ICIT 2015 The 7th International Conference on Information Technology
doi:10.15849/icit.2015.0028

SwiftEnc: Hybrid Cryptosystem with Hash-Based Dynamic Key Encryption

Yasir S. Alagl, El-Sayed M. El-Alfy
College of Computer Sciences and Engineering
King Fahd University of Petroleum and Minerals
Dhahran 31261, Saudi Arabia
{g200720290, alfy}@kfupm.edu.sa

***Abstract*—With the emerging need to store massive data in cyberspace and cross platforms, whether in local file systems or cloud-based services, certain security requirements must be met to efficiently protect confidentiality and privacy and manage the large number of keys and access policies. Most of the current encryption standards emphasize one of two trade-off factors: speed of encryption versus ease of key management. Though asymmetric-key encryption does not require the sender and receiver to share a common secret similar to symmetric-key encryption, the cost of the mathematical computations may be unaffordable. In this paper, we first review the state-of-the-art of hybrid cryptosystems. Then, we propose a novel scheme for lightweight encryption of bulk data based on recursive cryptographic hashes and dynamic keys. The effectiveness of the proposed scheme is demonstrated on three files having different sizes, types and contents.**

Keywords—data security; bulk data encryption; cryptographic hasing; hybrid crytosystems; dynamic keys; password-based key derivation; security vault.

I. INTRODUCTION

As the Internet grows in size and number of users, new technologies and applications inevitably emerge to comply with such growth and to satisfy various demands of users. Along with this growing and collaborative environment, the dependability on multiple technological platforms that serve as tools in accommodating many aspects of day-to-day tasks also increases. However, with the tremendous benefits these services provide comes the struggle of protecting users' sensitive data and files within underlying cross-platform systems. The ability to backup, share and synchronize files and folders is becoming crucial over time. The trend to store large volumes of various types of data and files securely is tempting due to durability, portability, flexibility and ease of share, and resistance to threats.

At the heart of security defense mechanisms, encryption arises to protect the confidentiality of valuable data from unauthorized access by programs and individuals [1]. However, at relatively high computational costs, encryption is usually delegated to other parties or skipped in total, thus, exposing the value of an asset to threats [2]. For example, Dropbox, which is a widely-used cloud-based service for hosting files, has been criticized for its weak protection of user's privacy since its first release in September 2008. Lately, similar to a competitive service known as SpiderOak, Dropbox allowed its customers to encrypt their files on the server using the Advanced Encryption Standard (AES) with 256-bit key. In contrast, SpiderOak stores an encrypted version of the decryption key as well in a manner that even the company's employers will not be able to decrypt these files without knowing the customer's password [3]. However, if an intruder managed to get that key, all files can be decrypted.

Generally, cryptosystems fall under two broad categories: symmetric and asymmetric [5]. Although symmetric-key encryption is proven to be relatively faster than asymmetric-key encryption [4], it suffers from two issues. First, it requires sharing a key between the encryption and decryption entities, which might be in different systems. Second, it requires a large number of unique shared keys. Consider a group of N members who engage in an exchange process of a valuable asset T times. Furthermore, consider that each exchange requires an asset to be encrypted with a uniquely generated symmetric key prior to exchanging it. Each member should encrypt a given asset $(N - 1) \times T$ times, in addition to sharing $(N - 1) \times T$ symmetric keys through other secure channels. Moreover, consider having a pool of assets all of which require exchange. The reader can notice the exponential growth in the number of keys and the overhead of sharing them. Examples of the popular symmetric-key encryption standards are Blowfish, International Data Encryption Algorithm (IDEA), Data Encryption Standard (DES), and Advanced Encryption Standard (AES).

Asymmetric encryption, on the other hand, doesn't require the disposal of keys upon each exchange, due to the concealment of the private key. Consider the previous scenario, however, with asymmetric encryption as a requirement for assets exchange. Each member in the group announces his own public key that should be used prior to commencing an exchange with him. This key can still be used with every subsequent exchange resulting in eliminating the overhead of key exchange and the generation of keys. The security of asymmetric encryption depends on the intractability of the discrete logarithm problem and hence comes with higher costs for the encryption and decryption process, i.e. it is relatively slower to encrypt bulk files. Examples of popular asymmetric-key encryption are RSA and ElGamal cryptosystems [5].

doi:10.15849/icit.2015.0028

Here comes the need of an algorithm that combines the merits of the two categories into what is known as hybrid cryptosystems [6]. The goal is to use the public/private key pairs, but maintain superior performance than that of asymmetric encryption. PGP, GnuPG and OpenPGP are examples of the popular hybrid cryptosystems [7]. Another example is a proprietary standard used by Microsoft for Encrypting File System (EFS) since the release of Windows NT Version 3.0.

In this paper, we introduce SwiftEnc, a lightweight hybrid scheme that can be used effectively to encrypt bulk data. SwiftEnc is a hash-based obfuscation algorithm that uses variable-length dynamic key computed based on the file to be encrypted. It also uses an existing asymmetric encryption algorithm to encrypt. The goal is to produce a cipher text in relatively faster time than those of asymmetric algorithms. Based on this scheme, a prototype of a security vault is developed for managing keys in a central store such as Windows Registry.

The rest of the paper is organized as follows. Section II reviews related work. Subsequently, Section III presents the proposed scheme, SwiftEnc algorithm, and describes each of its components in details. We then provide benchmarks comparing the proposed algorithm to existing encryption algorithms in Section IV. Finally, the paper conclusion is given in Section V.

II. Related Work

There have been many trails in both the academic and industry sectors to produce fast encryption algorithms. However, each within its own domain, there hasn't been much work on a general-purpose algorithm that encrypts any file with adequate performance. In this section, we shed the light on some of the recent work that has been made on fast encryption. Presumably, AES is considered the fastest accepted standard of an encryption algorithm worldwide [8]. However, it may not be suitable for very constrained environments and still more improvements are needed [9].

In [10], Wang et al. discussed the use of chaos-based fast image encryption algorithm for image encryption. They proposed combining the scanning process of an image on both stages of permutation and diffusion into one, thus reducing the time required for scanning dramatically. They partitioned the image into blocks of pixels and shuffled the blocks using spatiotemporal chaos and diffused them to change the pixel value at the same time. They also presented an efficient method for pseudo-random generation that is used within their algorithm [11].

In [12], Verkhovsky explained the nature of encryption using Gaussians that belong to complex numbers family. He proposed a new algorithm that finds all cubic roots of Gaussian integers. The algorithm introduces some constraints with regards to encryption time. However, decryption is substantially slower than encryption and hence it only fits applications where only the sender has limited time.

In [13], Hohenberger and Waters introduced an Attribute-Based Encryption (ABE) algorithm with fast decryption. ABE is an expansion of public-key encryption that allows users to encrypt and decrypt messages based on their attributes. However, the complexity of decryption increases as more attributes are utilized. The proposed ABE scheme allows a cipher text to be decrypted with constant number of pairing, specifically 2 pairings, by increasing the private key size.

In [14], LAE is described as a high-speed software block cipher that competes with AES on all standard platforms such as Intel, AMD and ColdFire. LAE works with 128-bit block size and similar key sizes to those of AES, i.e. 128, 192, and 256. It's shown that LAE is faster than AES due to the use of ARX operations (modular Addition, bitwise Rotation, and bitwise XOR) which are supported on most 32-bit and 64- bit platforms. Moreover, LAE rounds are all the same without requiring special end round. The authors also showed that LAE is secure against existing attacks.

Among the attempts to develop encryption algorithms with low implementation complexity comes a promising class of lightweight techniques [15], [16], [17], [18]. For instance, PRESENT is a lightweight block cipher that has been shown to be 2.5 times faster than AES [9].

The concept of dynamic keys or sequence of one-time symmetric cryptographic keys is described and analyzed in [19]. Based on this analysis, the advantages of dynamic keys are revealed in terms of security and efficiency. In essence, if the hacker is able to expose one message, the other messages remain secure. Lastly, in [20] and [21] some trials were made to accelerate the encryption process by the use of Graphical Processing Unit (GPU). However, these trials targeted High Performance Database Management System (DBMS). In [22], the use of GPUs was also noted to accelerate homomorphic encryption.

Some systems and platforms have developed to provide solutions for big data and to establish secure vault for the increased number of keys, certificates and policies. Examples of these systems are the IBM InfoSphere [24], Oracle TDE [25], Microsoft TDE [26], and Volumetric Data Security products [27].

III. Proposed Cryptosystem

In this section, we provide details on the proposed scheme, SwiftEnc. The implementation of SwiftEnc is aimed to be flexible and lightweight. Any available encryption algorithms can be included as long as they meet the requirements of SwiftEnc. The proposed scheme starts by acquiring a secret phrase (passphrase) from the user. This passphrase is used to generate a pair of public and private keys for the chosen asymmetric encryption algorithm. The private key can be discarded at this point while the public key should be stored. The user then selects a file to be encrypted and generates key material or key seed, h_0, from the file itself and some secure pseudo-random numbers. Once the key material is generated, it is passed through a sequence of hashing. To increase the security by maximizing the entropy, the input to each hashing

doi:10.15849/icit.2015.0028

step is output from the previous step XORed with a counter. The process stops once we acquire a bulk key, K_s, that has equal length to the file we intend to encrypt.

To encrypt the file, a simple operation similar to stream cipher is then used. In our case, we XOR the key, K_s, with the file to generate an obfuscated secure output file that can be shared over insecure medium or stored locally. Meanwhile, the public key that was generated from the user supplied passphrase is used to encrypt the initial seed, h_0, and store it with the obfuscated file. Figure 1 shows an outline for the process of encryption in SwiftEnc. The subsequent subsections provide more details on our implementation of the proposed SwiftEnc cryptosystem.

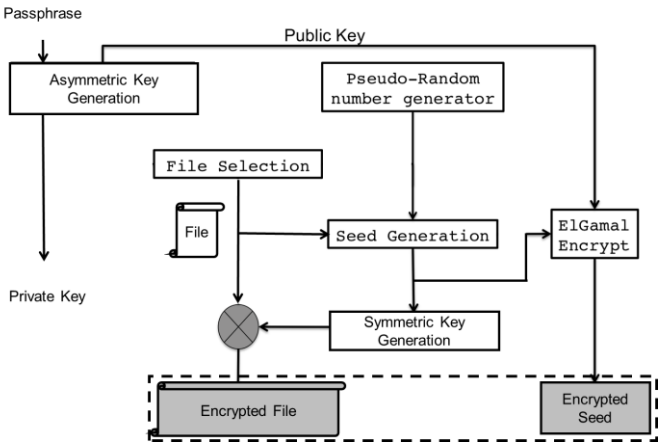


Fig. 1. Outline of the SwiftEnc process.

A. Key Generation and Management

Asymmetric encryption is used to protect the file encryption key seed, h_0. Our choice for asymmetric key generation and sharing in SwiftEnc is ElGamal public-key cryptosystem [23]. However, should the implementer of SwiftEnc make use of other asymmetric encryption algorithm, the general scheme still holds. For instance, using RSA requires the use of two keys: public key and private key. While the private key should be kept safe and private by its owner from unauthorized access (as the name suggests), the public key does not. Assuming a single platform on which the user intends to encrypt his assets or files for his own use, the public key can be kept in his home directory or in any sort of non-protected data store, e.g. Windows Registry. In a scenario where a group of members communicate securely back and forth, each member's public key should be announced within the group together with a certificate to authenticate the validity of the public key.

To avoid the need to a trusted third party to issue a certificate, we use ElGamal algorithm for key generation and sharing. This algorithm is based on Deffie-Hellman key exchange and uses two keys at each of the sender and the receiver sides. These keys are generated in such a way to allow them to share a session key. For example, assume *A* is the sender and *B* is the receiver. Then, *A* should have $K_{prv,A}$ and $K_{pub,A}$, and *B* has $K_{prv,B}$ and $K_{pub,B}$. The receiver, *B*, starts by defining a cyclic group *G* of order *p*, where *p* is a large prime number. This cyclic group has a generator *g*. *B* then selects a private key $K_{prv,B} < p - 1$ randomly from *G* and calculates a corresponding public key $K_{pub,B}$ as follows:

$$K_{pub,B} = g^{K_{prv,B}} \bmod p \quad (1)$$

B announces the tuple ($K_{pub,B}$, *g*, *p*) or stores it in a shared folder. If *A* wants to securely send a file to *B*, it should obtain the tuple ($K_{pub,B}$, *g*, *p*) and selects a private key $K_{prv,A}$ from the group *G* generated by (*g*, *p*). Then, *A* calculates an ephemeral public key $K_{pub,A}$ as follows:

$$K_{pub,A} = g^{K_{prv,A}} \bmod p \quad (2)$$

It also calculates a shared key, K_m, to be used for encrypting the file encryption key seed, h_0. The calculation of K_m is as follows:

$$K_m = (K_{pub,B})^{K_{prv,B}} \bmod p \quad (3)$$

Km will be used to encrypt the message using ElGamal encryption algorithm and the encrypted message together with $K_{pub,A}$ will submitted to the receiver. The decryption will be performed use inverse operation.

As an alternative approach in SwiftEnc, we used SHA-512 to hash the passphrase and the result *x* is identified as our private key. The passphrase could be of any length, complexity, and combination of characters' groups, e.g. uppercase, lowercase, special characters, numbers, etc. The use of a passphrase introduces usability rather than remembering a random number. The passphrase can be fixed for all files or can be changed for each file. In our case, we made it fixed for all files in the vault. The passphrase goes through a one-way hashing function such as MD5 or SHA-512 to produce a fixed-length hash string then use the first 128 bits or 512 bits, for example, as $K_{prv,B} = x$. Password-based key derivation is common in practice and industry standards such as PKCS and OpenPGP. In [28], a framework for the design and analysis of password-based key derivation functions (KDFs) is provided.

doi:10.15849/icit.2015.0028

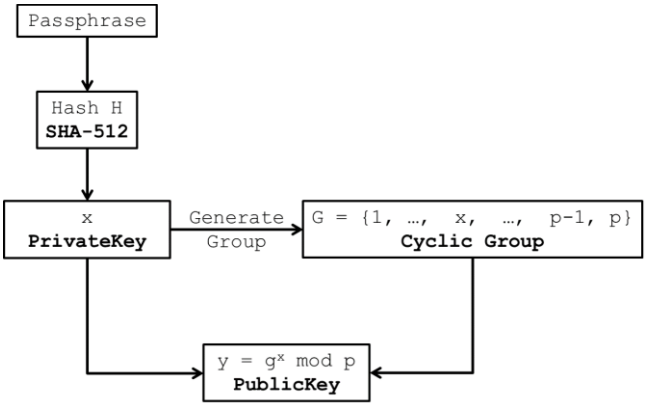


Fig. 2. SwiftEnc public/private key generation using ElGamal.

Afterthat, G is chosen such that: $x \in G$, x can be generated by g to some order $1 < x < p - 1$ where p is a prime number. In our implementation, we used the BigInteger.probablePrime() method in Java to generate a value for p. Once these conditions are met, we can identify our public key as per ElGamal and discard the private key, x, completely. Hence, avoid the overhead of storing an encrypted version of the private key; it can be generated whenever needed from hashing the passphrase. We can also pass x to the hash function, for the second time, and produce x_1 which can be used to check the validity of the entered passphrase in later operation, i.e. decryption. Figure 2 illustrates this process for public/private key generation.

One important concept to note here is that SwiftEnc doesn't use the public/private key pair for encrypting/decrypting assets. Indeed, the passphrase and generation of keys don't account for the confidentiality of the asset by any factor. However, the encryption/decryption depends on the asset itself as will be discussed in the subsequent sections.

B. Seed Generation

In SwiftEnc context, the seed refers to the string of characters that will be used to generate a symmetric key that, in turn, will be used to obfuscate the asset which the user intends to encrypt. However, this seed varies in length and value per each file. The seed is the secret that we want to insure that it's properly protected, as obtaining the seed reveals the confidentiality of the asset as we will see in Section III-D.

Since every file will have its own unique seed, the seed has to be stored along with the protected asset, however, in a confidential format. We will see in subsequent sections that the seed is necessary to decrypt the asset and return the file to its original state. The implementation of SwiftEnc can use any seed generation algorithm to associate a seed to a file under the following conditions: (a) The algorithm guarantees a sufficient degree of pseudo-randomness, and (b) The algorithm acquires very low probability of collision. In SwiftEnc, we create the seed from the first block of the file to be encrypted as indicated in Algorithm 1.

Data: File to encrypt (*FE*), Initial seed size (*SS*)
Result: Initial seed (h_0)
FS = *FE.getSize*();
count = 0;
while *count* < min(*SS*, *FS*) **do**
    h_0[*count*] = *FE.getNextByte*();
    h_0[*count*+1] = *SecureRandom*();
    count += 2;
end
while *count* < *SS* **do**
    h_0[*count*] = *SecureRandom*();
    count++;
end
return h_0;

Algorithm 1. Seed creation algorithm.

C. File Obfuscation

Once the seed is generated for the perspective asset that we intend to protect, the Symmetric Key Generation process and file obfuscation can start immediately. Obfuscation is the core of SwiftEnc on which the asset's data are being randomly scrambled to generate an encrypted file. Moreover, this operation occurs with minimal processing power and fast timing, hence the term Swift. To assure that SwiftEnc accommodates larger file sizes, we use buffered streams to process the file sequentially.

We generate a key from the seed by recursively hashing it. Since SwiftEnc is using SHA-512, the first 512 bits (64 bytes) of the key will be the hash of the initial seed h_0. The following 64 bytes will contain the hash of the resulting hash from the previous step, and so on. We repeat this operation until we reach a key equal in length to the first 64 bytes multiple that is larger than the file size. Next, we perform a regular XOR operation between each byte of the asset and the key and send/store the result as our encrypted file. The use of XOR with the hash gives SwiftEnc the low processing power and better performance over other encryption algorithms, however, we haven't discussed what gives it a confidentiality level. Algorithms 2 and 3 demonstrate these processes. The illustration of the prototype operation is depicted in Fig. 3. The XOR operation is reversible in nature. So, we can use this property to decrypt the file and retrieve the original cleartext file. By only having the encrypted file and the seed, we can generate the same key by hashing the seed recursively and XORing it with the file, thus, revealing our file back.

doi:10.15849/icit.2015.0028

Data: File to encrypt (*FE*), Initial Seed (h_0)
Result: Symmetric encryption key (K_s)
FS = FE.getSize();
SS = h_0*.getSize*();
count = ceil(*FS* / *SS*);
key_0 = *hash*(h_0); // key_0 subscript means block
K_s = key_0[0..*SS*]; take the first *SS* bytes
i = 1;
while *i* < *count* **do**
    key_i = *hash*($key_{i-1} \oplus i$);
    $K_s = K_s \,||\, key_i[0..SS]$; // concatenation
    i ++;
end
return K_s;

Algorithm 2. Symmetric key generation for file encryption.

Data: File to encrypt (*FE*), Key (K_s)
Result: Encrypted file (*EF*)
FS = FE.getSize();
i = 0;
while *i* < *FS* **do**
    $EF[i] = FE[i] \oplus K_s[i]$;
    i ++;
end

Algorithm 3. File obfuscation

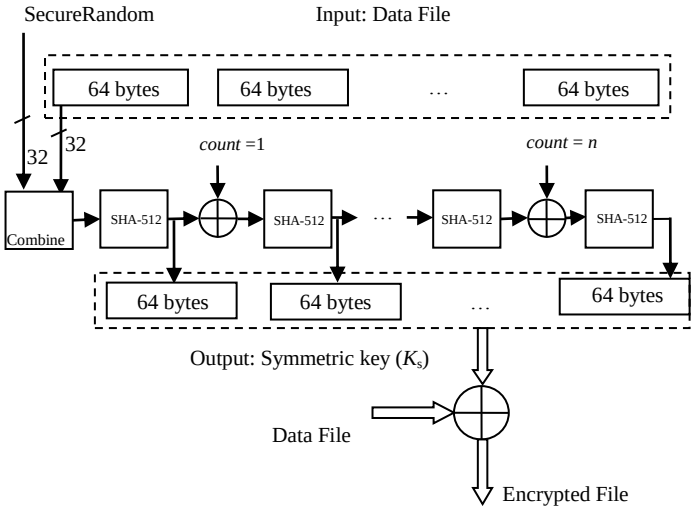


Fig. 3. Main steps for generating symmetric key and encrypting the data file.

D. Seed Cryptography

As we have seen, the seed is generated by extracting its value from the file and a pseudo-random number generator. The seed is also used to create a key that is equal in length to the length of the file. Once we obtain the key, we can encrypt our asset immediately and discard the key completely. However, obtaining the seed compromises the security of the system and redeems the asset unsecured. Once an unauthorized entity obtains the seed, our asset is no longer protected.

To thwart against such threat, the owner of the asset should provide a layer of protection over the seed. SwiftEnc ensures that this layer is implemented by encrypting the seed using any well-known Asymmetric Encryption algorithm; in our case we have chosen ElGamal as discussed above using K_m from Eq. (3):

$$h_0' = h_0.K_m \bmod p \tag{4}$$

The encrypted seed, h'_0, should be stored next to the file, appending/pre-appending it to the file, or in a data-house that could link it to the file. Upon decryption, we should retrieve the seed with respect to the file, decrypt it using ElGamal, then initiate the de-obfuscation as stated in Section III-C. To decrypt the seed, ElGamal has to calculate K_m at the receiver then use its inverse in the cyclic group *G* to decrypt the seed:

$$K_m = (K_{pub,A})^{K_{prv,B}} \bmod p \tag{5}$$

$$h_0 = h_0'.K_m^{-1} \bmod p \tag{6}$$

IV. EVALUATIONS

We have developed the algorithm described above and built a prototype for a security vault as a central location for managing encrypted files and passwords. Figure 4 shows part of the user interface for the main menu and the security vault. We report some empirical experiments to benchmark SwiftEnc with another password-based hybrid encryption algorithm (Rijndael-RSA) [29]. Rijndael-RSA encrypts and decrypts using 256-bit Rijndael key where the key is encrypted using 1024-bit RSA key, which is password-encrypted. All implementations were conducted in Java and experiments were run on the same machine using the specifications shown in Table I.

TABLE I. EXPERIMENTS SPECIFICATION

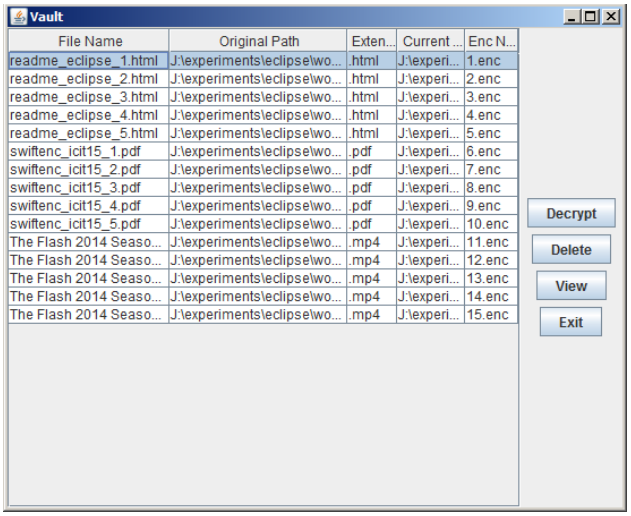
OS	64 bit Windows 7 Professional
Processor	Intel Core i5-33M CPU 2.7GHz
Memory	4 GB
Implementation	Java 1.7

The algorithms are tested on three files of different sizes and content types. The first file is the readme file that comes with eclipse and contains HTML. The second file is the PDF of an initial version of this paper. The third file is MP4 file

doi:10.15849/icit.2015.0028

corresponding to "The Flash 2014 Season 1 Episode 01". The performance measures are reported in terms of: average time and speed. The time includes I/O reading and writing times, the key generation, encryption and decryption. The speed is calculated as size in MB divided by time in seconds. Table II illustrates the average times in seconds for five runs as well as the speed.

(a) Main menu interface



(b) Vault interface

Fig. 4. Screenshot a security vault protype based on SwiftEnc for encryption and decryption.

TABLE II. COMPARISION OF AVERAGE OVERALL TIME (SEC) AND SPEED (MB/SEC) APPROX TO 4 DECIMAL DIGITS

Input File		SwiftEnc		Rijndael-RSA	
Size (MB)	**Type**	**Time**	**Speed**	**Time**	**Speed**
0.1	HTM	0.2444	0.4092	2.9204	0.0342
0.773	PDF	0.6598	1.1716	14.6696	0.0527
272	MP4	187.7854	1.4485	4925.0770	0.0552

V. CONCLUSION

This paper discussed the trade-offs of encryption algorithms and how they can impose a barrier on the value of assets due to their relatively high processing time. We introduced a new hybrid algorithm, SwiftEnc, and security vault prototype, that can be used to overcome this barrier and allow for rapid encryption with low processing power. The vault prototype provides a central local store for securely managing keys and encrypted files. The framework can be customized with different cryptographic functions to accommodate various security standards enforced by an organization. SwiftEnc showed better performance when compared to another algorithm. When used for communication over the Internet, message exchanges between the sender and the receiver can also include timestamp and nonce to counter replay attacks.

ACKNOWLEDGMENT

The authors would like to acknowledge the support provided by King Fahd University of Petroleum & Minerals (KFUPM) during this work.

REFERENCES

[1] D. R. Stinson, Cryptography: Theory and Practice. CRC Press, 2005.

[2] A. Nadeem and M. Y. Javed, "A performance comparison of data encryption algorithms," in *Proc. IEEE International Conference on Information and Communication Technologies*, 2005, pp. 84–89.

[3] S. Latha, K. Raju, and S. Santhi, "Overview of dropbox encryption in cloud computing," Transactions on Engineering and Sciences, vol. 2, no. 3, pp. 27–32, 2014.

[4] A. Al-Hasib and A. Haque, "A comparative study of the performance and security issues of AES and RSA cryptography," in *Proc. Third International Conference on Convergence and Hybrid Information Technology, ICCIT '08*, vol. 2, Nov 2008, pp. 505–510.

[5] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, Handbook of Applied Cryptography. CRC Press, 2010.

[6] A. W. Dent, "Hybrid cryptography," Information Security Group, University of London, Tech. Rep., 2005.

[7] The International PGP Home Page. [Online]. Available: http://www.pgpi.org/

[8] F. P. Miller, A. F. Vandome, and J. McBrewster, Advanced Encryption Standard. Alpha Press, 2009.

[9] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. Robshaw, Y. Seurin, and C. Vikkelsoe, "PRESENT: An ultralightweight block cipher," in *Cryptographic Hardware and Embedded Systems - CHES 2007, Lecture Notes in Computer Science*, vol. 4727. Springer, 2007, pp. 450–466.

[10] Y. Wang, K.-W. Wong, X. Liao, and G. Chen, "A new chaos-based fast image encryption algorithm," Applied Soft Computing, vol. 11, no. 1, pp. 514–522, 2011.

[11] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway, "A concrete security treatment of symmetric encryption," in *Proc. 38th Annual Symposium on Foundations of Computer Science*, 1997, pp. 394–403.

[12] B. Verkhovsky, "Cubic root extractors of gaussian integers and their application in fast encryption for time-constrained secure communication," Int. J. of Communications, Network and System Sciences, vol. 4, pp. 197–204, 2011.

[13] S. Hohenberger and B. Waters, "Attribute-based encryption with fast decryption," in *Public-Key Cryptography–PKC 2013, Lecture Notes in Computer Science*. Springer, 2013, vol. 7778, pp. 162–179.

[14] D. Hong, J.-K. Lee, D.-C. Kim, D. Kwon, K. H. Ryu, and D.-G. Lee, "LEA: A 128-bit block cipher for fast encryption on common processors," in *Information Security Applications*. Springer, 2014, pp. 3–27.

[15] T. Eisenbarth, S. Kumar, C. Paar, A. Poschmann, and L. Uhsadel, "A survey of lightweight-cryptography implementations," IEEE Design & Test of Computers, vol. 24, no. 6, pp. 522–533, 2007.

[16] B. Adida, S. Hohenberger, and R. L. Rivest, "Lightweight encryption for email," in *USENIX Steps to Reducing Unwanted Traffic on the Internet Workshop (SRUTI)*, 2005.

[17] E. Choo, J. Lee, H. Lee, and G. Nam, "SRMT: A lightweight encryption scheme for secure real-time multimedia transmission," in *Proc. International Conference on Multimedia and Ubiquitous Engineering*, 2007, pp. 60–65.

doi:10.15849/icit.2015.0028

[18] D. Engel and A. Uhl, "Lightweight JPEG2000 encryption with anisotropic wavelet packets," in *Proc. IEEE International Conference on Multimedia and Expo, ICME '06*, 2006, pp. 2177–2180.

[19] H. H. Ngo, X. Wu, P. D. Le, C. Wilson, and B. Srinivasan, "Dynamic key cryptography and applications," International Journal of Network Security, vol. 10, no. 3, pp. 161–174, 2010.

[20] H. Jo, S.-T. Hong, J.-W. Chang, and D. H. Choi, "Data encryption on gpu for high-performance database systems," Procedia Computer Science, vol. 19, pp. 147–154, 2013.

[21] ——, "Offloading data encryption to GPU in database systems," The Journal of Supercomputing, pp. 1–20, 2014.

[22] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Accelerating fully homomorphic encryption using GPU," in *Proc. IEEE Conference on High Performance Extreme Computing (HPEC)*, 2012, pp. 1–5.

[23] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," in *Advances in Cryptology*. Springer, 1985, pp. 10–18.

[24] http://www-01.ibm.com/software/data/infosphere/

[25] http://www.oracle.com/technetwork/database/options/advanced-security/index-099011.html

[26] https://msdn.microsoft.com/en-us/library/bb934049.aspx

[27] http://www.vormetric.com/

[28] F. F. Yao, and Y. L. Yin, "Design and analysis of password-based key derivation functions," in *Topics in Cryptology*, Springer, 2005, pp. 245-261.

[29] J. Garms and D. Somerfield, Professional Java Security. Wrox. 2001.