Security analysis of salt||password hashes

Praveen Gauravaram
Tata Consultancy Services Innovation Labs
Tata Consultancy Services Limited
Hyderabad, India
p.gauravaram@tcs.com

***Abstract*—Protection of passwords used to authenticate computer systems and networks is one of the most important application of cryptographic hash functions. Due to the application of precomputed memory look up attacks such as birthday and dictionary attacks on the hash values of passwords to find passwords, it is usually recommended to apply hash function to the combination of both the salt and password, denoted salt||password, to prevent these attacks.**

In this paper, we present the first security analysis of salt||password hashing application. We show that when hash functions based on the compression functions with easily found fixed points are used to compute the salt||password hashes, these hashes are susceptible to precomputed offline birthday attacks. For example, this attack is applicable to the salt||password hashes computed using the standard hash functions such as MD5, SHA-1, SHA-256 and SHA-512 that are based on the popular Davies-Meyer compression function. This attack exposes a subtle property of this application that although the provision of salt prevents an attacker from finding passwords, salts prefixed to the passwords do not prevent an attacker from doing a precomputed birthday attack to forge an unknown password. In this forgery attack, we demonstrate the possibility of building multiple passwords for an unknown password for the same hash value and salt. Interestingly, password||salt (i.e salts suffixed to the passwords) hashes computed using Davies-Meyer hash functions are not susceptible to this attack, showing the first security gap between the prefix-salt and suffix-salt methods of hashing passwords.

***Keywords*-Cryptography, Computer systems security, Hash functions, Compression functions, Password and Salt.**

I. Introduction

Cryptographic hash functions are the work horses of cryptography and they significantly contribute to the growing demands of secure and efficient digital information processing. Applications of hash functions include (but not limited to) efficient digital signature generation and verification, message integrity, password protection, signcryption mechanism, cryptographic commitment protocols, key derivation functions (KDFs) and message authentication codes (MACs).

A hash function takes as input an arbitrary length message and outputs a fixed length bit string called hash value or message digest. A hash function should possess three fundamental security properties: collision resistance, second preimage resistance and preimage resistance. The requirement of these properties depends on the application in which hash function is used. This implies that an attack on a hash function which contradicts any of these properties could undermine the security of the application which needs that property. For example, a collision attack on a hash function defeats the security of digital signatures and a preimage attack on a hash function compromises the security of applications such as password protection and MACs.

In this paper, we consider the analysis of password protection application of hash functions. For security against precomputed birthday or dictionary attacks, in this application, users' passwords are hashed along with the corresponding salt [12], [28], [29], [11], [30], [31]. It is a common practice to prefix the salt denoted, salt, to the password denoted, password, and process the combination salt||password by using a hash function and store the hash value along with the user name and salt in the computer system's database. We remark that in the earlier operating systems such as UNIX, password hashes were stored in the publicly accessible \etc\passwd file. In the modern Linux operating systems they are stored in \etc\shadow file which requires privileged access [31]. For our analytical purposes, we assume that salted password hashes are stored in the password file and this file is accessible to everyone.

In this paper, we present the first ever security analysis of salt||password hashing application. We show that if it is easy to find *fixed points* for the compression functions of hash functions as for the standard hash functions MD5 [27], SHA-1, SHA-256 and SHA-512 [21], [22] then it is easy to mount birthday attacks on the hash values of the salted passwords even without prior knowledge of the candidate salt. While our attack does not find a password, it helps to forge an unknown password by building several passwords that hash to the same hash value as the unknown password for the same salt. Potential information security applications where this forgery attack can be applied are still unknown although such instances may not be ruled out. For example, our approach might be combined with social engineering practices [1], [28] on finding passwords to further explore applications where our attack could be used. We address some countermeasures that could be employed in practice to thwart our attack and any possible implications.

The rest of the paper is organized as follows: In Section II, we give an overview of the popular and widely

978-0-7695-4959-0/13 $25.00 © 2013 IEEE
DOI 10.1109/ACSAT.2012.49

used hash function and compression function designs. In Section III, we discuss common methods used to protect passwords and discuss their security. In Section IV, we analyse password and salt hashing application when hash functions based on the Davies-Meyer compression functions are used. In Section V we discuss some countermeasures and recommendations. In Section VI, we conclude the paper.

II. Cryptographic hash functions and compression functions

Cryptographic hash functions process an arbitrary length message into hash values of fixed length [18]. We denote a t-bit hash function by $H : \{0,1\}^* \rightarrow \{0,1\}^t$ where t is the size of hash value in bits. For efficiency reasons, hash functions used in practice are iterated designs where in a compression function is iterated throughout the hashing process. A compression function takes as inputs a fixed length message and a state value and outputs a new state value. It is denoted by $f : \{0,1\}^b \times \{0,1\}^n \rightarrow \{0,1\}^n$ where message is of b bits and input and output states are n bits each. An output transformation $g : \{0,1\}^n \rightarrow \{0,1\}^t$ is applied to the output of final compression function to obtain t-bit hash value. An iterated hash function H processes an input message M as follows:

1) *Preprocessing:* Before hash function computation, the input message M is padded with a bit 1 and a sufficient number of 0 bits followed by the binary encoded representation of the length of the unpadded message so that the length of the padded message is a multiple of the block length b of the compression function. Padding the message with the length of the unpadded portion of the message (also called true length of the message) ensures that certain trivial collision attacks [16] do not apply to the hash function. The padded message is denoted by $\mathsf{Pad}(M) = M_1 \| M_2 \| \ldots \| M_N$ where the blocks M_i are of b bits each and the block M_N contains the binary format of the original length of the message.
2) *Compression function iteration:* Each message M_i is processed using the compression function f as defined by $f(H_{i-1}, M_i) = H_i$ where H_{i-1} and H_i are the input and output chaining values (also called states) and H_0 is the initial value (IV) of the hash function. In the paper, we often denote the compression function operation by $f(h, m)$.
3) *Outputting hash value:* The output chaining value H_N of the final compression function is processed by using the output transformation g to obtain the hash value H_t.

Merkle-Damgård [19], [5] is a popular iterated hash function construction whose framework has been used in the design of widely used hash functions such as MD5, SHA-1, SHA-256 and SHA-512. This hash function construction is illustrated in Figure 1. The hashing process for these designs is similar to the above method with the exception that there is no application of the final transformation and the chaining value of the final compression function is the hash value. Hash functions that are designed using Merkle-Damgård framework are vulnerable to "length extension weakness" where it is possible to compute a new hash value from a known hash value and the length of the message from which this previous hash value was computed without knowing the message.

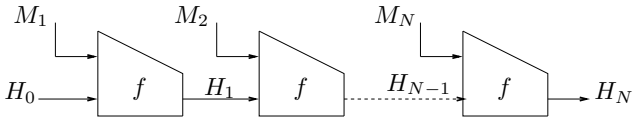

Figure 1. Merkle-Damgård hash function construction

A. Compression functions of some standard hash functions

Many hash function designs proposed in the literature use single block length block cipher based compression functions [26]. Out of sixty-four possible designs, twelve are provably collision resistant and (second) preimage resistant in the ideal cipher model [2], [3]. However, none of these twelve designs behave as a fixed input length random oracle, a random oracle (also called ideal hash function) whose input length is fixed, and are easily differentiable even if the underlying block cipher is ideal [15].

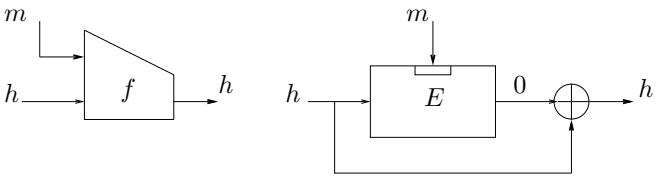

Figure 2. Finding fixed points in the Davies-Meyer compression function

Davies-Meyer is the most popular compression function of these 12 designs and this construction was used in many widely used standard hash functions such as MD5, SHA-1, SHA-256 and SHA-512. The Davies-Meyer compression function f is defined by $f(h, m) = E_m(h) \oplus h$ where E is the block cipher with the message block m as the key and the input chaining value h as the plaintext. In the Davies-Meyer compression function, for every unique message block, it is possible to find a *fixed point* (h, m) such that $f(h, m) = h$. This is possible by finding $h = E_m^{-1}(0)$ as illustrated in Figure 2 and therefore, $f(h, m) = h$. Note that an attacker has no control over the obtained chaining value h. Easily found *fixed points* in compression functions were exploited in several interesting hash function attacks [6], [14], [13], [7], [9], [8], [10] after the idea was proposed by Dean [6].

B. Security properties of hash functions and compression functions

Hash functions are expected to possess the following security properties. We define these properties for H.

1) Collision resistance: It should take $2^{t/2}$ operations of H to find two messages M and N such that $M \neq N$ and $H(M) = H(N)$.
2) Second preimage resistance: For a challenged target message M and its hash value under H, it should take 2^t operations of H to find another message N such that $N \neq M$ and $H(M) = H(N)$. We recall that for a target message of 2^d blocks, a second preimage for a Merkle-Damgård hash function can be found in 2^{t-d} operations of the compression function [14].
3) Preimage resistance: For a challenged target hash value Y, it should take 2^t operations of H to find a message M such that $H(M) = Y$.

Similarly, any compression function f is expected to posses the following properties:

1) Collision resistance: It should take $2^{t/2}$ operations of f to find two different pairs (h, m) and (h^*, m^*) such that $f(h, m) = f(h^*, m^*)$.
2) Second preimage resistance: For a challenged pair (h, m), it should take 2^t operations of f to find a pair (h^*, m^*) such that $(h, m) \neq (h^*, m^*)$ and $f(h, m) = f(h^*, m^*)$.
3) Preimage resistance: For a challenged compression function output Y, it should take 2^t operations of f to find a pair (h, m) such that $f(h, m) = Y$.

In this paper, unless stated otherwise, we consider the analysis of $\mathsf{salt}\|\mathsf{password}$ processed using Merkle-Damgård hash functions based on the Davies-Meyer compression function. This implies that our analysis is applicable to $\mathsf{salt}\|\mathsf{password}$ hashes computed using MD5, SHA-1, SHA-256 and SHA-512 hash functions.

III. Hash functions for password protection

In general, users of computer systems and networks use their user names and passwords to authenticate to these systems. It is a common practice to store hash values of the passwords along with the respective user names in the password file of a computer system. Otherwise, an attacker who can access the password file can see the passwords and use them to access the systems. Storing hash values would prevent an attacker from doing this trivial "password look up".

However, storing hash values of passwords in the password file does not offer protection against some attacks [11], [12], [28], [29], [30], [31] as outlined below:

- An attacker can do a bruteforce attack by guessing all possible combinations of characters from a given character set and for passwords up to a given length [28]. However, this attack is impractical for large character sets.
- An attacker can simply precompute a database of hash values of several randomly chosen passwords, sort them and compare them against the hash values in the password file. Once the database of hash values of passwords is created, it can be used to compare hash values of multiple password files. The complexity of this attack is about $2^{t/2}$ wherein a password file containing $2^{t/2}$ t-bit hash values requires a precomputed database of $2^{t/2}$ hash values of randomly chosen passwords to find at least one password with a probability of at least 50% due to birthday paradox.
- Dictionary attack is a special case of birthday attack where passwords that are likely to succeed such as dictionary words are hashed as part of the precomputation of the attack. The generic complexity of dictionary attacks is at most to that of birthday attack but often much less than this complexity with low success rate. Precomputed birthday and dictionary attacks to find passwords may become efficient when there are large amount of entries in the password files and when there are many such files.
- If the task is to find the password of a specific user, then birthday style attacks do not work; preimage attacks are required whose generic complexity is 2^t to find the password of a specific t-bit hash value.

In general, attempts to find passwords from the hash values are based on the birthday style attacks. Recently, an attack [25] leaked the passwords of 6.5 million LinkedIn subscribers and hash values in the password file were computed using 160-bit SHA-1 hash function. To our knowledge, techniques used for this attack are not available in the public domain. However, one may guess that birthday or dictionary collision attack might have been used in the attack. It is also possible to employ further refined variants of the dictionary attacks such as rainbow tables [24] to reduce the storage requirements of the naive dictionary attack at the expense of a slightly longer look up times[1]. There were also several other instances where attackers can find passwords by mere guessing or by social engineering [1], [28].

To avoid above attacks, it has been recommended to randomize a password with a salt before applying the hash function [12], [28], [30], [31]. That is, when a user enters the user name and $\mathsf{password}$ to authenticate to a system, the $\mathsf{password}$ is prepended with the salt assigned for this user. The password files contain the fields with the information of user name, salt and hash value. The $\mathsf{salt}\|\mathsf{password}$ combination is processed using the hash function H to produce the hash value $H(\mathsf{salt}\|\mathsf{password})$. Once the user enters the $\mathsf{password}$, this hash value together with the salt

[1]Precomputed rainbow tables were also used to attack cryptographic algorithms to compromise the security of applications such as GSM [20].

are compared against the combination of salt and hash value stored in the password file of the computer system. If they match, the user will be successfully authenticated to the system. Even if the size of salt value is reasonably small (e.g 32-bit), the complexity of precomputing the hash values of trail passwords for any salt in the password file is very high and is close to that of preimage attack (i.e 2^t memory is required for t-bit hash values). Moreover, salt values in the password file may repeat for only few $H(\mathsf{salt}\|\mathsf{password})$ hash values and an attacker ends up comparing a large precomputed database (close to 2^t) for each salt against small number of entries in the password file with the same salt value. For salt values of large size (e.g 128-bit salts), the cost of the attack would be very similar to preimage attack as password file hardly contains any repetition of salts. Therefore, it is highly impractical to mount birthday style attacks on $H(\mathsf{salt}\|\mathsf{password})$ hash values in order to find even one password.

IV. Forgery attack on salt||password hash values computed using Davies-Meyer Merkle-Damgård hash functions

In this section, we show that although hashing of salted passwords would complicate to find passwords, they don't prevent offline birthday attacks if it is easy to find *fixed points* for the compression functions. We illustrate this attack by doing a forgery attack on the salt||password hash values of the password file where the password values are unknown. We denote these unknown passwords by X. We assume that salted passwords are hashed with a hash function based on the Davies-Meyer compression function f. In this scenario, independent of the salt value, an attacker can develop a long password whose salted hash value under H would collide with the hash value of salt||X under H. The attack makes a reasonable assumption that the maximum length of the passwords used by all users would be of a fixed multiple of the block length of the compression functions of hash functions after padding (e.g for SHA-1 and SHA-256 the block length is 512 bits) and attacker knows this maximum length. That is, all users use salt||X values that have the same number of bits after applying the conventional padding of the hash functions described in Section II.

The attack works as follows:

1) Find the length of the forgery password to be found so that the exact length padding information can be included in the *fixed point* block. For example, if the salt||X hash value required only one block processing under H then if the attacker wishes to append one more block to it then the length padding in the forgery password represents the binary encoded format of two blocks excluding the padding bits in the second block.
2) Build a database consisting of *fixed point* hash values by processing a list of randomly chosen passwords under the Davies-Meyer compression function f. Let these passwords be denoted password*. This step is similar to the precomputation step of the birthday attack on the password hashes computed without the salt discussed in section III with the exception that this step is applied on the compression function. Usually the block size of the compression functions is large enough so that there is always enough freedom for an attacker to trail his chosen passwords. He fixes the last few bits of the *fixed point* block for padding bits as determined in Step 1 and uses the remaining freedom to trail passwords.
3) Compare the *fixed point* hash values computed in Step 2 with the hash values of $H(\mathsf{salt}\|\mathsf{X})$ values stored in the password file. When there is a match, it means that $H(\mathsf{salt}\|\mathsf{X}) = H(\mathsf{salt}\|\mathsf{X}\|\mathsf{password}^*)$[2]. Therefore, salt||X||password* is the forgery of salt||X where the main password X remains unknown to the attacker.

If the attacker wishes longer password forgeries then he determines the required padding length in Step 1 and fixes it while finding *fixed point* blocks in Step 2. For instance, forgeries that look like salt||X||password*||password*||... can also be constructed for salt||X hash values. Hence, by exploiting the length extension weakness of Merkle-Damgård hash functions together with the ability to easily compute *fixed points* for the compression functions, it is possible to forge salt||password hashes without the knowledge of password values. Note that an attacker does not have to know the salt value before completing the attack and the attack is independent of its value and size and whoever chooses the salt, his/her password will be forged.

A. Some remarks

1) Our attack does not find the password X but would lead to developing an extended password X||password* from the previous unknown password X. However, this property itself shows that salt||password hash values do not offer security against offline birthday attacks. In addition, to falsely authenticate to the computer system it is just enough to know the concealed format of X and attacker can append to it the required padding bits that make X a padded block and *fixed point* password* to authenticate to the system. Note that an attacker can also falsely authenticate to the computer system by only using concealed format of X; this leads to an open question about in which applications the forgery passwords can provide additional benefit that concealed passwords do not.
2) It was noted in [28] that "The inclusion of a random value in the password hashing process that greatly decreases the likelihood of identical passwords returning the same hash." Although this is true as identical

[2]Note that to simplify the notation, we have not represented padding bits next to salt||X and next to salt||X||password*.

passwords would have different salts, our attack shows that it is still possible to show two distinct passwords with an identical salt can return the same hash value. In addition, not setting a limit on the maximum length of the password as noted in [28] does not seem to be a secure practice. That is, having long passwords or passphrases do not necessarily prevent the forgery attack.

3) The attack does not work when the positions of salt and password are interchanged as in password||salt. In this case, the salt chosen by the user would become suffix part of the password and the attack poses a constraint that the attacker must predict this salt to fix in the *fixed point* block and precompute *fixed points* for a collision match with the password||salt hashes in the password hash table. Obviously meeting this constraint increases the complexity of the attack far beyond birthday attack complexity. Moreover, fixing a particular salt value in the *fixed point* block limits the attacker to only try to forge password hashes that have the same salt as his *fixed point* salt. Therefore, the complexity of this approach will be similar to that of finding passwords through the dictionary or birthday attack on the hash values of salt||password or password||salt combinations, which is much more difficult.

V. Some Countermeasures and Recommendations

The following countermeasures not only prevents our attack but also they serve as general guidelines on how to use hash functions for the protection of passwords.

- Use hash functions that truncate hash values either by using an output transformation as done by the wide-pipe hash function [17] construction or by just chopping output bits as done by chop-Merkle-Damgård design [4] (e.g SHA-384 [22] is the chopped variant of SHA-512). These designs thwarts attempts that combine the length extension weakness of Merkle-Damgård and *fixed points* of compression functions. In addition, large internal state sizes of these designs increase the computational complexity of birthday style attacks significantly. Since SHA-3 finalists [23] are secure against length extension attacks, it is recommended to use these designs together with salt for password protection application instead of SHA-1, SHA-256 and SHA-512 which have length extension weakness.
- The process of stretching [28] the password and salt combination is another recommendation for security against our attacks. In this method, the salt and password combinations are processed several times using the hash function. The length-extension weakness of Merkle-Damgård hash functions is destroyed by doing so and hence, our attack is not possible even if it is easy to find *fixed points* for the compression function.
- Proposing a password security policy which sets limits on both the minimum and maximum lengths of the user passwords such that after padding the password, the length of the padded password is equivalent to the size of one block of the compression function. This countermeasure would be effective for the computer systems and networks that use SHA-1, SHA-256 and SHA-512 hash functions for salt||password hashing.

VI. Conclusion

In this article, we considered the first ever analysis of the salted passwords hashing application. We analysed the security of hash functions based on the Davies-Meyer compression function that process salted passwords. We presented an offline birthday forgery attack on this application which contradicts a general belief that hash values computed over the passwords with prepended salts would complicate birthday attacks. We also discussed some countermeasures against our attack and they serve as general guidelines on how hash functions need to be used for password protection.

Acknowledgments:
The author would like to thank John Kelsey (National Institute of Standards and Technology, USA) for valuable comments on the paper.

References

[1] Ross J. Anderson. *Security Engineering - A Guide to Building Dependable Distributed Systems*. John Wiley and Sons, Inc., second edition, 2008.

[2] John Black, Phillip Rogaway, and Thomas Shrimpton. Black-Box Analysis of the Block-Cipher-Based Hash-Function Constructions from PGV. In Moti Yung, editor, *Advances in Cryptology-CRYPTO*, volume 2442 of *Lecture Notes in Computer Science*, pages 320–335. Springer, 2002.

[3] John Black, Phillip Rogaway, Thomas Shrimpton, and Martijn Stam. An Analysis of the Blockcipher-Based Hash Functions from PGV. *Journal of Cryptology*, 23(4):519–545, 2010.

[4] Jean-Sebastien Coron, Yevgeniy Dodis, Cecile Malinaud, and Prashant Puniya. Merkle-Damgård Revisited: How to Construct a Hash Function. In Victor Shoup, editor, *Advances in Cryptology-CRYPTO*, volume 3621 of *Lecture Notes in Computer Science*, pages 430–448. Springer, 2005.

[5] Ivan Damgård. A Design Principle for Hash Functions. In Gilles Brassard, editor, *Advances in Cryptology-CRYPTO 1989*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427. Springer, 1989.

[6] Richard Drews Dean. *Formal Aspects of Mobile Code Security*. PhD thesis, Princeton University, 1999.

[7] Praveen Gauravaram and John Kelsey. Linear-XOR and Additive Checksums Don't Protect Damgård-Merkle Hashes from Generic Attacks. In Tal Malkin, editor, *Topics in Cryptology - CT-RSA*, volume 4964 of *Lecture Notes in Computer Science*, pages 36–51. Springer, 2008.

[8] Praveen Gauravaram, John Kelsey, Lars R. Knudsen, and Søren S. Thomsen. On Hash Functions using Checksums. *Int. J. Inf. Sec*, 9(2):137–151, 2010.

[9] Praveen Gauravaram and Lars R. Knudsen. On Randomizing Hash Functions to Strengthen the Security of Digital Signatures. In Antoine Joux, editor, *Advances in Cryptology - EUROCRYPT 2009*, volume 5479 of *Lecture Notes in Computer Science*, pages 88–105. Springer, 2009.

[10] Praveen Gauravaram and Lars R. Knudsen. Security Analysis of Randomize-Hash-then-Sign Digital Signatures. *Journal of Cryptology*, 25(4):748–779, 2012.

[11] Jin Hong and Sunghwan Moon. A Comparison of Cryptanalytic Tradeoff Algorithms. *Journal of Cryptology*, 2012. Online version of this article is accessible at http://rd.springer.com/article/10.1007/s00145-012-9128-3.

[12] Antoine Joux. *Algorithmic Cryptanalysis*, chapter 5, pages 155–184. Chapman & Hall CRC Cryptography and Network Security Series. CRC Press, 2009.

[13] John Kelsey and Tadayoshi Kohno. Herding Hash Functions and the Nostradamus Attack. In Serge Vaudenay, editor, *Advanes in Cryptology-EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 183–200. Springer, 2006.

[14] John Kelsey and Bruce Schneier. Second Preimages on n-bit Hash Functions for Much Less than 2^n Work. In Ronald Cramer, editor, *Advances in Cryptology-EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 474–490. Springer, 2005.

[15] Hidenori Kuwakado and Masakatu Morii. Indifferentiability of Single-Block-Length and Rate-1 Compression Functions. *IEICE Fundamentals*, 90-A(10):2301–2308, 2007.

[16] Xuejia Lai and James L. Massey. Hash Functions Based on Block Ciphers. In R. A. Rueppel, editor, *Advances in Cryptology-EUROCRYPT*, volume 658 of *Lecture Notes in Computer Science*, pages 55–70. Springer, 1992.

[17] Stefan Lucks. A Failure-Friendly Design Principle for Hash Functions. In Bimal Roy, editor, *Advances in Cryptology-ASIACRYPT*, volume 3788 of *Lecture Notes in Computer Science*, pages 474–494. Springer, 2005.

[18] Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*, chapter 9, pages 321–383. The CRC Press Series on Discrete Mathematics and its Applications. CRC Press, 1997.

[19] Ralph Merkle. One way Hash Functions and DES. In Gilles Brassard, editor, *Advances in Cryptology-CRYPTO 1989*, volume 435 of *Lecture Notes in Computer Science*, pages 428–446. Springer, 1989.

[20] Steven Meyer. Breaking GSM with Rainbow Tables. *CoRR*, abs/1107.1086, 2011.

[21] National Institute for Standards and Technology. *Federal Information Processing Standard (FIPS PUB 180-3) Secure Hash Standard*, 2008. Available at http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf (Accessed on 01/08/2012).

[22] NIST. *FIPS PUB 180-2-Secure Hash Standard*, August 2002. Available at http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf (Accessed on 31/07/2012).

[23] NIST. Round 3. Official notification from NIST, 2010. The link is available at http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/index.html (Accessed on 31/07/2012).

[24] Philippe Oechslin. Making a Faster Cryptanalytic Time-Memory Trade-Off. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 617–630. Springer, 2003.

[25] Poul-Henning Kamp. LinkedIn Password Leak: Salt Their Hide. ACM website, 2012. Available at http://queue.acm.org/detail.cfm?id=2254400 (Accessed on 31/07/2012).

[26] Bart Preneel, René Govaerts, and Joos Vandewalle. Hash Functions Based on Block Ciphers: A Synthetic Approach. In Douglas R. Stinson, editor, *Advances in Cryptology-CRYPTO*, volume 773 of *Lecture Notes in Computer Science*, pages 368–378. Springer, 1993.

[27] Ronald Rivest. The MD5 Message-Digest Algorithm. Internet Request for Comment RFC 1321, Internet Engineering Task Force, 1992.

[28] Karen Scarfone and Murugiah Souppaya. Guide to Enterprise Password Management (Draft). NIST Special Publication 800-118 (Draft), 2009. Available at http://csrc.nist.gov/publications/PubsDrafts.html (Accessed on 31/07/2012).

[29] James Michael Stewart. *Comp TIA Security + Review Guide*, chapter 6. John Wiley & Sons, 2011.

[30] Wikipedia. Dictionary Attack, 2012. Available at http://en.wikipedia.org/wiki/Dictionary_attack (Accessed on 31/07/2012).

[31] Wikipedia. Password Cracking, 2012. Available at http://en.wikipedia.org/wiki/Password_cracking (Accessed on 31/07/2012).