Analysis of Implementations to Secure Git for Use as an Encrypted Distributed Version Control System

Russell G. Shirey, Kenneth M. Hopkinson, Kyle E. Stewart
Douglas D. Hodson, Brett J. Borghetti

Department of Electrical and Computer Engineering
Air Force Institute of Technology (AFIT)
2950 Hobson Way, Wright-Patterson AFB, Ohio 43433, USA

{russell.shirey, kenneth.hopkinson, kyle.stewart, douglas,hodson, brett.borghetti}@afit.edu

Abstract

This paper analyzes two existing methods for securing Git repositories, Git-encrypt and Git-crypt, by comparing their performance relative to the default Git implementation. Securing a Git repository is necessary when the repository contains sensitive or restricted data. This allows the repository to be stored on any third-party cloud provider with assurance that even if the repository data is leaked, it will remain secure. The analysis of current Git encryption methods is done through a series of tests that examines the performance trade-offs made for added security. This performance is analyzed in terms of size, time, and functionality using three different Git repositories of varying size. The three experiments include initializing and populating a repository, compressing a repository through garbage collection, and modifying then committing files to the repository. The results show that Git maintains functionality with each of these two encryption implementations at the cost of time and repository size. The time increase is found to be a factor ranging from 14 to 38 times the original time. The size increase over multiple commits of edited files is found to increase linearly proportional to the working set of files.

1. Introduction

The last half century has been a time for rapid growth in technology, specifically with regard to computers. Computers have evolved from expensive, military developed systems, to large mainframes used by Fortune 500 companies, to the advent of the personal computer, to the most recent mobile and cloud computing on minimal devices. Memory and storage have always been an integral part of any computer system and as systems have modernized, file systems have been created and evolve in order to keep up. Version control is often used by software developers and enables them to keep historical versions of source code and project files for access at a later time, while also enabling complex merging between multiple versions of the same file [1].

One popular distributed version control system is Git. It is open source and handles revisions for large source code repositories. Git does not encrypt source code files, which potentially limits its use for sensitive projects-for example, many Department of Defense (DoD) projects. Sensitive means that the contents are restricted to a select access group. While unencrypted version control works if every computer is in a secure area, it is not possible to leverage non-secure storage mediums. Even if the network is secure, if one of the computers is compromised, the unencrypted file contents are then accessible. It is critical to have an encrypted repository in this case. With the ease of use of cloud computing today, if the repository can be securely stored in the cloud, it provides efficiencies in not having to create a separate network and host the data. The entire repository can be encrypted and transferred, however, this requires a large overhead for small file changes as the entire contents must be encrypted and transferred, not just the specific files that have changed.

A literature search for Git encryption found no results. Two open source Git encryption implementations exist: Git-encrypt and Git-crypt.

1530-1605 2015
DOI 10.1109/HICSS.2015.625

Both implementations are more secure than unencrypted Git. While Git-crypt is the more secure of the two implementations, it doesn't allow as much compression as Git-encrypt. This study examines these two implementations for encrypting Git repositories as well as unencrypted Git because the three implementations adequately represent the trade-space balance between performance and security. They are tested with unencrypted Git in order to provide a side-by-side performance comparison. The areas of performance examined are size, time, and functionality. They are tested over three scenarios of Git commands over multiple repetitions. These three experiments include: 1) initializing and populating a repository, 2) compressing a repository through garbage collection, and 3) modifying then committing files to the repository. There are three distinct existing Git repository test subjects: the Linux Kernel, Git source code, and Popping iOS application. These test subjects vary in size and complexity. The analysis of the performance finds that encryption increases CPU time from 14 to 38 as compared to standard (non-encrypted) Git performance. The size increase over multiple commits of edited files is found to increase linearly proportional to the working set of files for Git-crypt, and nearly the same as unmodified Git for Git-encrypt. Both of these Git-encryption methods are found to be functionally transparent to the Git user and enable secure storage of repositories on a third-party file hosting or cloud service, with only the encryption key holders able to access the information within the repository.

This research introduces version control and Git and discusses the differences between Git and other version control systems. It then describes related work, introducing the two existing Git-encryption implementations: Git-encrypt and Git-crypt. The test methodology is then explained. Then the results of the testing are explained. Lastly, the conclusion of the research is discussed and determines that Git-encryption can be used for sensitive projects.

2. Introduction to Version Control with Git

At its core, a version control system allows multiple users to store changes between different versions of files and switch between versions or merge them with ease. Some popular features prevalent in most version control systems is ability to backup and restore files, synchronization of files, undoing changes, track ownership, and branching and merging [1]. All of these combined promote efficient software development and supports the processes associated with configuration management.

Some file systems have version control built in, such as Google File System and Bigtable [2.3]. A dedicated version control system, however, benefits as it has functionality specific to the task of version control. There are several popular open source and commercial version control systems that have been developed through the years. Clearcase was an early commercial product developed by IBM in the 1990s and implemented configuration management version control [1]. Subversion is currently a very popular open source version control system. These are both examples of centralized version control systems.

Centralized version control systems are characterized by a single source repository. This single, so-called master repository is accessed by all developers to check out and check in version commits (i.e., revisions) [4]. There should be a limited list of who has access to write to the central repository. This type of system has worked well in the past, but poses a challenge with regard to scalability and limitations in work flow.

To get around this, decentralized version control systems have been developed. These include systems such as Git, Mercurial, Bazaar, and BitKeeper. These allow developers to check out or clone an existing repository for their own use and have full rights for that instance. Because each branch in a distributed repository is a full repository, a canonical branch is identified by convention within the development group. Some projects may have several principle branches [4]. Distributed version control systems also provide multiple backups in case of a failure of one of the repositories and limit the load placed on a single repository.

Linus Torvalds published the first version of Git in April 2005 [5]. Git was originally designed and developed for Linux Kernel management. It has since grown very popular and is now used in many software projects. Git fully mirrors each repository in a distributed environment and stores full snapshots of each file version in a repository, with references to any file that changed, similar to a mini file system. This approach gives Git a very powerful branching functionality. Other version control systems store changes made to individual files over time, rather than snapshotting the whole repository every commit [6]. *Figure 1* shows a visual example of Git storage:

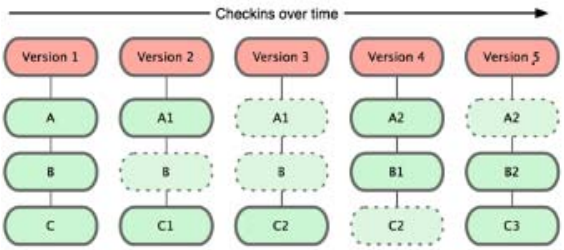

Figure 1. Git versions, www.Git-scm.org

Integrity is built into the internal structure of Git. Every file in a Git repository is check summed with a SHA-1 hash, a one-way function with arbitrary long input and a pseudorandom and fixed length output, when checked in and items are stored by hash reference instead of file name [6]. The data object is called a blob and stores the contents of a file.

3. Related Work

Although nothing in the published literature was found concerning a secure Git implementation, there is much published research on securing the public cloud [7,8], deterministic authenticated cryptography [9,10], other version control systems [1,4,5], and work which uses Git as a file-system [11,12]. There are two different open source Git-encryption schemes that can be found through web search [13,14]. Other implementations can be found, but these two provided an ample sample set for initial testing, as discussed in this section. These implementations appear to be developed by individual hobbyists and lack much information or detail with regard to verification and validation testing. No apparent measures of security are documented, such as the effects on the encrypted data structures within Git. These implementations provide Git repository read access control through encryption by using Git filters.

The first implementation is Git-encrypt, a series of scripts written by Woody Gilk [13]. The second is Git-crypt, a C++ library written by Andrew Ayer [14]. Both use Git's smudge and clean filters (*Figure 2*), which allow custom code to intercept and edit data as it comes from or goes to the repository [6]. These two schemes encrypt and decrypt as data travels to and from the repository through the filters

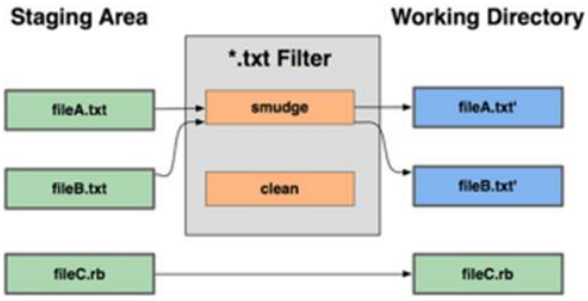

Figure 2. Git filters, www.Git-scm.org

Git-encrypt uses a series of scripts to take advantage of the filters [13]. Due to the nature of the command line scripts, it is much slower than C/C++ implementations. The program calls OpenSSL command line functions. The default cipher used is AES-256-Electronic Code Book (ECB), which is not semantically secure for messages containing more than one block of data due to susceptibility to chosen plaintext attack [15]. As an example of this, if an attacker sends a challenge of two, two-block messages, with similar plaintext, such as "Hello World" and "Hello Hello," the attacker will be able to easily determine which resulting ciphertext corresponds to the plaintext. The attacker can substitute known plaintext-ciphertext blocks in place of existing ones and foil the message. ECB mode is more secure than unencrypted Git and takes advantage of Git compression because identical plaintext blocks encrypt to identical ciphertext blocks. This type of encryption is chosen in this research to examine a range of encryption techniques, as the Git-crypt implementation is more secure but does not allow for the same compression as Git-encrypt.

In Git-encrypt, a user can change the cipher mode to any OpenSSL scheme, such as cipher block chaining [13]. One problem with alternative block modes is that there is no flexibility for selection of initialization vectors, a pseudorandom or random input to the cryptographic protocol providing randomness of encryption of the same data and key. The implemented initialization vector is not random, which results in insecure encryption for all schemes which require a random initialization vector. Additionally, there is no message authentication code (MAC). The MAC is used to verify the integrity of the message or file – that it has not been altered in any way [15].

Lastly, the user has the option of picking a password and a salt, a random value that is appended

to a password in order to provide more security to the password, or setting either to random [13]. If the user inputs both of these variables, there is no check to ensure they are indistinguishable from uniformly random. If two dictionary words are chosen and then concatenated together, the attacker will have an advantage of breaking the combination by trying dictionary words in the attack. If the user decides to select the random choice implemented in Git-encrypt, the code calls the Linux /dev/random function [13]. This function is proven to not be indistinguishable from uniformly random, and not secure [16].

Git-crypt is faster than Git-encrypt since it is written in C++ and it provides more cryptography functionality [14]. The algorithm uses AES-256 counter mode with an initialization vector derived from SHA-1-HMAC hash. This scheme of hashing the message and then encrypting it provides a random initialization vector derived from the message. This is because the hash is a pseudo random function with the input determined by the file. Since the file is used to generate the initialization vector, then deterministic authenticated encryption is achieved. This means the same plaintext will yield the same ciphertext, since the initialization vector is derived from the plaintext by a set function [7]. *Figure 3* shows an example of this at a high level:

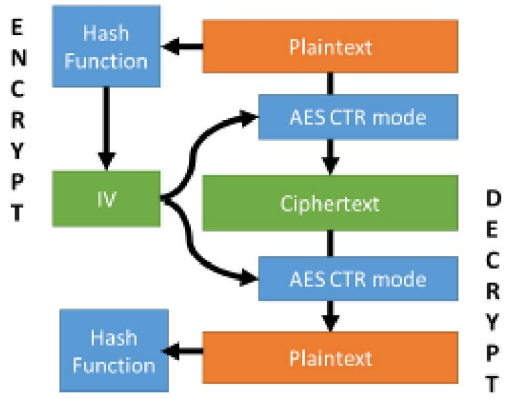

Figure 3. Hash and then encrypt [15]

The encryption in *Figure 4* uses two separate keys: the first key is used by the hash function to generate the initialization vector used by the encryption mode. Integrity is provided as the decrypted message is passed to the hash function to see if the output initialization vector matches what was received. If they match, it is determined that the message was sent from someone with a correct key and has not been altered [15].

Since Git typically involves several parties working with the same information, a plaintext query must yield the same ciphertext query. Combining authentication and encryption is a good practice and works in many scenarios [15]. Deriving the initialization vector from the hash of the message also determines (with very high probability) that the same initialization vector will not be reused to encrypt a different message.

In Git-crypt, the key is derived using the RAND_bytes_openssl function. This key is stored unencrypted locally, which could be destructive if malicious attackers could access the hard drive and reveal it, so the user must protect their own key [14].

There have been complaints about using filters to encrypt Git on the basis that it takes away from the lightweight and efficient design [17]. These are valid concerns, but in order to fully collaborate in a seamless and non-intrusive manner with encryption transparent to Git, a Git encryption implementation is needed. There is usually a performance trade-off for security, and designers must make sure the benefit outweighs the cost.

4. Methodology

In the previous sections, existing implementations of Git encryption are introduced and discussed. This research analyzes the performance of these encryption implementations in terms of CPU time, size, and functionality. Three distinct real-world Git repositories are tested with different function scenarios. The results show how each test compares in performance between the two Git encryption implementations and unencrypted Git.

The first test repository is the Linux Kernel, selected for its wide-spread use and enterprise-like structure. Second is the Git program source code. This selection represents a medium sized project with wide-spread use. Lastly, a small-scale program called Popping was found by looking through the popular repositories on Github, an online storage and sharing area for Git repositories. It is developed by Schneiderandre and is a collection of animation examples of Apple iOS applications. The size characteristics of each is shown in *Table 1*.

Table 1. Test Program Sizes

Program	Size	Objects	Commits
Linux Kernel	135.8 MB	48510	451,700
Git Program	5.18 MB	2689	36,684
Popping Program	0.11 MB	179	95

4.1. Experiment 1: Adding all files to the initial repository

The first test characterizes the performance of initializing a brand new repository. This scenario is the worst-case for a project in terms of computing because every file is new to the repository and is passed through the encryption filters. This test is performed by initializing a new repository using the 'git init' command. Next, all of the files of the existing repository to be tested are copied to this new repository folder location. Then the 'git add' command is run with a wildcard argument, which stages all the new files, regardless of name. To stage the files, Git passes them through the Git filters as they are staged. Next, the 'git commit' command is run, committing the files in the staging area to the repository. This complete process is timed and repeated a total of ten iterations for consistency of results.

4.2. Experiment 2: Initial size comparison

Once the files have been committed, the size of each repository is recorded and compared each other. This comparison is done using the 'git count-objects' command. The size is recorded and then the garbage collector is run. The garbage collector in Git compresses the data within the Git repository by using a set of heuristics programmed in the internal structure of Git. This garbage collection is automatically run during some circumstances or can be run manually using the 'git gc' command. After running this command, the size of the repositories is compared to see how the unencrypted Git data compression compares to the data compression of two Git-encryption implementations.

4.3. Experiment 3: Size growth with file modifications

The final test measures the worst-case scenario of editing all files in the repository. Only a small number of bytes are edited, but since every file is changed, every file must once again traverse through the Git filters to the staging area and be committed. This test is used for comparison to see how much a small change in data among a large number of objects has on the size of the repository. The test characterizes the inefficiency of the storage size of encrypted Git and determines the effectiveness that the garbage collection has in compression over a series of commits. The test runs a script to append the text "hello" to all files within the repository. The files are then added to the staging area using the 'git add' command. Then the files are committed to the repository using the 'git commit' command. The Linux Kernel test is performed by running this method five times and then running the garbage collector after the fifth iteration. The Git program test is performed by running this same test but running garbage collection after each iteration, to see if there are any differences that can be seen verses waiting until the last iteration as with the Linux Kernel test. Differences in the compression of the data based on when the garbage collection is run show in this method of testing. Lastly, the Popping program follows the Git program test procedure.

The test bench for these test cases is a basic Linux environment. Specifically they are run on Ubuntu-64 bit operating system running on a virtual machine within VMWare Workstation utilizing one Intel Core I7 (3.6GHZ) processor and 4GB of RAM.

5. Experimental Results

The previous section describes the methodology for testing. This section examines the results of the tests. The results show that functionally Git performs the tested commands identically under all three implementations. This is because the encryption schemes use the Git filters that are intended for modifying data in a transparent manner to Git.

5.1. Experiment 1: Adding all files to the initial repository

The first test subject is the Linux Kernel. The results are shown in the below *Figure 4*:

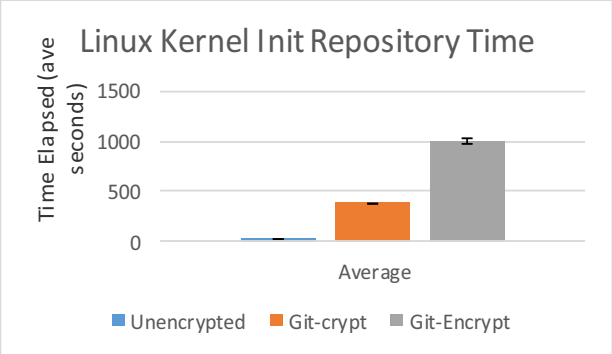

Figure 4. Experiment 1: Linux Kernel

The performance time for running through the filters of the encryption slows down the staging and commit process by a factor of 14 for Git-crypt, and a factor of 38 for Git-encrypt. The Linux Kernel is very large compared to most Git repositories yet this time increase does not make it infeasible to use, if security is desired. The exact CPU time averages for each of the three implementations is shown in *Table 2*:

Table 2. Experiment 1: Linux Kernel

Type:	Ave CPU Time:	Std Dev:
No Encryption	26.2 seconds	0.95
Using Git-crypt	380.1 seconds	8.46
Using Git-encrypt	1003.9 seconds	40.65

The second test subject is the Git program. The results of experiment one are shown in *Figure 5:*

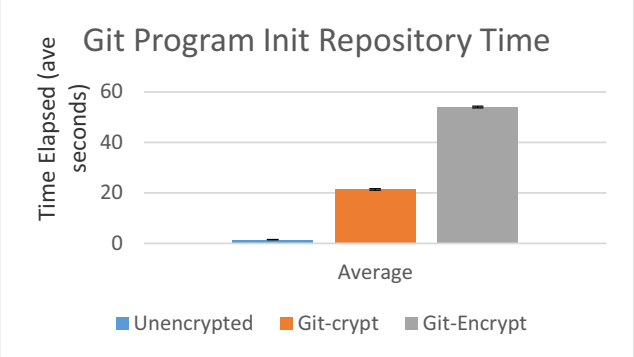

Figure 5. Experiment 1: Git Program

The time increase for encryption is worse compared to with the Linux kernel. The initialization time is slowed by a factor of 14 for Git-crypt, and a factor of 37 for Git-encrypt. This result is very close to the performance of the Linux Kernel. The Git program is a much smaller project in terms of size and the time reflects that. The exact averages in time for each of the three implementations is shown in *Table 3*:

Table 3 Experiment 1: Git Program

Type:	Ave Time:	Std Dev:
No Encryption	1.47 seconds	0.189
Using Git-crypt	21.36 seconds	0.456
Using Git-encrypt	54.04 seconds	0.518

The final test subject is the Popping Program. The results are shown in *Figure 6*:

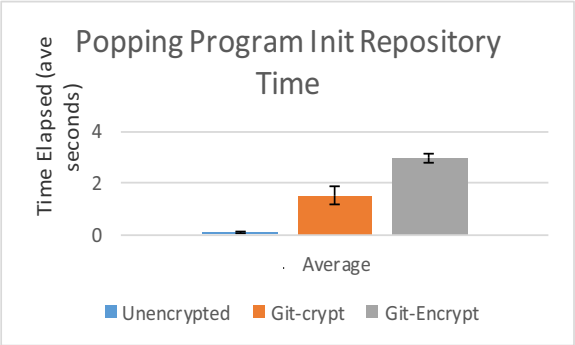

Figure 6. Experiment 1: Popping Program

The performance time for initializing a Git repository increases by a factor of 18 for Git-crypt, and a factor of 35 for Git-encrypt. These factors are higher than the performance of the Linux Kernel and less than with the Git program. The exact averages in time for each of the three implementations are shown in *Table 4*:

Table 4. Experiment 1: Popping Program

Type:	Ave Time:	Std Dev:
No Encryption	0.085 seconds	0.019
Using Git-crypt	1.501 seconds	0.506
Using Git-encrypt	2.980 seconds	0.247

5.2. Experiment 2: initial size comparison

In terms of size, unencrypted Git yields a repository roughly half the size of both encrypted versions. When garbage collection is run, the unencrypted Git repository reduced 50%, whereas the size of the repository using the encrypted implementations of Git is only reduced by 20%. This is because Git's garbage collection algorithm is unable to combine certain parts of the blob that have similar plaintext but very different ciphertext. The graph of results is in *Figure 7*.

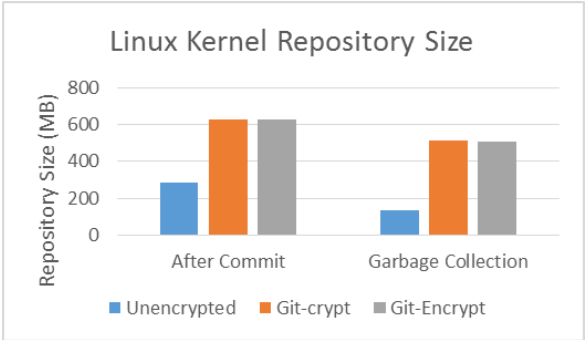

Figure 7. Experiment 2: Linux Kernel

The size of each implementation directly after the commit and after garbage collection is run is shown in *Table 5*:

Table 5. Experiment 2: Linux Kernel

Type	Size	After GC
No Encryption	287 MB	135 MB
Using Git-crypt	627 MB	514 MB
Using Git-encrypt	628 MB	508 MB

The size of unencrypted Git yields a repository 80% of the size of both encrypted versions. When garbage collection is run, the unencrypted Git repository is reduced by 75%, whereas the size of the repository using the encrypted implementations of Git is only reduced by 20%. This is for the same reasons as the Linux Kernel. The graph is shown in *Figure 8*:

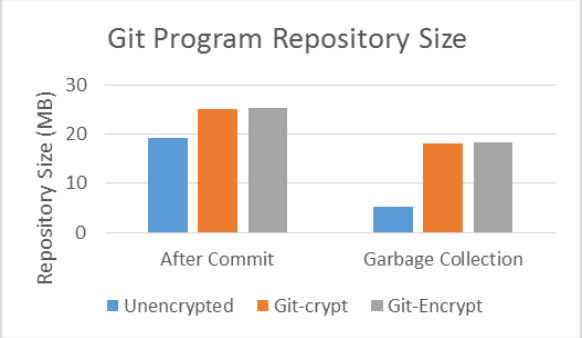

Figure 8. Experiment 2: Git Program

The size of each implementation directly after the commit and after garbage collection is run is shown in *Table 6*:

Table 6. Experiment 2: Git Program

Type	Size	After GC
No Encryption	19.2 MB	5.2 MB
Using Git-crypt	25.2 MB	18.2 MB
Using Git-encrypt	25.4 MB	18.3 MB

The Popping program yields a repository 80% the size of both encrypted versions. When garbage collection is run, the unencrypted repository is reduced by 75%, whereas the size of the repository using the encrypted implementations of Git are reduced by 20%. This is the same type of reduction as the previous two programs. The graph of results is in *Figure 9*:

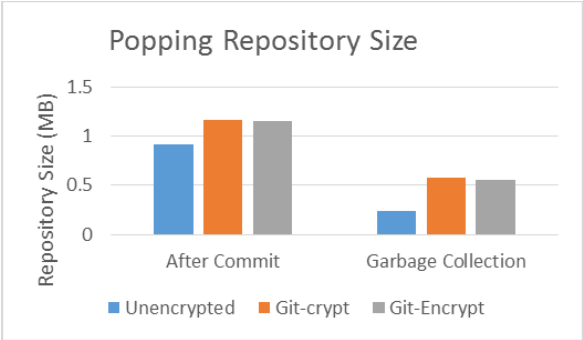

Figure 9. Experiment 2: Popping Program

The size of each implementation directly after the commit and after garbage collection is run is shown in *Table 7*:

Table 7. Experiment 2: Popping Program

Type	Size	After GC
No Encryption	0.916 MB	0.1 MB
Using Git-crypt	1.16 MB	0.58 MB
Using Git-encrypt	1.156 MB	0.56 MB

5.3. Experiment 3: Size growth with file modifications

The final experiment is performed by editing the end of every file by adding "hello" and then staging all the modified files and committing them to the repository. The growth in size is measured as this process is repeated five times. The repositories all grow at the same rate of adding 287 MB of data every iteration. This is because there is no packing. After the fifth iteration, garbage collection is run and the results are shown in the *Figure 10*:

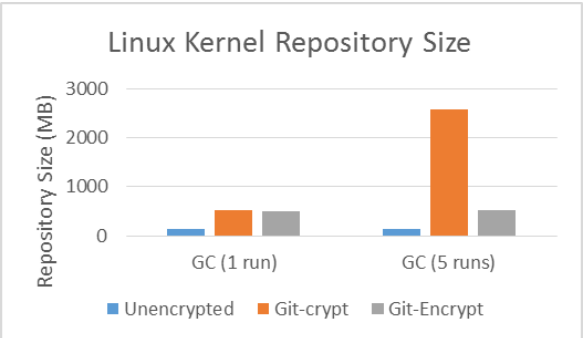

Figure 10. Experiment 3: Linux Kernel

The results show that the size of Git-crypt grows very fast. This is due to the higher entropy and failure to be condensed as before. The unencrypted and Git-encrypt repository hardly grow compared to the size growth after one run. The exact values are shown in *Table 8*:

Table 8. Experiment 3: Linux Kernel

Type	Size
No Encryption	152 MB
Using Git-crypt	2577 MB
Using Git-encrypt	525 MB

In the same manner as the Linux Kernel, the Git program grows linearly for Git-crypt, and does not grow very much for the other two implementations. The difference in this test is that the garbage collector is run after each iteration, but is found that when the garbage collection is run does not have a substantial effect on the fifth iteration. The results of this test are shown in *Figure 11*:

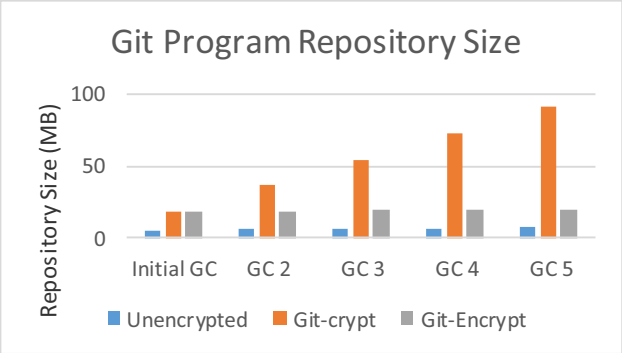

Figure 11. Experiment 3: Git program

Again, the size of Git-crypt grows very fast in a linear manner relative to the other implementations for the same reasons as the previous case. This result is the same as the Linux Kernel test. The exact values are in *Table 9*:

Table 9. Experiment 3: Git program

Type	2 GC	3 GC	4 GC	5 GC
No Encryption	5.8 MB	6.2 MB	6.7 MB	7.2 MB
Using Git-crypt	34.4 MB	54.6 MB	72.9 MB	91.1 MB
Using Git-encrypt	18.5 MB	18.8 MB	19.1 MB	19.4 MB

The Popping program performs similarly to the Git program. It grows linearly for Git-crypt, and does not grow fast in size for the other two implementations. This test is again run using the garbage collector after each iteration. The results of this test are shown in *Figure 12*:

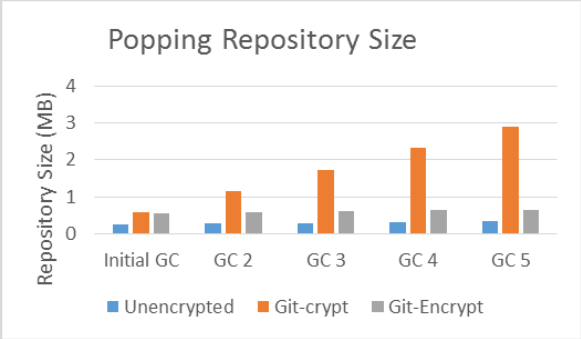

Figure 12. Experiment 3: Popping program

The size of Git-crypt grows very fast in a linear manner. This growth is for the same reasons as the previous two cases. The exact values are shown in *Table 10*:

Table 10. Experiment 3: Popping program

Type	2 GC	3 GC	4 GC	5 GC
Without encryption	.266 MB	.287 MB	.306 MB	.329 MB
Using Git-crypt	1.155 MB	1.731 MB	2.31 MB	2.89 MB
Using Git-encrypt	.584 MB	.607 MB	.631 MB	.653 MB

The three experiments are summarized in *Table 11*. Git without encryption is shown as the baseline and the other methods are compared to it for each of the experiments and test sizes:

Table 11. Summary of Experiments

Type of Test (size category)	No Encryption (baseline)	Git-Crypt	Git-Encrypt
Speed ratio of repository init (Small)	1.00	17.66	35.06
(Medium)	1.00	14.53	36.76
(Large)	1.00	14.51	38.32
Size ratio of initial repository (Small)	1.00	5.8	5.6
(Medium)	1.00	3.5	3.52
(Large)	1.00	3.81	3.76
Size ratio of growth with file changes after 5 iterations (Small)	1.00	8.78	1.98
(Medium)	1.00	12.65	2.69
(Large)	1.00	16.95	3.45

6. Conclusion and Future Work

This study examines two existing methods for securing Git repositories, Git-encrypt and Git-crypt, and compares their performance relative to the default Git implementation. They are tested side-by-side to unencrypted Git through a series of three tests. These tests examine the performance impact in terms of size, time, and functionality of initializing and populating a repository, compressing a repository through garbage collection, and modifying then committing files to a repository. From the results in the previous section, the two existing Git-encryption implementations are shown to provide full functionality for these tasks. They increase the time to execute Git functions with the time increase ranging from a factor of 14 to a factor of 38, depending on the scenario. The size increase for the Git-encryption implementations in initializing the repository is larger than default Git by a factor ranging from 3 to 9. The size increase of editing and then committing the repository files was similar for unencrypted Git and Git-encrypt and was found to increase linearly for Git-crypt, proportional to the working set of files. The reason for this is that in Git-encrypt with ECB mode, if a few bytes are altered, then the ciphertext alteration is limited to the blocks that correspond to those bytes. In Git-crypt, if even just one byte is altered, then the entire ciphertext is changed. As discussed in section three, the default implementation of Git-encrypt is not cryptographically secure, however, it is more secure than unencrypted Git and provides an interesting middle-ground test case of higher performance for a reduced level of security. Git-crypt is as cryptographically secure as the underlying AES and hash implementations it uses and depends on the IV being unique as well as a pseudorandom hash function.

As discussed in the introduction, a system is needed to provide a secure implementation for Git, so that the benefits of Git can be used in sensitive and restricted projects stored in unsecure areas, such as a public cloud. Currently, there is no formal method for this and to keep the git repository confidential the entire repository must be encrypted and transferred with every update to the repository.

These tests represent the worst-case scenario in that every file in the repository is added and then edited. These tests were run on a basic virtual machine in a controlled environment. Future tests should be run in a cloud environment and also tested on more powerful hardware to emulate enterprise business. Future work that will be beneficial is to introduce a system to accurately modify a wider range of files with a varying size of modifications through a series of commits based on current practices of work in Git repositories. This will make for more realistic testing to determine usability of these encryption implementations. The remaining functionality of Git also needs to be tested with these implementations with more test subjects included from varying coding languages, plus other file types such as sounds, images and video. Research into an encryption scheme that goes beyond using Git filters is valid research in this area as well. In this, the performance can be analyzed and the performance trade-offs in a Git-encryption system can possibly be reduced. Lastly, a secure key creation and storage mechanism needs to be added before these two schemes can be implemented in real-world sensitive repositories.

7. Acknowledgement

The views expressed in this document are those of the authors and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government.

8. References

[1] N. B. Ruparelia. The history of version control. ACM SIGSOFT Software Engineering Notes 35(1), pp. 5-9. 2010.

[2] Ghemawat, Sanjay, Howard Gobioff, and Shun-Tak Leung. "The Google file system." ACM SIGOPS Operating Systems Review. Vol. 37. No. 5. ACM, 2003.

[3] Chang, Fay, et al. "Bigtable: A distributed storage system for structured data."ACM Transactions on Computer Systems (TOCS) 26.2 (2008): 4.

[4] B. De Alwis and J. Sillito. Why are software projects moving from centralized to decentralized version control systems? Presented at Cooperative and Human Aspects on Software Engineering, 2009. CHASE'09. ICSE Workshop on. 2009.

[5] D. Spinellis. Git. Software, IEEE 29(3), pp. 100-101. 2012.

[6] Scott Chacon, "Pro Git," Springer-Verlag New York, Inc., New York, NY

[7] Snehil Suresh Wakchaure, Simrit Kaur Arora, "Implementation of a Secure Distributed Storage System."

[8] C. Wang, Q. Wang, K. Ren, N. Cao and W. Lou. Toward secure and dependable storage services in cloud computing. Services Computing, IEEE Transactions on 5(2), pp. 220-232. 2012.

[9] Phillip Rogaway, Thomas Shrimpton, "Deterministic Authenticated-Encryption, a Provable-Security Treatement of the Key-Wrap Problem," Advances in Cryptology – Eurocrypt '06.

[10] D. Harkins, "Synthetic Initialization Vector (SIV) Authenticated Encryption Using the Advanced Encryption Standard (AES)," RFC 5297 IETF, http://tools.ietf.org/search/rfc5297

[11] Joey Hess. Git-annex. http://Git-annex.branchable.com/, September 2012.

[12] Jan Philipp Luhrig, File synchronization using Git-annex assistant." http://media.itm.uni-luebeck.de/teaching/ws2013/sem-cloud-computing/File_synchronization_using_Git-annex_assistant.pdf

[13] Woody Gilk, Git-encrypt, https://Github.com/shadowhand/Git-encrypt

[14] Andrew Ayer, Git-crypt, https://Github.com/AGWA/Git-crypt

[15] Dan Boneh, "Introduction to Cryptography", https://class.coursera.org/crypto-preview/lecture

[16] Bruce Schneier, "Insecurities in Linux /dev/random" https://www.schneier.com/blog/archives/2013/10/insecurities_in.html

[17] E-mail from Junio C Hamano, http://article.gmane.org/gmane.comp.version-control.Git/113221