well as a fake hop_count = 1, and transmits the message to the next hop (i.e., node I_1) towards the source node S (see Figure 20, step b). The intermediate node I_1 that receives the RREP message generated by the malicious node M; creates a new route table entry for the destination, in which it stores the destination address (dst), next_hop, hop_count incremented by one, and the fake $SQN_{malicious}$ value from the RREP message to its $SQN_{dst\_node\_entry}$ field. Subsequently, it updates the received RREP message with the incremented hop_count and with the next_hop field set equal to its own address. Finally, it forwards the RREP message towards the source node S. When the source node S receives the RREP message, it creates a new route table entry for the destination, in which it stores the destination address (dst), next_hop, hop_count incremented by one, and the fake $SQN_{malicious}$ value from the RREP message to the $SQN_{dst\_node\_entry}$ field.

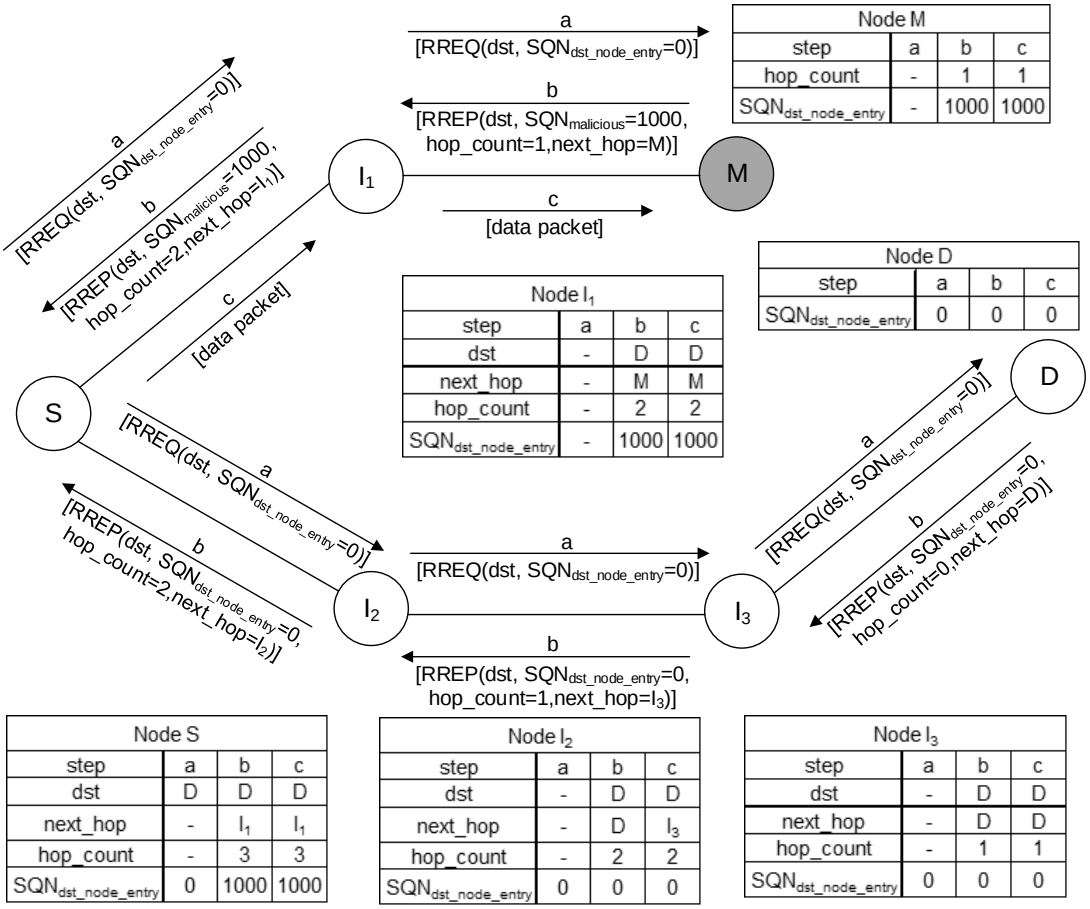

Figure 20: The "reactive" blackhole attack (step a: route request, step b: route reply, step c: data transmission)

On the other hand, the destination node D generates a RREP message (with parameters SQN_{dst_node} = 0, hop_count = 0) and transmits it to the next hop (i.e., node I_3) towards the source node S (see Figure 20, step b). Each of the intermediate nodes (i.e., I_3 and I_2) that receive the RREP message generated by the destination node D, create a new route table entry for the destination, in which they store the destination address (dst), next_hop, hop_count incremented by one, and SQN_{dst_node} value from the RREP message. Subsequently, they update the received RREP message with the incremented hop_count and with the next_hop field set equal to their own address. Finally, they forward the RREP message towards the source node S. When the source node S receives the RREP message generated by the destination node D, it compares the SQN value between the entry stored in the route table (i.e., $SQN_{dst\_node\_entry}$) and the value in the RREP message (i.e., SQN_{dst_node}) and, since the later contains a lower value, the RREP message is discarded.

Once the route discovery process is completed, the source node S looks up its route table for the next_hop node of destination D (i.e., node I_1) and transmits a data packet to it (see Figure 20, step c). Subsequently, node I_1 receives the data packet and checks if the packet is addressed for itself. Since the data packet destination field indicates that the message's destination is node D, node I_1 looks up its route table for the next_hop node of destination D (i.e., node M) and forwards the data packet to it. Finally, once the malicious node M receives the data packet, it can perform one of the two possible actions: it either (i) arbitrarily drops the data packet, or (ii) selectively drops the packet based on a percentage of target packet drops.

4.2.1.3. Related Work

The blackhole attack has been repeatedly analyzed in the literature. In [138], the authors provide an overview of routing attacks that target MANETs, including the blackhole attack. Furthermore, the authors survey several detection mechanisms that attempt to address blackhole attacks and outline their strengths and weaknesses. [139], [140], and [141], have conducted a comprehensive set of simulations that illustrate the effects of a blackhole attack to the AODV routing protocol. In particular, the authors focus on the second part of the attack (i.e., packet drop) and evaluate its impact to the packet delivery rate of the network, the end-to-end delay, as well as the throughput, under various mobility scenarios. However, none of these works provide any insights regarding the first step of the attack, the related routing parameters that are exploited by a malicious

node, or how these parameters affect the attack itself (i.e., such as the percentage of routes won by a malicious node).

A variety of detection mechanisms for blackhole attacks in AODV also exists in the literature and even though we provide an evaluation of the most recently proposed solutions, a comprehensive analysis of all the related literature requires an extensive review, which is outside the scope of this thesis. In [142], a distributed cooperative mechanism (DCM) is proposed to resolve blackhole attacks, by monitoring data packets transmitted by neighboring nodes. If a node has not routed any data packets during a fixed time-threshold, then the monitoring node will transmit a "test packet" through the suspicious node, destined for another cooperating detection node. If the later receives the "test packet," then the suspicious node is legitimate; otherwise, it is considered malicious. The primary disadvantage of this scheme is that malicious nodes may attempt to exploit this mechanism, by analyzing the duration of time before a malicious node is detected (i.e., estimate the threshold value), and subsequently, the routing of at least one packet within this time-frame (i.e., selective drop).

To address the limitation of [142], [143]proposes the use of a dynamically updated normal profile. In this scheme, the normal profile is updated dynamically, using monitored data collected during a period of time in which no malicious behavior was detected. It utilizes a support vector machine classifier (SVM) for detecting an attack by monitoring the delay between data transmissions. Although the use of dynamic profiles may reduce the rate of false positives in volatile networks; on the other hand, by relying on data transmissions for detection, attacks in which data packets are selectively dropped, remain undetected.

In [144], the authors propose a mechanism to detect blackhole attacks by checking if the SQN of a RREP message is higher than a dynamic threshold value, which is an indication of a blackhole attack. The value of the threshold is updated by calculating the difference between the SQNs of the RREP message and the average of the previously received SQNs. However, in case of high mobility, the exchanged routing information is greatly increased (i.e., caused by link breakages), resulting in an unexpected increase in the SQNs of control packets, and thus, leading to considerably high false alarms. Moreover, the proposed solution requires many significant modifications to the AODV protocol.

In [145], the authors propose a reputation scheme called Prevention of Cooperative Black-Hole Attacks (i.e., PCBHA). In this scheme, each node maintains reputation scores for the other nodes of the network and when a route is required, the source node selects the route that includes intermediate nodes with the highest reputation scores. The carried out simulation results show that the performance of the AODV protocol is not deteriorated, considerably, using the proposed solution. However, the reputation information exchanged between nodes results in additional communication overhead and the proposed scheme is vulnerable to byzantine attacks, since a colluding group of malicious nodes may exploit the proposed scheme by providing fake reputation values that are high.

A modified version of AODV, referred as the Gratuitous-AODV (i.e., GAODV), has been proposed in [146], in order to address the issue of blackhole attacks. In GAODV, when a source node receives a RREP from an intermediate node, it sends a verification message to the destination node. The latter should also provide an acknowledgment message to the source node. If the source node does not receive the acknowledgment, then the intermediate node is considered malicious and thus, the advertised routing path is not used. However, the functionality of GAODV requires extensive modifications to the original AODV protocol, raising compatibility issues and it introduces considerable delay in the route discovery process.

Finally, in [147], the authors propose a detection mechanism called the Anti-Blackhole Mechanism (i.e., ABM), which captures both RREQ and their corresponding RREP messages and, subsequently, estimates the difference between the two. When this difference exceeds a predefined threshold, an alarm is raised informing all nodes on the network to cooperatively isolate the malicious node. ABM requires each node to run in promiscuous mode in order to capture, store, and, subsequently, process the RREQ and RREP messages within their radio range. Consequently, monitoring nodes are hindered with computational and storage overheads, as well as increased energy consumption. In addition, during the collection of captured traffic, malicious activities are not detected (i.e., non-real-time detection). The functionality of ABM also requires the operation of a modified version of the AODV protocol (i.e., MAODV), raising compatibility issues with the AODV protocol.

In summary, existing detection mechanisms are limited in the sense that their deployment requires significant modifications to the AODV protocol [146] [147], while

some of the proposed solutions add considerable performance delays and communication overheads [142] [145] [146]. Even more importantly, the majority of these mechanisms attempt to resolve if a blackhole attack takes place, based only on the second step of the attack (i.e., packet drop) [142] [143] [145] [146]. Thus, they do not completely mitigate the attack (since detection can only be achieved after the malicious node wins the route discovery process), and they are effective, only, when the malicious node indiscriminately drops all of the forwarded traffic. On the other hand, our proposed detection mechanism is capable of detecting a blackhole attack during its first step (i.e., during the exploitation of the route discovery process), limiting the ability of a malicious node to drop packets, and thus, induce damage onto the network. Furthermore, by disassociating the detection of an attack from packet drop monitoring, the proposed detection mechanism is capable of detecting not only the blackhole attack but also the greyhole, in which a malicious node selectively drops packets, in order to avoid detection, in which a malicious node might selectively drop packets, in order to avoid detection. Finally, the proposed mechanism alleviates any associated communication overheads and does not require any modifications to the existing AODV routing protocol.

4.2.2. Blackhole attack intensity

In a blackhole attack, the objective of a malicious node is to attract as much traffic as possible, in order to maximize the number of packets that can be dropped, when legitimate source nodes transmit data. This is achieved during the first step of the attack, in which the malicious node provides a fake SQN (i.e., denoted as $SQN_{malicious}$) greater than all other SQN values provided by legitimate nodes, and, thus, wins all the received route requests. This can be clearly seen in the example of section 4.2.1.2; at step b. Furthermore, the parameter $SQN_{malicious}$ affects not only the source node that initiated the route request, but also all intermediate nodes (such as node I_1 in the example) that stored this parameter in their routing tables. However, the malicious node cannot discern what the current values are for the SQNs of other nodes. Thus, it must increment the SQN with a value high enough, to overcome legitimate nodes competing for the route discovery process (i.e., nodes I_2 and I_3 in the example). We define this increment as the *blackhole intensity* parameter or parameter L for short. Let $SQN_{malicious}$ be equal to the destination SQN in the RREP message (i.e., SQN_{dst_node}), incremented by a value L. That is,

$$SQN_{malicious} = SQN_{dst\_node} + L, L \geq 0 \quad (1)$$

Evidently, the value of the destination SQN_{dst_node} in the RREP message will be selected by the attacker so that to be the highest between the destination SQN received in the RREQ message and the one stored in its routing table (if it has a stored one). In the example presented in section 4.2.1.2, the malicious node increments SQN_{dst_node} by a blackhole intensity parameter value equal to 1000. The blackhole intensity parameter plays a crucial role to the success of the attack, because it determines whether or not the malicious node will win a route request, and thus, attract traffic. However, there is no indication as to what values this parameter should hold, and how this affects the outcome of the attack. For example, if the malicious node selects a relatively "small" value for L, then the malicious node might not win all of the route requests. This result might be further exacerbated under different network conditions. In particular, a higher number of traffic will lead to higher SQN values for competing legitimate nodes, and thus, even less route request wins for the malicious node. On the other hand, selecting a relatively "high" value for L may be counterproductive, because after some threshold, the malicious node will be wining all of the received route requests, and thus, higher values of L yield no further benefit. Moreover, since our goal is to utilize SQNs for detection, there is an additional incentive for the attacker to use the lowest values of L possible, in order to hinder the ability of a detection mechanism to distinguish its malicious activity. In order to accurately quantify the impact of the blackhole intensity parameter, we have conducted a comprehensive set of simulations that are presented in the following section.

4.2.3. Using machine learning to detect malicious actions

In this section, we analyze and evaluate a novel blackhole detection mechanism that is capable of detecting blackhole attacks during their first step. Particularly, we provide an architectural overview of the proposed detection mechanism, we identify the computational overhead associated with the operation of the proposed mechanism, and we comparatively evaluate the performance of the proposed mechanism through an extensive set of simulations. The proposed mechanism uses a non-parametric version of the Cumulative Sum (CUSUM) test [148], with the goal of detecting abrupt changes in the normal behavior of SQNs, caused by the occurrence of blackhole attacks. Two variants of this mechanism are presented, depending on the type of threshold used (i.e., static or dynamic). The CUSUM test is a suitable solution for infrastructure-less

networks, since, it does not impose significant computational overheads [149] [150], meaning that the performance of the AODV protocol is not deteriorated. Moreover, it is insensitive to traffic patterns with unknown distribution, making the detection mechanism generally applicable, regardless of the employed application-layer protocols. Another advantage of using the CUSUM test is related to the fact that, given an appropriate threshold value, it detects the attack at the earliest possible time while maintaining a low percentage of false positives. It is evident that a fast detection mitigates the impact of blackhole attacks, because it limits the ability of an attacker to drop packets.

Architecture of the proposed detection mechanism

In the proposed scheme, each network node executes an instance of the detection mechanism, which relies solely on local audit data (i.e., there is no cooperation between nodes). Each of these instances, can be implemented at the application or routing layer of a device, alleviating the need for any AODV protocol modifications. During their execution, they passively monitor the SQN parameter values stored in the nodes' routing table, and, at predefined time intervals, run the CUSUM test, in order to determine if a blackhole attack takes place. More specifically, in case of a Linux based device, we have identified three different implementation options [151], [152]: i) sniffing, in which the node will promiscuously sniff all incoming packets on a network interface (the code to perform sniffing is built into the kernel and is available to user-space programs by using the Packet Capture Library (libpcap)); ii) kernel modifications, using either patches (low portability – low complexity solution) or recompilation of the whole kernel (high portability – high complexity solution); iii) Netfilter, which is a packet filtering framework implemented as a set of hooks at well-defined places in the Linux TCP/IP networking stack. The CUSUM test is a change point detection algorithm, which evaluates the statistical distribution of SQNs prior to change and after, and subsequently, raises an alarm if the difference between the two exceeds some threshold. The later can be either dynamic (i.e., dynamic threshold CUSUM) or static (i.e., standard CUSUM). In this analysis, both threshold variants are elaborated, and, subsequently, the most suitable threshold mechanism is selected, by comparatively evaluating the detection accuracy and the rate of false positives between the two. The detection mechanism calculates the statistical distribution of SQNs based on the monitoring feature $SQN_{total\_rate\_i}(t)$ (see eq. 2). Formally, for some node i

executing an instance of the detection mechanism, we define this monitoring feature as the rate of increase for the sum of the SQNs included in the node's i routing table:

$$SQN_{total\_rate\_i}(t) = \frac{\left(\sum_{j=1}^{K} SQN_{routing\_table\_i\_j}(t)\right) + SQN_i(t)}{t} \qquad (2),$$

where $SQN_{routing\_table\_i\_j}(t)$ is the SQN value at time t of node j stored in the routing table of node i. K is the total number of entries in the routing table of node i, while $SQN_i(t)$ is the value of SQN of node i at time t.

At network initialisation, the CUSUM algorithm requires an initial statistical distribution of SQNs to compare to. As a result, two phases are incorporated into the detection mechanism, a training phase and a normal phase. We assume that during training, no attack takes place (i.e., training can be performed in a controlled environment), while during the normal phase, any node on the network can perform a blackhole attack. Furthermore, in both phases, the CUSUM algorithm is executed at a predefined, time interval. Since the detection of an attack requires the execution of the CUSUM algorithm, this time interval represents the detection time of the proposed mechanism. Therefore, it would seem practical to keep the time interval at the lowest possible value so that attacks are resolved quickly. However, this interval has an associated tradeoff: lower values produce more frequent executions of the detection mechanism, and, consequently, higher induced overhead. Larger values, on the other hand, may lead to: (a) the calibration of an outdated threshold value, resulting in a higher percentage of false positives, and (b) a greater percentage of packets dropped by the malicious node. Thus, the most optimal time interval is the largest possible value that produces the least amount of false positives and packets dropped. In through simulations, we identify the most optimal time interval value.

Training phase

During the training phase, at each time interval, the CUSUM algorithm first calculates a random sequence X_n which we define as the difference between two successive sampling values of the monitoring feature $SQN_{total\_rate\_i}(t)$. That is,

$$X_n = SQN_{total\_rate\_i}(n) - SQN_{total\_rate\_i}(n-1), X_0 = 0 \qquad (3).$$

Next, the CUSUM test transforms X_n to another random sequence Z_n such as:

$$Z_n = X_n - C, C \in R \quad (4),$$

where C is a constant variable that is equal to the upper bound of the mean value $E[X_n]$. The CUSUM algorithm also requires the calculation of a random sequence Y_n that represents the cumulative sum of the positive values of Z_n. Y_n is defined as the maximum value between zero and $Y_{n-1} + Z_n$. That is:

$$Y_n = max(0, Y_{n-1} + Z_n), \text{ where } n \in \mathrm{N} \text{ and } Y_0 = 0 \quad (5).$$

The value of the threshold N is computed at the end of the training phase by each node. Its value is equal to the mean value of the n samples of X_n. That is,

$$N = E[X_n] \quad (6).$$

The selection of threshold *N* regulates the following intrinsic tradeoff: having a relatively "small" threshold may lead to a high percentage of false positives, since even legitimate increases in the statistical distribution of SQNs will lead to false alarms, while, on the other hand, having a relatively "high" threshold may lead to false negatives, since increments to the SQN by a malicious node may not exceed the threshold, and, therefore, the attack will not be detected. We have based the selection of threshold *N* on previous literature [153] [154], in which it yielded the most optimal results in terms of false positives/negatives.

Normal phase

During the normal phase, at each time interval, the CUSUM algorithm calculates all three random sequences X_n, Z_n, Y_n. It then uses the random sequence Y_n and the threshold N to detect blackhole attacks. In particular, the detection is based on the following simple rule: if at any time interval n, the random sequence Y_n exceeds the threshold N (i.e., $Y_n > N$), then a blackhole attack is detected and an alarm is raised to inform other nodes on the network. Finally, in the dynamic threshold variant of CUSUM, for each time interval in which an attack is not detected, the threshold N is also recalculated, to a value equal to the mean of X_n, X_{n-1} (7). Figure 21 summarises the operation of the CUSUM algorithm during both phases.

$$N = E[X_n, X_{n-1}] \quad (7).$$

CUSUM Algorithm

```
Input1: K   // Number of routing table entries
Input2: is_dynamic  // Boolean indicating the type of CUSUM (if TRUE
then CUSUM is dynamic)
01: set Y_0=0, n=1;
02: while Training
03:   compute Xn,C,Zn;
04:   if Y_(n-1)+Zn>0 then
05:     Yn=Y(n-1)+Zn;
06:   else
07:     Yn=0;
07: compute N;
09: while Detection
10:   compute Xn,C,Zn;
11:     if Y(n-1)+Zn>0 then
12:       Yn=Y(n-1)+Zn;
13:     else
14:       Yn=0;
15:     if Yn>N then
16:       raise an alarm
17:     else
18:       if is_dynamic = TRUE then
19:         compute N;
20:   n=n+1;
```

Figure 21: Pseudocode of the CUSUM algorithm

4.2.4. Results and discussion

Section 4.2 of this thesis provided a comprehensive analysis of the blackhole attack, identified a new critical attack parameter (i.e., blackhole intensity), and evaluated the impact of that parameter to the performance of the attack, through an extensive set of simulations. Based on the results of the simulations, we identified a quantitative relation between SQNs and blackhole attacks. This outcome led to the proposal of a novel detection mechanism, which utilizes a dynamic threshold cumulative sum (CUSUM) test to detect abrupt changes in the normal behavior of SQNs.

5. Conclusions

In this thesis, we have addressed the problem of user authentication in online services by holistically investigating the users' security both on server and client side. Particularly, we examined the security of online user accounts by proposing a framework that allows us to quantify the cost time of password guessing both for brute force and dictionary attacks. We also identified the default hashing schemes of various CMS and web applications frameworks and concluded that the majority of CMS and web applications frameworks do not offer secure default settings for password storage. 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. Finally, we provided a set of best practices and alternative solutions to enhance the security of password storage. Based on our analysis we advocate that password hashing standards should be updated to require and not merely suggest the use of new secure functions, such as memory hard hash functions.

Knowing that passwords are one of the weakest links in user security, this thesis investigates the security of FIDO UAF protocol, which provides strong authentication and a simplified registration and authentication procedure. However, the critical functionality of the UAF protocol typically operates in a consumer platform such as a mobile device, which is susceptible to a variety of attacks such as malware and viruses. Based on a comprehensive security analysis, we have identified several vulnerabilities that may be exploited by an attacker in order to compromise the authenticity, privacy, availability, and integrity of the UAF protocol. Regarding volatile memory protection, we have also investigated techniques that can be applied at the software level either form the OS or the applications to protect the user's passwords in the volatile memory. Particularly, we discovered that Windows use built-in safeguards to protect against memory disclosure attacks by deleting the volatile memory contents after the termination of a process. It is important to note that most Linux distributions do not have such safeguards. Lastly, we proposed software functions and techniques in C/C++ programming language that can be used by developers to protect the data in the volatile memory of their applications.

Lastly, this thesis proposes two solutions for continuous authentication and detection of malicious actions via the use of biometrics and machine learning. The first, gaithashing, is a two-factor authentication scheme that secures gait features in an

efficient manner. The performance of the gaithashing scheme achieves EER=0% for type 1 and 3 impostors (i.e., type 1 impostor uses his/her own gait features and his/her own token, while type 3 impostors use compromised gait features and they own token for authentication). It also achieves very high accuracy (EER=10.8%) for type 2 impostors (i.e., an impostor that uses a compromised token and his/her own gait features for authentication). The second, performs a comprehensive analysis of the blackhole attack. As a result, a new critical attack parameter is identified (i.e., blackhole intensity), which quantifies the relation between AODV's sequence number parameter and the performance of blackhole attacks.

5.1. Publications

The contribution of this thesis can be found in the following per-reviewed conference proceedings and journals.

5.1.1. Journal Articles

- Christoforos Ntantogian, Stefanos Malliaros, Christos Xenakis, "Gaithashing: a two-factor authentication scheme based on gait features," Computers & Security, Elsevier Science, Vol. 52, Issue 1, pp: 17-32, July. 2015.
- Christoforos Panos, Christoforos Ntantogian, Stefanos Malliaros, Christos Xenakis, "Analyzing, quantifying, and detecting the blackhole attack in infrastructure-less networks," Computer Networks, Elsevier Science, Vol. 113, Issue 1, pp: 94-110, February 2017.
- Christoforos Ntantogian, Stefanos Malliaros, Christos Xenakis, " Evaluation of Password Hashing Schemes in Open Source Web Platforms", Computer & Security, Elsevier Science, [Under review]

5.1.2. Conference/Workshop Publications

- Stefanos Malliaros, Christoforos Ntantogian, Christos Xenakis, " Protecting sensitive information in the volatile memory from disclosure attacks, " In Proc. 11th International Conference on Availability, Reliability and Security (ARES 2016), Salzburg, Austria, August 2016.
- Christoforos Panos, Stefanos Malliaros, Christoforos Ntantogian, Angeliki Panou, Christos Xenakis, " A Security Evaluation of FIDO's UAF Protocol in Mobile and Embedded Devices, " In Proc. Towards a Smart and Secure Future Internet: 28th International Tyrrhenian Workshop (TIWDC), Palermo, Italy, Sept. 2017.

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

[7] 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.

[8] 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].

[9] 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.

[10] 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.

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

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

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

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

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

[16] 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.

[17] 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.

[18] 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.

[19] "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].

[20] 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].

[21] 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.

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

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

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

[25] 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].

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

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

[28] 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..

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

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

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

[32] 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.

[33] 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.

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

[35] 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.

[36] 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.

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

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

[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] 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.

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

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

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

[49] 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.

[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: http://www.fidoalliance.org/specifications. [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] "FIDO Certified Products," F.I.D.O. Alliance,, [Online]. Available: https://fidoalliance.org/certification/fido-certified-products/. [Accessed June 2017].

[63] "FIDO UAF Protocol Specification v1.1: FIDO Alliance Proposed Standard.," F.I.D.O. Alliance. [Online]. [Accessed 2016].

[64] C. Panos, C. Xenakis, P. Kotzias and I. Stavrakakis, "A specification-based intrusion detection engine for infrastructure-less networks," *Computer Communications*, vol. 54, no. C, pp. 67-83, 2014.

[65] "TCPA main specification v. 1.2," Trusted Computing Platform Alliance, [Online]. Available: http://www.trustedcomputing.org.

[66] J. Winter, "Trusted computing building blocks for embedded linux-based ARM trustzone platforms," in *Proceedings of the 3rd ACM workshop on Scalable trusted compu-ting*, 2008.

[67] "SAMSUNG SDS FIDO Server Solution V1.1 - Certification Report," 2015. [Online]. Available: https://www.commoncriteriaportal.org/files/epfiles/KECS-CR-15-73%20SAMSUNG%20SDS%20FIDO%20Server%20Solution%20V1.1(eng).pdf.

[68] C. Helrmeier, D. Nedospasov, C. Tarnovsky, J. S. Krissler, C. Boit and J. P. Peirfert, "Breaking and entering through the silicon," *Computer and Communications Security (CCS)*, pp. 733-744, 2013.

[69] T. Cooijmans, J. Ruiter and E. Poll, "Analysis of secure key storage solutions on Android," in *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices. ACM*, 2014.

[70] S. Di, "Exploiting Trustzone on Android," in *Black Hat US*, 2015.

[71] D. Rosenberg, "Qsee trustzone kernel integer over flow vulnerability," in *Black Hat Conference*, 2014.

[72] T. Cooijmans, "Secure key storage and secure computation in Android," Redboud University, Nijmegen.

[73] D. Lucas, A. Dmitrienko, A.-R. Sadeghi and M. Winandy , "Privilege escalation attacks on android," in *International Conference on Information Security*, 2010.

[74] P. C. Abhishek, "Student Research Abstract: Analysing the Vulnerability Exploitation in Android with device-mapper-verity (dm-verity)," 2017. [Online].

[75] D. Thom and M. Marse, "Subverting Android 6.0 fingerprint authentication," 2016. [Online].

[76] F. Alliance, "FIDO security reference," 2014. [Online]. Available: https://www.fidoalliance.org/specifications.

[77] Q. Darren and R. C. Kim-Kwang, "Dropbox analysis: Data remnents on user machines," *Digital Investigation,* vol. 10, no. 1, pp. 3-19, 2013.

[78] Q. Darren and R. C. Kim-Kwang, "Digital droplets: Microsoft SkyDrive forensic data remnants," *Future Generation Computer Systems,* vol. 29, no. 6, pp. 1378-1394, 2013.

[79] Q. Darrem and R. C. Kim-Kwang, "Google Drive: Forensic analysis of data remnants," *Journal of Network and Computer Applications,* vol. 40, pp. 179-193, 2014.

[80] C. Hyunji, P. Jungheum, L. Sangjin and K. Cheulhoon, "Digital forensic investigation of cloud storage services," *Digital Investigation,* vol. 9, no. 2, pp. 81-95, 2012.

[81] D. Apostolopoulos, G. Marinakis, C. Ntantogian and C. Xenakis, "Discovering Authentication Credentials in Volatile memory of Android Mobile Devices," in *12th IFIP conference on e-business, e-services, e-society (I3E 2013)*, 2013.

[82] J. Sylve, A. Case, L. Marziale and G. G. Richard, "Acquisition and analysis of volatile memory from android devices," *Digital Investigation,* vol. 8, pp. 175-184, 2012.

[83] C. Ntantogian, D. Apostolopoulos, G. Marinakis and C. Xenakis, "Evaluating the privacy of Android mobile applications udner forensic analysis," *Computers & Security,* vol. 42, pp. 66-76, 2014.

[84] X. Chen, R. Dick and A. Choudhary, "Operating System Controlled Processor-Memory Bus Encryption," in *Design, Automation and Test in Europe*, 2008.

[85] P. Peterson, "Technologies for Homeland Security (HST)," in *2010 IEEE International Conference* , 2010.

[86] V. Nagarajan, R. Gupta and A. Krishnaswamy, "Compiler-assisted memory encryption for embedded processors," in *HiPEAC'07 Proceedings of the 2nd international conference on High performance embedded architectures and compilers*, 2007.

[87] Y. Chenyu, B. Rogers, D. Englender, D. Solihin and M. Prvulovic, "Improving Cost, Performance, and Security of Memory Encryption and Authentication," in *Computer Architecture ISCA '06*, 2006.

[88] H. Daeyoung, B. Luis, S. S. Lim and N. Dutt, "DynaPoMP: dynamic policy-driven memory protection for SPM-based embedded systems," in *Proceedings of WESS '11 Proceedings of the Workshop on Embedded Systems Security*, 2011.

[89] B. Rogers, Y. Solihin and M. Prvulovic, "Memory predecryption: Hiding the latency overhead of memory encryption," *ACM SIGARCH Computer Architecture News,* vol. 33, pp. 27-33, 2005.

[90] G. Duc and R. Keryell, "CryptoPage: An Efficient Secure Architecture with Memory Encryption, Integrity and Information Leakage Protection," in *in Computer Security Applications Conference ACSAC '06*, 2006.

[91] D. Lie, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell and M. Horowitz, "Architectural support for copy and tamper resistant software," in *ASPLOS IX Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, 2000.

[92] G. Suh, C. O'Donnell and S. Devadas, "Aegis: A Single-Chip Secure Processor," *Design & Test of Computers*, vol. 24, no. 6, pp. 570-580, 2007.

[93] L. Guan, J. Lin, B. Luo, J. Jin and J. Wang, "Protecting Private Keys against Memory Disclosure Attacks Using Hardware transactional memory," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[94] S. Chhabra and Y. Solihin, "i-NVMM: A secure non-volatile main memory system with incremental encryption," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2011.

[95] Z. Youtao, G. Lan, Y. Jun, Z. Xiangyu and R. Gupta, "SENSS: security enhancement to symmetric shared memory multiprocessors," in *11th International Symposium on High-Performance Computer Architecture, 2005. HPCA-11*, 2005.

[96] S. Weigond, H. Lee, M. Ghosh and L. Chenghuai, "Architectural support for high speed protection of memory integrity and confidentiality in multiprocessor systems," in *roceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, PACT 2004*, 2004.

[97] L. Su, S. Courcambeck, P. Guillemin, C. Schwarz and R. Pacalet, "SecBus: Operating System controlled hierarchical page-based memory bus protection," in *Design, Automation & Test in Europe Conference & Exhibition, DATE '09*, 2009.

[98] M. Russinovich, "Windows Sysinternals, ProcDump v8.0," Technet, [Online]. Available: https://technet.microsoft.com/en-us/sysinternals/dd996900.aspx.

[99] "The Linux Kernel Archives," [Online]. Available: https://www.kernel.org/doc/gorman/html/understand/understand009.html.

[100] "GRSecurity," [Online]. Available: http://grsecurity.net/.

[101] "Atomicorp Linux Distribution," [Online]. Available: https://atomicorp.com/.

[102] "IPFire Linux Distribution," [Online]. Available: http://www.ipfire.org.

[103] "Alpine Linux Distribution," [Online]. Available: http://www.alpinelinux.org.

[104] "Pentoo Linux Distribution," [Online]. Available: http://www.pentoo.ch.

[105] "Hardened Linux Distribution," [Online]. Available: http://hardenedlinux.sourceforge.net.

[106] "Subgraph OS Linux Distribution," [Online]. Available: http://subgraph.com/sgos.

[107] "MSDN, RtlSecureZeroMemory routine," Microsoft, [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/hardware/ff562768%28v=vs.85%29.aspx.

[108] J. Damato, "MSC06-C. Beware of compiler optimizations", Software Engineering Institute – Carnegie Mellon University," [Online]. Available: https://www.securecoding.cert.org/confluence/display/c/MSC06-C.+Beware+of+compiler+optimizations.

[109] T. Plum, "C11: The New C Standard," [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3631.pdf.

[110] S. Guelton, "A glance at compiler internals: Keep my memset," [Online]. Available: http://blog.quarkslab.com/a-glance-at-compiler-internals-keep-my-memset.html.

[111] "Compiler optimization and the volatile keyword," ARMKEIL Microntroller Tools, [Online]. Available: http://www.keil.com/support/man/docs/armcc/armcc_chr1359124222941.htm.

[112] C. Rathgeb and A. Uhl, "A survey on biometric cryptosystems and cancelable biometrics," *EURASIP Journal on Information Security*, pp. 1-25, 2011.

[113] A. T. B. Jin, D. N. C. Ling and A. Goh, "Biohashing: two factor authentication featuring fingerprint data and tokenised random number," *Pattern Recognition,* vol. 37, no. 11, p. 22452255, 5117.

[114] A. Lumini and L. Nanni, "An improved biohashing for human authentication," *Pattern Recognition*, vol. 40, no. 3, p. 10571065, 2007.

[115] A. B. J. Teoh, W. Kuan and S. Lee, "Cancellable biometrics and annotations on biohash," *Pattern Recognition,* vol. 41, no. 6, pp. 2034-2044, 2008.

[116] C. Wang, J. Zhang, L. Wang, J. Pu and X. Yuan, "Human identification using temporal information preserving gait template," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 34, no. 11, pp. 2164-2176, 2012.

[117] H. Hu, "Multi-view gait recognition based on patch distribution feature and uncorrelated multilinear sparse local discriminant canonical correlation analysis," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 24, no. 4, pp. 617-630, 2014.

[118] W. Kusakunniran, Q. Wu, J. Zhang, Y. Ma and H. Li, "A new view-invariant feature for cross-view gait recognition," *IEEE Transactions on Information Forensics and Security,* vol. 8, no. 10, pp. 1642-1653, 2013.

[119] M. Milovanovic, M. Minovic and D. Starcevic, "Walking in colors: Human gait recognition using kinect and cbir," *IEEE Multimedi,* vol. 20, no. 4, pp. 28-36, 2013.

[120] W. Kusakunniran, Q. Wu, J. Zhang and H. Li, "Gait recognition across various walking speeds using higher order shape configuration based on a differential composition model," *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics,* vol. 42, no. 6, pp. 1654-1668, 2012.

[121] S. Sivapalan, D. Chen, S. Denman, S. Sridharan and C. Fookes, "Gait energy volumes and frontal gait recognition using depth images," in *IEEE Intenational Joint Confernece on Biometrics (IJCB '11)*, 2011.

[122] J. Ryu and S. Kamata, "Front view gait recognition using spherical space model with human point clouds," in *18th IEEE Intenational Conference on Image Processing (ICIP)*, 2011.

[123] M. Hu, Y. Wang and Z. Zhang, "Multi-view multi-stance gait identification," in *18th IEEE International Conference on Image Processing (ICIP*, 2011.

[124] M. Mcguire, "An overview of gait analysis and step detection in mobile computing devices," in *IRRR 4th International Confernece on Intelligent Networking and Collaborative Systems (INCoS)*, 2012.

[125] D. Ioannidis, D. Tzovaras, I. G. Damousis, S. Argyropoulos and K. Moustakas, "Gait recognition using compact feature extraction transforms and depth information," *IEEE Transactions on Information Forensics and Security*, vol. 2, no. 3, pp. 623-630, 2007.

[126] T. Hoang and D. Choi, "Secure and privacy enhanced gait authentication on smart phone," *Hindawi, The Scientific World Journal*, 2014.

[127] S. Argyropoulos, D. Tzovaras, D. Ioannidis and M. Strintzis, "A channel coding approach for human authentication from gait sequences," *IEEE Transactions on Information Forensics and Security*, vol. 4, no. 3, pp. 428-440, 2009.

[128] N. Radha and S. Karthikeyan, "An evaluation of fingerprint security using non-invertible bio-hash," *International Journal of Network Security & Its Applications*, vol. 3, no. 4, pp. 118-128, 2011.

[129] A. B. J. Teoh and D. C. L. Ngo, "Cancellable biometrics featuring with tokenised random number," *Pattern Recognition Letters*, vol. 26, no. 10, pp. 1454-1460, 2005.

[130] A. T. B. Jin and T. Connie, "Remarks on biohashing based cancelable biometrics in verification system," *Neurocomputing*, vol. 69, no. 16-18, pp. 2461-2464, 2006.

[131] T. Connie, A. Teoh, M. Goh and D. Ngo, "Palmhashing: a novel approach for dual-factor authentication," *Pattern Analysis and Applications*, vol. 7, no. 3, pp. 255-268, 2004.

[132] R. Fuksis, A. Kadikis and M. Greitans, "Biohashing and fusion of palmprint and palm vein biometric data," in *IEEE International Conference on Hand-Based Biometrics (ICHB)*, 2011.

[133] R. Arun and G. Rohin, "“Feature Level Fusion Using Hand and Face Biometrics," in *Proceedings of SPIE conference on Biometric Technology for Human Identification II*, 2005.

[134] "ISO/IEC TR:24722:2007 Information Technology - Biometrics - Multimodal and other multibiometric fusion".

[135] Z. Huang, Y. Liu, C. Li, M. Yang and L. Chen, "A robust face and ear based multimodal biometric system using sparse representation," *Pattern Recognition,* vol. 46, no. 8, pp. 2156-2168, 2013.

[136] P. V. L. Veeraraghavan, "Trust in mobile ad hoc networks," in *Telecommunications and Malaysia International Conference on Communications, 2007. ICT-MICC 2007*, 2007.

[137] C. Perkins, E. Belding-Royer and S. Das, "Ad hoc On-Demand Distance Vector (AODV) Routing," *IETF RFC 3561,* 2003.

[138] B. Kannhavong, H. Nakayama , Y. Nemoto , N. Kato and A. Jamalipour, "A survey of routing attacks in mobile ad hoc networks," *IEEE Wireless Communications,* vol. 14, no. 5, pp. 85-91, 2007.

[139] A. Bala, B. Munish and S. Jagpreet, "Performance analysis of MANET under blackhole attack," in *Networks and Communications, NETCOM'09*, 2009.

[140] S. Sharma and R. Gupta, "Simulation study of blackhole attack in the mobile ad hoc networks," *Journal of Engineering Science and Technology,* vol. 2, 2009.

[141] E. Barkhodia, S. Parulpreet and G. K. Walia, "Performance analysis of AODV using HTTP traffic under Black Hole Attack in MANET," *Comput. Sci. Eng. Int. J.(CSEIJ) 2,* vol. 3, 2012.

[142] Y. Chang Wu, W. Tung-Kuang, C. Rei Heng and C. Shun Chao, "A Distributed and Cooperative Black Hole Node Detection and Elimination Mechanism for Ad Hoc Networks," in *Emerging Technologies in Knowledge Discovery and Data Mining*, Springer Berlin Heidelberg, 2007, pp. 538-549.

[143] J. F. Joseph , B. S. Lee , A. Das and B. C. Seet, "Cross-Layer Detection of Sinking Behavior in Wireless Ad Hoc Networks Using SVM and FDA," *IEEE Transactions on Dependable and Secure Computing,* vol. 8, no. 2, pp. 233-245, 2011.

[144] P. N. Raj, B. Prashant and B. Swadas, "Dpraodv: A Dynamic Learning System Against Blackhole Attack in AODV Based Manet," in *CoRR abs/0909.2371*, 2009.

[145] D. LathaTamilselvan and V. Sankaranarayanan, "Prevention of Co-operative Black Hole Attack in MANET," *Journal of Networks,* vol. 3, no. 5, pp. 13-20, 2008.

[146] S. K. Dhurandher, I. Woungang, R. Mathur and P. Khurana, "GAODV: A Modified AODV against single and collaborative Black Hole attacks in MANETs," in *27th International Conference on Advanced Information Networking and Application Workshops*, 2013.

[147] M. Y. Su, "Prevention of selective blackhole attacks on mobile ad hoc networks through intrusion detection systems," *Computer Communications,* vol. 34, pp. 107-117, 2011.

[148] M. B. a. I. Nikiforov, Detection of Abrupt Changes: Theory and Application, 1993.

[149] B. B. a. B. Darkhovsky, Nonparametric Methods in Change-Point Problems, Kluwer Academic Publishers, 1993.

[150] T. B. T. Y. C. W. Patrick P. C. Lee, "On the detection of signaling DoS attacks on 3G/WiMax wireless networks," *Computer Networks,* vol. 53, no. 15, pp. 2601-2616, 2009.

[151] R. K. T. P. Gupta, "Design Strategies for AODV Implementation in Linux," *International Journal of Advanced Computer Science and Applications(IJACSA),* vol. 1, no. 6, 2010.

[152] E. M. B.-R. Ian D. Chakeres, "AODV Routing Protocol Implementation Design," in *Proceeding of IEEE 24th International Conference on Distributed Computing Systems Workshops*, 2004.

[153] P. L. C. a. R. K. Tao, "Proactively detecting distributed denial of service attacks using source IP address monitoring," in *International Conference on Research in Networking*, 2004.

[154] V. a. P. F. Siris, "Application of anomaly detection algorithms for detecting SYN flooding attacks," *Computer communications,* vol. 22, no. 9, pp. 1433-1442, 2006.