https://doi.org/10.48009/2_iis_2013_444-451

Issues in Information Systems
Volume 14, Issue 2, pp.444-451, 2013

DISTRIBUTED KEY SYSTEMS: ENHANCING SECURITY, FAULT TOLERANCE AND DISASTER RECOVERY IN CLOUD COMPUTING

Erich Rice, Saint Cloud State University, rier1201@stcloudstate.edu
Paul Safonov, Saint Cloud State University, psafonov@stcloudstate.edu
Dennis Guster, Saint Cloud State University, dcguster@stcloudstate.edu

ABSTRACT

Cloud computing has increased the need for sound and sophisticated security strategies. Although encryption is the primary mechanism to provide security in data transmission, cloud computing offers both advantages and disadvantages to enhancing the encryption process. Encryption algorithms currently available are robust, but are well known and require a "key" to make a session unique. Therefore, if the key is compromised the encryption can be unlocked no matter how robust the algorithm. In the classical model, a node in a network contains the entire key, and if compromised, then potentially the entire cloud's security could be breached. Also, if the key were destroyed then all the data it protects could be lost, because the data couldn't be unencrypted. Therefore, splitting the key and storing portions on multiple nodes within a cloud would make it more difficult to compromise or lose the key. Further, the key could be striped (like a RAID, Redundant Array of Independent Disks) so if distributed across multiple nodes some could fail and the key could still be recovered. This methodology could create greater security, while providing greater fault tolerance by allowing retrieval of the entire key even if some nodes were compromised or destroyed.

Keywords: Distributed Encryption, Information Technology (IT) Security, Cloud Computing, Data Redundancy, Disaster Recovery and Fault Tolerance

INTRODUCTION

In this era of constant threats from malicious actors on the Internet, the need for more secure and robust transmission of data has never been more paramount. This threat has increased even more by the wide spread use of cloud computing, due to new threats such as data loss or leakage, non-secure interfaces or API's, account or service hijacking, and malicious insider threats [5]. The recent attacks on the likes of the Federal Reserve, New York Times, and even Apple has shown us that the need for increased security in our ever more connected world will only gain greater importance going forward. While the cost of sending data over the Internet is quite attractive it is certainly a hostile data transmission environment. And although some of the data currently sent over the Internet is in the clear, any sensitive data needs to be adequately encrypted, or it can easily be viewed by prying eyes. Most would agree that the newer forms of encryption algorithms available today are very capable of adequately encapsulating the data, and therefore would be considered quite robust [1]. However, the manner in which encryption is used can become problematic. This problem is often created by the need to use a cryptographic key, which is needed to make the session as unique as possible. A common example would be Secure Shell (SSH), which is a cryptographic network protocol used to attain secure data transmission, which in turn creates a secure channel within an insecure network such as the internet [23]. While the encryption algorithm itself maybe robust, the problem then becomes if the key is compromised, the underlying data transmitted across the channel is put in peril because the key in effect can be used to "unlock" the encapsulated message.

Most would agree that the current algorithms are very robust, but they are also widely known and therefore there is a great deal of information on how they are structured and perform. By undertaking a little research on the Internet a potential hacker can obtain the ability to crack common encryption methodologies [11]. While this is bad news for current algorithms being utilized, it is far worse for older algorithms, a couple of versions behind the current cutting edge. This creates the impetus for one to keep their encryption strategies up to date. Of course, hackers can further complicate this problem by leveraging current advanced technology. A prime example of this is taking advantage of the enormous amount of processing power provided by a cloud-computing environment. This can be seen both on a CPU level, as with Moore's Law which states that transistors or integrated circuits double roughly every two years [14] but also with the growing power and availability of computing clusters or extremely high powered computers

[9]. Therefore, one needs to be prepared to combat numerous attack vectors against the encryption methodology being utilized; including brute force attacks, social engineering, side channel attacks and compromising the key. Of course, compromising the key offers the quickest and easiest solution to compromising the encrypted data, and therefore it is imperative to defend the key at all costs.

A prime methodology that can be used to protect the key is to control how the key is derived, such as through the use of a random number generator [2], and further keeping it as secure and secret as possible once it is created. One method that is commonly used is the concept of key agreement protocols, which is discussed in [23], and uses both the client and server to derive the shared key that they will use. This strategy enhances the security of the key by increasing the difficulty to corrupt it, because both the client and the server would need to be compromised to attain the entire key structure [7]. Several software companies have proposed various options for effective key management at the enterprise level [12], as well as recommendations from government agencies such as NIST [23]. However, splitting the key within a cloud-computing environment has not been widely researched and appears to not have been implemented commercially at this time. From the literature it appears that the most popular commercially available option is using a third party entity as an intermediary between the client and server, which in turn acts as a key translation center and evaluates the method of encryption [4], although as mentioned earlier a non-insecure interface or API is one of the greatest threats to cloud infrastructures [5].

An evaluation of the different methods that have been either proposed [12] or implemented [23, 12] reveals a common theme. Within that theme there are two salient points: the first being that if the key is compromised then the encrypted data can be utilized by a hacker for their own undoubtedly malicious purposes; and second if the key is destroyed by a hacker then the encrypted data is lost and can no longer be used by the rightful owner [16]. A manager of an enterprise architecture could look upon the second scenario as the more frightening of the two, because a hacker breaking in and stealing credit card numbers could lead to adverse publicity and inconvenience to its customers. However, if the records were permanently made unavailable to the company as a result of a destroyed key, then the outcome might even be worse, perhaps even leading to the complete dissolution of the company itself as the importance of data has become ever more critical to commercial enterprises. A situation of this type emphasizes the need for redundancy in the cryptographic key. However, there is a tradeoff in that the more keys create a higher probability that the encrypted data can be compromised. This situation then leads to a tradeoff between redundancy level and key vulnerability, and thus overall security.

So therefore, a key is the crucial part of the encryption strategy and accordingly key length needs to be adequately evaluated to ensure adequate security robustness is attained. One widely implemented solution is to have encryption keys and passwords stored in escrow with a secure third party [13]. The use of this third party concept does redirect the security responsibility by placing the key on an external host. However, that host still stores the entire key and therefore would simply become a new and very tempting target to a hacker. Given the vulnerabilities of storing an entire key on a single host, it is paramount to have an effective key management plan in place. This plan needs to be consistent across the entire enterprise. Therefore, effective key management is the lynchpin to successful use of encryption in an enterprise cloud-computing environment and further research is needed especially related to key storage strategies. This logic spills over to personnel as well and can become problematic in large security groups, especially when a member leaves [21]. Imagine if the member who was leaving the enterprise had devised the key distribution strategy and in so doing had failed to document it well!

The goal then should be to balance the need for greater security robustness with the need for fault tolerance and backup redundancy for the key, against the increased system overhead and attack points created by breaking up the key across multiple distributed nodes [16].

Literature Review

Distributed encryption methods spread across multiple nodes is not a new or novel idea, although it has not seemed to catch on yet for enterprise level applications. At least since the mid-1990s the concept of breaking up an encrypted key across more than one node has been looked at, and even implemented in test scenarios [15]. Here the authors implemented a distributed encryption and decryption system such that there was (n – t) nodal support, where n was the number of nodes in the system and t was the number of nodes that could be faulty due to hardware or software malfunctions. In their system the encryption is started at a certain "source node" and then proceeds along a

path which contains the various broken up pieces of the key, until a *completed path* is performed with all the necessary pieces of the key [15, p. 12-13] Ultimately they were able to implement their design to show that the algorithm which they had created worked with a t = 2 logic, while traveling five completed paths, within a Unix environment on a pair of Sun workstations [15, p. 19]

A more recent example of a work looking at a distributed key methodology across multiple nodes looked at basic key management, and a simplistic strategy of splitting the key up across a number of nodes (N) with a key size of 16 bytes [10]. In the authors example the key was split across four nodes, with a (N – 2) logic whereby up to two of the nodes could be compromised and the system would still function. In this example the keys used were relatively small, only 16 bytes total, with each of the nodes containing 12 out of the 16 bytes, in a sliding window pattern so that any two of the nodes would have the requisite bytes of the entire 16 byte key [10, pp. 5-6]. Guster, et al., also discussed one of the main advantages to the distributed key system, that a hacker even if they took over one of the key nodes would more than likely not know the total length of the key, and would thereby increase the difficulty in successfully implementing a brute force cracking of the key. They also looked at the possibility of generating the key, or portions of the key, through the use of a quantum number generator, which could increase the robustness of the key by making it more random and less vulnerable to pattern recognition and subsequent cracking [6, 18] which would build upon what was recommended by NIST [23].

Another example of a distributed cryptographic key system being implemented can be seen in [17] where a Unix script was devised to split an RSA [19] algorithm across several hosts. A pre-generated key would then be read in a file and the output that was created by the Unix script would then be treated as fractions of the initial file in the working directory. Upon reforming the fragments, there would be a check against an MD5sum hash that had been generated to assure that file integrity had been maintained. While this was a good initial step at implementing a working solution to the problem, it was acknowledged that it lacked the complexity needed for a production environment [17, p. 10]. The concept was taken to another level with the system described in [16] where a distributed encryption system was developed for a multi-nodal environment using NodeJS. Such a system as this could easily be adapted for a cloud-computing environment where multiple nodes could be distributed across the cloud, while being housed in geographically distinct data centers, thus meeting both security robustness measures of breaking up the key, as well as increased redundancy by distributing the key among multiple nodes [16, p. 8].

Ultimately the goal of any distributed encryption system would be to unlock the data that had been encrypted on a database, the database itself could be distributed across multiple nodes, utilizing a similar logic as in [16] such that even if certain data blocks are lost they can be recovered [22]. The striping of the data across nodes would be beneficial as far as load balancing and would aid in the tuning of the database structure as well, but the main advantage would be in the recovery and resiliency of the data itself. Another implementation created a distributed storage system that would provide regenerative capability, again built upon the [n, k] maximum-distance-separable (MDS) framework described in [22]. Though it adds another variable to the equation, such that even if an eavesdropper does gain access to the data stream, the replication of the entire dataset when a node is compromised by restricting the data that is exposed to an eavesdropper in the repair process is limited [20]. Both of these models could further aid in the development of a fully functional distributed encryption system, based on a cloud-computing framework across geographically distinct datacenters.

RESEARCH METHODOLOGY

Randomness and the Key

The randomness of the seed number for the key generation is an important part in determining the robustness of the final encryption. The more random the seed number is, the more unlikely it would be a hacker could re-engineer the algorithm in use, and replicate the key. Though it can be a rather difficult operation, given enough iterations a pattern can begin to present itself and that is when the underlying algorithm can become visible and therefore vulnerable [6]. Thus the more random the seed number is, the better off the overall system integrity will be. While there are many "random" number generators currently available, they are random only to a certain point, at which the underlying algorithm used to generate the numbers begins to repeat itself and it non-randomness is exposed.

For instance in [16] a pseudo-random number generator built into a Javascript library was utilized for one instance to generate the seed number for the encryption key, although it was found to be random in small sample sizes, after thousands of iterations patterns developed in the numbers generated. NIST recommends using an approved Random Bit Generator (RBG), which have been specified in various publications [2, p. 11] that creates a number based on the bit pattern created by the RBG. The RBG's recommended by NIST can be either software algorithms or physical devices, and have a greater randomness than that found in the Javascript library pseudo-random number generator. But again the RBG's have an underlying algorithm, albeit more intricate and advanced, though eventually patterns in the bit sequencing could be exposed as nothing relying on classical physical principles can be absolutely unpredictable [3].

Therefore, as was determined in [16] a better answer to the "random-number generators" or the RBG's could be through the use of a quantum mechanical based number generation system. By eliminating the reliance on classical physics principles, perhaps a truly "random" seed number could be found to greatly enhance the reliability and robustness of currently available encryption methods. Already much research has been devoted to this issue as perhaps as much as $50 billion is lost every year in the United States due to identity theft, and known security breaches have occurred because of random-number generation that has been broken by hackers [3]. Although commercially available quantum-number generators are available and provide a far more random number set than those created by the classical based "random-number generators", they have not been yet scientifically proven to be 100% random. NIST's Information Technology Lab (ITL) is currently working on developing a truly random sequence of numbers that is in no way correlated to anything prior to its generation, which it would then provide free over the Internet, allowing for a trusted common standard [3]. The hope is that such a system can be implemented within the next few years, however, even with the technology currently available quantum-number generators offer a far superior level of randomness and should be considered when developing a best practices framework for encryption keys.

Determining the Key Size

The first choice that would need to be made in building out a distributed encryption system would be to determine the size of the key. The larger the key the more robust the security it could provide, however, at a certain point the increased size of the key would run into the law of diminishing returns and the increase in security would become negligible versus the increased computing overhead to encrypt and decrypt the keys. So a balancing act needs to be performed to find a key size that provides the requisite amount of security robustness versus the computing overhead needed to perform the encryption functions. At one time a 128-bit key might have been sufficient to provide a decent amount of security robustness, however, as was pointed out earlier the increase in computing power has made smaller key sizes a non-viable option [9]. Even with a truly random seed number provided by a quantum-number generator the relatively small number of computations needed to break the smaller key sizes have made their use obsolete. Moving forward, NIST recommends using at least a 224-bit key with 2048-bit encryption using the 3TDEA symmetric algorithm, although the level of security required could alter this requirement [8]. In the test system developed in [16] the key size chosen was 24 bytes or 192-bit, to simplify the splitting up of the key across the chosen number of nodes in the system. Table 1 below shows the 24-byte key broken up over the 6 nodes:

Table 1: Splitting the Key Across the 6 Nodes (X=missing bytes)

Node	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
1	x	x	x	x	x											x						x		x
2	x					x	x	x	x							x	x				x			
3		x				x				x	x	x					x	x					x	

4			x				x			x			x	x				x	x			x		
5				x				X			x		x		x				x	x	x			
6					x				x			x		x	x					x			x	x

Table 2 below shows the key represented by the two Hex characters under each of the 24 bytes across the top of the Table, the Initialization Vector (IV) and SALT are then listed under the key, and finally the six rows beneath them represent the six nodes of the distributed key system and show the key broken up according to the method described in Table 1.

Table 2: Key (Top Boxes) with Initialization Vector (IV) and Salt

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
D8	77	9B	57	4A	BB	43	34	5F	77	96	C3	42	19	42	79	47	2A	D6	E5	A1	B3	50	4F

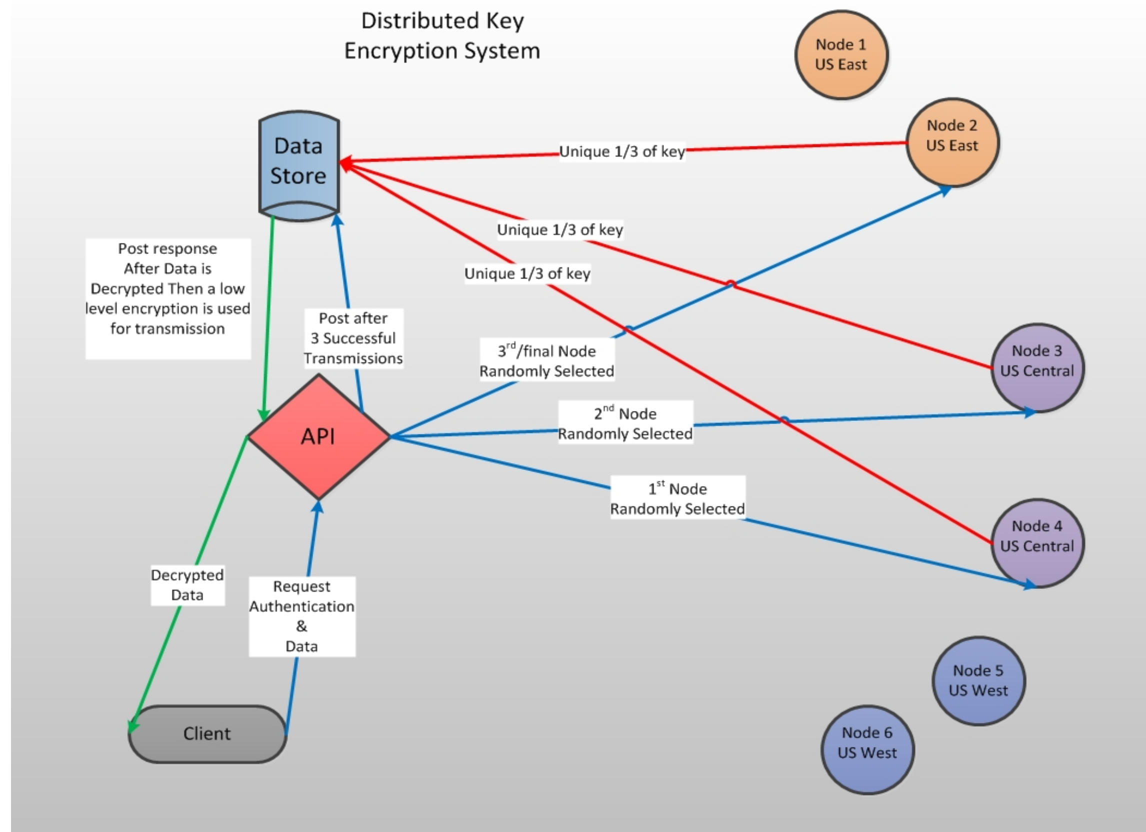
IV= 99BA8C95FBB459DBE5E92C769FFC55BB
SALT=2B336181CE15142C

					BB	43	34	5F	77	96	C3	42	19	42		47	2A	D6	E5	A1		50	
	77	9B	57	4A					77	96	C3	42	19	42			2A	D6	E5		B3	50	4F
D8		9B	57	4A		43	34	5F				42	19	42	79			D6	E5	A1	B3		4F
D8	77		57	4A	BB		34	5F		96	C3			42	79	47			E5	A1		50	4F
D8	77	9B		4A	BB	43		5F	77		C3		19		79	47	2A				B3	50	4F
D8	77	9B	57		BB	43	34		77	96		42			79	47	2A	D6		A1	B3		

Distributed Encryption Architecture

Once an appropriate key size has been decided upon the next step would be to determine how the distributed encryption method would be broken up across the cloud environment. In their test system, Redman, et al., distributed the key across six nodes, where three nodes were required to replicate the full key and also to unlock the data [16]. This (6-3) nodal logic was utilized as a basic configuration to show the advantages to having a distributed encryption system, namely that up to two of the nodes could be compromised by a hacker or brought down by some other means, and the data would remain secure as three of the nodes are required to unlock the encrypted data. This would not preclude the use of more nodes, which could very well provide an even greater level of security and fault tolerance, a system say with a (12-6) nodal logic could provide greater security and even more redundancy and fault tolerance. One drawback to the smaller number of nodes in a system is that as the number of nodes compromised gets closer to the maximum level, for instance 2 in the (6-3) nodal logic, the computations required to determine the remaining portions of the key would be greatly reduced and could thus make the system more vulnerable. However, as mentioned earlier with the key size, the law of diminishing returns eventually takes over and the increased overhead in system administration and communication latency between them makes the use of ever-greater numbers of nodes superfluous. Another potential consideration for communication latency comes into play when looking at whether the nodes are split up over a LAN or WAN connection, and what the maximum data transmission units are for the lines they are set up on are. If the distributed encryption system were a part of an e-commerce operation, more than three-second latency could negatively effect end user experience, and be irreparably detrimental to the

bottom line [16]. Thus, more than likely the distributed encryption environment nodes would be set up on a back channel high-transmission line that could quickly and effectively produce the desired connections and distribution of the various key parts to fulfill the un-encryption process. The diagram below shows a distributed key encryption system with a six-node structure that has been geographically split across the cloud.



The other important consideration in looking at the number of nodes used in the distributed encryption method is how the key is broken up over the nodes. In [16] a sliding window approach was used whereby the key was split across the nodes to allow any three to replicate the key. To take full advantage of the added security a distributed encryption system can provide it was imperative to make sure that any two nodes did not recreate the key. And finally, to prove the fault tolerance and added recoverability of the distributed key system it was necessary to split the key in such a way so as up to three nodes could be lost, including their portions of the key, and the remaining three nodes could recreate it. Without this overlapping of the key structure one of the main benefits of the distributed encryption system would be lost, namely the recoverability of the key from the distributed nodes, thus using this type of key splitting logic could prove problematic requiring a great deal of effort to figure it out prior to deployment.

Utilizing the sliding window method requires accurately calculating the amount bits of the key to place on each node and the specific order in which to break the key up [16, p. 7] without this time consuming and labor intensive calculations the replicability of the key could not be assured. For this reason other methods of breaking up the key across multiple nodes could be utilized, such as placing random lengths of the key on different nodes, say one node might contain 32 bits of the key while another might contain 64 bits. This again would require precise calculations to assure that the entire key could be replicated through the minimum number of nodes specified within the distributed key logic. Another option might be creating a distributed key structure with dynamic random lengths, obviously, the more random the key lengths on each node could make it more difficult to crack the encryption being

utilized. If a hacker had no idea how much of the key was on any given node, or if there was even a portion of the actual key, then their ability to brute force crack the remaining portion of the key would become very difficult to perform. Ultimately, any key splitting logic that was used would need to take into account the requirements of unlocking the data with the minimum number of nodes specified, while at the same time allowing recovery of the entire key with the minimum number of nodes specified in the distributed key logic.

CONCLUSIONS

Ultimately the utility of a distributed encryption system will be determined by the balancing of security robustness versus performance concerns due to system delay. The factors to take into account when determining the security robustness would be the importance of the key size, sufficiency of the encryption algorithm being used, and the probability that the entire key can be replicated from the distributed encryption environment. The key needs to be robust enough to meet current and future security needs while not being excessively large which could bog down computing resources, as does the algorithm used to perform the encryption. Meanwhile, perhaps the true unsung benefit of the distributed encryption method is being able to replicate the key if a node or multiple nodes are lost or destroyed. In a typical, centrally located key management system, if either a hacker compromises the key or some other disaster causes the key to be lost or destroyed, then the encrypted data becomes unrecoverable. In todays “Big Data” world the loss of data has undoubtedly become an even more frightening possibility than the exposure of data, something that has ruled the headlines for the last few years. If a company like Google were to somehow “lose” the vast amount of analytical data that it had accrued through its search engine, its ability to accurately forecast market tendencies would be greatly compromised, at least in the short term. Those types of dire occurrences might represent a very small window of high value targets, but the increasing use of mobile devices has opened up an ever-increasing amount of potential attack vectors for malicious hackers to try and exploit.

Therefore, a distributed encryption system might not represent a viable option for all types of authentication however in those instances when a high level of security is required it may be a very useful tool. The balancing act then becomes making sure that the level of security robustness does not negatively affect system performance, which more often than not encompasses business requirements. Those requirements will most often hinge upon timing measures, whether an application allows access within a timely manner, typically three seconds is considered the maximum an end-user is willing to wait before it becomes an issue. The factors that were discussed in this paper to maximize the security robustness could all potentially cut into that seemingly large time frame when considering computing timeframes, but three seconds is very short on a human-scale window of time. Thus it should be noted that it takes only one delay in the system to potentially push it out of the “sweet spot” of three seconds, which could run afoul of most end users. The application of a distributed encryption system on a cloud environment for certain uses could be extremely beneficial if the added security benefits outweighed the potential drawbacks of time delays. With the advent of the BYOD movement, a distributed encryption system could provide a means to reliably authenticate a wide and disparate variety of mobile devices and lock down the data that they access on a cloud based computing environment.

REFERENCES

1. Abdalla, M., Bellare, M., & Neven, G. (2008). Robust Encryption, *Cryptology ePrint Archive*, *2008/440*. Retrieved from http://eprint.iacr.org/2008/440.pdf.
2. Barker, E., & Roginsky, A. (2012). Recommendation for Cryptographic Key Generation, *NIST Special Publication 800-133.* Retrieved from http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-133.pdf.
3. Bienfang, J. (2012). Truly Random Numbers – But Not by Chance. Retrieved from http://www.nist.gov/pml/div684/random_numbers_bell_test.cfm.
4. Boyd, C., & Mathuria, A. (1998). *Protocols for Authentication and Key Establishment.* Berlin, Germany. Springer.
5. Cloud Security Alliance. (2010). Top Threats to Cloud Computing, *Version 1.0.* Retrieved from https://cloudsecurityalliance.org/topthreats/csathreats.v1.0.pdf.
6. Eastlake, D., Crocker, S., & Schiller, J. (1994). Randomness Recommendations for Security, *IETF RFC 1750*. Retrieved from http://www.ietf.org/rfc/rfc1750.txt.

7. Filipe, J., & Obaidat, M. (2008). Two Types of Key-Compromise Impersonation Attacks against One-Pass Key Establishment Protocols, *Communications in Computer and Information Science*, *23*, 227-238.
8. Giry, D. (2013). BlueKrypt - Cryptographic Key Length Recommendation. Retrieved from http://www.keylength.com/en/4/.
9. Goodin, D. (2012, December 9). 25-GPU Cluster Cracks Every Standard Windows in <6 Hours: All your passwords belong to us. *Arstechnica*. Retrieved from http://arstechnica.com/security/2012/12/25-gpu-cluster-cracks-every-standard-windows-password-in-6-hours/.
10. Guster, D., Brown, C., & Sultanov, R. (2010). Enhancing Key Confidentiality by Distributing Portions of the Key Across Multiple Hosts, *Midwest Instruction and Computing Symposium, 43rd MICS*. Retrieved from http://micsymposium.org/mics_2010_proceedings/paper_11.pdf.
11. Herong, Y. (2013). Cryptography Tutorials – Herong's Tutorial Examples. Retrieved from http://www.herongyang.com/Cryptography/.
12. Mearian, L. (2009, February 17) Sun offers open source encryption key management protocol. *ComputerWorld*. Retrieved from http://www.computerworld.com/s/article/9128101/Sun_offers_open_source_encryption_key_management_protocol
13. Moore, F. (2005, August 1). Preparing for Encryption: New Threats, Legal Requirements Boost Need for Encrypted Data, *Computer Technology Review*. Retrieved from http://www.gbdata.co.uk/site/wp-content/uploads/2010/06/Preparing-for-encryption.pdf
14. Moore's Law. Retrieved March 10, 2013 from Wikipedia: http://en.wikipedia.org/wiki/Moore's_law.
15. Postma, A., De Boer, W., Helme, A., & Smit, G. (1996). Distributed Encryption and Decryption Algorithms. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.3186.
16. Redman, J., Rice, E., Han, S., Anderson, R., Schwarting, J., & Paulson, B. (2013). Can A Distributed Key System Broken Up Over Multiple Nodes Provide Greater Security Robustness While Meeting System Performance Requirements?, *Presented at the Midwest Instruction and Computing Symposium*, *46th MICS*, La Crosse, Wisconsin.
17. Rego, I., Gillespie, N., & Guster, D. (2011). Implementing a Distributed Key Algorithm to Enhance Confidentiality and Provide Fault Tolerance, *Midwest Instruction and Computing Symposium*, *44th MICS*. Retrieved from http://micsymposium.org/mics_2011_proceedings/mics2011_submission_8.pdf.
18. Ristenpart, T., & Yilek, S. (2010). When Good Randomness Goes Bad: Virtual Machine Reset Vulnerabilities and Hedging Deployed Cryptography, *NDSS Symposium 2010.* Retrieved from http://www.internetsociety.org/doc/when-good-randomness-goes-bad-virtual-machine-reset-vulnerabilities-and-hedging-deployed.
19. RSA (algorithm). Retrieved May 28, 2013 from Wikipedia: http://en.wikipedia.org/wiki/RSA_(algorithm).
20. Shah, N. B., Rashmi, K. V., & Vijay Kumar, P. (2011). Information-theoretically Secure Regenerating Codes for Distributed Storage, *Globecom 2011.* Retrieved from http://www.eecs.berkeley.edu/~nihar/publications/secrecyInStorage.pdf.
21. Tseng, Y. (2003). A Scalable Key Management Scheme with Minimizing Key Storage for Secure Group Communications, *International Journal of Network Management. 13*, *Issue 6*, 419-425.
22. Xu, L. (2005). Hydra: A Platform for Survivable and Secure Data Storage Systems. Retrieved from https://gnunet.org/sites/default/files/w8paper13.pdf.
23. Ylonen, T., & Lonvick, C. (2006). The Secure Shell (SSH) Authentication Protocol, *IETF RFC 4252*, Retrieved from http://tools.ietf.org/html/rfc4252