

Air Force Institute of Technology

AFIT Scholar

Theses and Dissertations

Student Graduate Works

3-26-2015

Git as an Encrypted Distributed Version Control System

Russell G. Shirey

Follow this and additional works at: <https://scholar.afit.edu/etd>



Part of the [Computer Engineering Commons](#)

Recommended Citation

Shirey, Russell G., "Git as an Encrypted Distributed Version Control System" (2015). *Theses and Dissertations*. 57.

<https://scholar.afit.edu/etd/57>

This Thesis is brought to you for free and open access by the Student Graduate Works at AFIT Scholar. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of AFIT Scholar. For more information, please contact AFIT.ENWL.Repository@us.af.mil.



GIT AS AN ENCRYPTED DISTRIBUTED VERSION CONTROL SYSTEM

THESIS

Russell G. Shirey, Captain, USAF

AFIT-ENG-MS-15-M-022

**DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY**

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

DISTRIBUTION STATEMENT A.
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

AFIT-ENG-MS-15-M-022

GIT AS AN ENCRYPTED DISTRIBUTED VERSION CONTROL SYSTEM

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the
Degree of Master of Science in Computer Engineering

Russell G. Shirey, BS

Captain, USAF

March 2015

DISTRIBUTION STATEMENT A.
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AFIT-ENG-MS-15-M-022

GIT AS AN ENCRYPTED DISTRIBUTED VERSION CONTROL SYSTEM

Russell G. Shirey, BS

Captain, USAF

Kenneth M. Hopkinson, PhD (Chairman)

Douglas D. Hodson, PhD (Member)

Brett J. Borghetti, PhD (Member)

Abstract

This thesis develops and presents a secure Git implementation, Git Virtual Vault (GV2), for users of Git to work on sensitive projects with repositories located in unsecure distributed environments, such as in cloud computing. This scenario is common within the Department of Defense, as much work is of a sensitive nature. In order to provide security to Git, additional functionality is added for confidentiality and integrity protection. This thesis examines existing Git encryption implementations and baselines their performance compared to unencrypted Git. Real-world Git repositories are examined to characterize typical Git usage and determine if the existing Git encryption implementations are capable of efficient performance with regards to typical Git usage. This research shows that the existing Git encryption implementations do not provide efficient performance. This research develops an improved secure Git implementation, GV2, with transparent authenticated encryption. The fundamental contribution of this research is developing GV2 to perform Git garbage collection on plaintext data before encrypting the data. The result is a secure Git implementation that is transparent to the user with only a minor performance penalty, compared to unencrypted Git.

Acknowledgments

I would like to express sincere appreciation to my faculty advisor, Dr. Kenneth Hopkinson, committee members, Dr. Douglas Hodson and Dr. Brett Borghetti, and doctoral student, Kyle Stewart, for their guidance and support throughout the course of this research effort. Their insight and experience was certainly appreciated.

Russell G. Shirey

Table of Contents

	Page
Abstract	iv
Acknowledgments.....	v
Table of Contents	vi
List of Figures	viii
List of Tables	x
I. Introduction	1
Research Objectives	1
Background.....	2
Problem Statement.....	7
Organization	9
II. Literature Review	12
Chapter Overview.....	12
Cryptography	13
Version Control	20
Research Background.....	27
Summary.....	37
III. Methodology	38
Chapter Overview.....	38
Phase One: Secure Git Baseline Phase.....	41
Phase One, Experiment I: Adding all files to the initial repository.....	42
Phase One, Experiment II: Initial size comparison	42
Phase One, Experiment III: Size growth with file modifications.....	43

Phase Two: Git Characterization Phase	44
Phase Three: Secure Git Improvement Phase	53
IV. Analysis and Results.....	64
Chapter Overview.....	64
Phase One: Secure Git Baseline Phase	64
Phase One, Experiment I: Adding all files to the initial repository.....	65
Phase One, Experiment II: Initial size comparison	69
Phase One, Experiment III: Size growth with file modifications.....	73
Phase Two: Git Characterization Phase	81
Phase Three: Secure Git Improvement Phase	88
V. Conclusions and Recommendations	101
Chapter Overview.....	101
Conclusions of Research	101
Significance of Research	102
Recommendations for Action.....	102
Recommendations for Future Research.....	103
Summary.....	103
Bibliography	105

List of Figures

Figure 1. Percentage share for projects tracked by Open HUB [6]	6
Figure 2. ECB Mode Encryption	15
Figure 3. CTR Mode Encryption	16
Figure 4. GCM High Level Encryption [31]	18
Figure 5. GCM High Level Decryption [31]	19
Figure 6, Git Storage [1]	22
Figure 7. Git Objects [3]	23
Figure 8. Git Packfile and Index Example [3]	25
Figure 9. Git Smudge Filter [1].....	26
Figure 10. Git Clean Filter [1]	26
Figure 11. GitBAC Diagram [42]	28
Figure 12. BigQuery Java Query	46
Figure 13. RStudio repository push and size data.....	47
Figure 14. Git Push Data.....	48
Figure 15. Git Repository Size Data	49
Figure 16. GV2 Repository Size Increase Diagram.....	59
Figure 17. Sample .git Directory [72]	60
Figure 18. Phase One, Experiment I: Linux Kernel.....	66
Figure 19. Phase One, Experiment I: Git Program	67
Figure 20. Phase One, Experiment I: Popping Program	68

Figure 21. Phase One, Experiment II: Linux Kernel	70
Figure 22. Phase One, Experiment II: Git Program.....	71
Figure 23. Phase One, Experiment II: Popping Program.....	72
Figure 24. Phase One, Experiment III: Linux Kernel	74
Figure 25. Phase One, Experiment III: Git Program	76
Figure 26. Phase One, Experiment III: Popping Program	77
Figure 27. Random Repository Commit Average Data.....	83
Figure 28. Trending Repository Commit Average Data.....	85
Figure 29. Trending Repository Line of Code Average Addition / Commit.....	86
Figure 30. Trending Repository Line of Code Average Removal / Commit.....	87
Figure 31. JGit Git Program Push Time Comparison.....	90
Figure 32. Git Encrypted and Unencrypted Config File Comparison	92
Figure 33. Git Encrypted and Unencrypted Structure Size Comparison	92
Figure 34. Scrambled Blob .git Directory.....	93
Figure 35. Encrypted and Unencrypted Packfile Comparison.....	94
Figure 36. Linux Kernel Git and JGit Adding Initial Files Time Comparison.....	97
Figure 37. Git Program Git and JGit Adding Initial Files Time Comparison	98
Figure 38. Popping Program Git and JGit Adding Initial Files Time Comparison	98

List of Tables

Table 1. Phase One Test Program Sizes	42
Table 2. GitHub Random Repository Selection	50
Table 3. GitHub Monthly Trending Repository Selection.....	51
Table 4. Sample .git Directory Explained [72]	61
Table 5. Phase One, Experiment I: Linux Kernel.....	66
Table 6. Phase One, Experiment I: Git Program	68
Table 7. Phase One, Experiment I: Popping Program	69
Table 8. Phase One, Experiment II: Linux Kernel.....	70
Table 9. Phase One, Experiment II: Git Program	72
Table 10. Phase One, Experiment II: Popping Program.....	73
Table 11. Phase One, Experiment III: Linux Kernel	75
Table 12. Phase One, Experiment III: Git Program.....	77
Table 13. Phase One, Experiment III: Git Program.....	78
Table 14. Phase One: Summary of Experiments	79
Table 15. Random Repository Commit Statistics.....	82
Table 16. Trending Repository Commit Statistics.....	84
Table 17. GV2 Git Program Size Comparison	95
Table 18. Git and JGit Adding Initial Files Time Comparison.....	97
Table 19. GV2 Speed Performance Compared to Git-crypt In Adding Initial Files	100
Table 20. GV2 Encrypted Repository Size Compared to Git-crypt	100

GIT AS AN ENCRYPTED DISTRIBUTED VERSION CONTROL SYSTEM

I. Introduction

Research Objectives

The goal of this thesis is to find or develop a viable secure Git implementation that can be used while working with sensitive data to protect that data. Current mainstream implementations of Git do not provide cryptographic protection of source code [1-3]. To achieve the goal of this thesis, the research objective is to demonstrate a modification to Git that allows for use as a fully functional and secure distributed version control system that can be used for sensitive projects. This research defines secure as controlled read-only access and integrity protection. Controlled read-only access means ensuring that if anyone without the proper credentials accesses the repository, they are not able to read or decipher any bits of information contained therein. Integrity protection means that if the encrypted repository data is maliciously altered, anyone with the proper key will know the data has been compromised when they try to decrypt it. Fully functional means that the set of all Git commands work identically on the secure Git repository as they do a traditional unencrypted Git repository.

This is new research that has interest from DoD and other organizations who want to leverage software development using a third party cloud service provider, while retaining confidentiality of the source code. In the future, many traditional applications will be modified to support this same type of secure functionality. Added security often poses trade-offs, usually in terms of ease of use and performance, when dealing with applications. This research investigates these performance trade-offs to determine if using

secure Git is a viable alternative to the archaic zip and then encrypt method that is currently available to provide cryptographic protection to Git repositories.

Background

In software development, distributed version control systems are becoming increasingly popular as software projects are often developed in physically distributed work environments [4]. Git is one such distributed version control system and has been rapidly gaining users in recent years [5, 6]. In order to fully understand why Git was developed, how it has become so widely used, and, most importantly, why securing Git is necessary, a brief overview of computer and networking history is necessary.

Microelectronics and computers have experienced rapid growth in the past half century. Moore's Law has accurately predicted the growth of the number of devices per silicon die, which has set the pace of innovation in one of the most dynamic of the world's industries [7]. As microelectronic technology continues to evolve, allowing for more transistors on a smaller die, a wide range of innovative applications is plausible. Computers are one major benefactor of this technological revolution [8]. Taking advantage of the smaller and more powerful microprocessors, computers have evolved from expensive, military developed systems, to large mainframes used by Fortune 500 companies, to the advent of the personal computer.

In addition to the advancements yielding increases in power and decreases in size of computers, technology dealing with communication between computers, and other portable electronic devices, have made significant steps forward in terms of functionality and usability in the same timeframe [9]. Inter-device communication has evolved from

expensive, slow, proprietary protocols to the advent and growth of the internet, using widely accepted TCP/IP and UDP protocols. The adoption of protocol by users has enabled transferring information through interconnected networks across the world. This development has led to a dependence on the internet to increase personal and workspace productivity, as organizations no longer need to work in the same vicinity, but can have access to and share resources across a country, or even continents, with minimal time delay.

Bandwidth increases allow for more data to be sent at faster speeds [10]. This allows for real-time applications to be executed over the internet, such as video teleconferencing, video gaming, and collaborative office or document work. This has led to large growth in the area of cloud computing in the recent decade. The National Institute of Standards and Technology (NIST) went through 15 drafts to develop a paper to define cloud computing [11]. This thesis uses the definition that cloud computing is, “Internet-based computing in which large groups of remote servers are networked so as to allow sharing of data-processing tasks, centralized data storage, and online access to computer services or resources.” [12]

Using cloud computing services allows organizations or individuals to bypass the expensive costs and overhead of setting up and managing infrastructure [13]. They also have options to scale their cloud computing resources over time and rent only what they need from a cloud computing service provider, such as Amazon Web Services (AWS), Google AppEngine, or Microsoft Windows Azure, to name a few [14]. The elastic nature of scaling computing resources with cloud computing allows companies to save money

but contemplating cloud computing brings security concerns relating to access control or data security, as the company no longer has full control over their resources and security setup [15]. Additionally, some companies may question the service level agreements that the cloud service providers stand by, specifically in the areas of reliability and availability [16].

With the movement of computing from centralized to decentralized systems, file systems have also been created and evolved in order to keep up with these changes [17-22]. File systems now employ the options of storing data in many different formats, in a variety of locations, with many different protocols for security, access, and backup options. Since modern file systems have become so complex with multiple workers accessing the same files, version control systems are often used in order to keep track of changes, merge work together, and also to revert back to previous work when mistakes are made, or new changes are unwanted. These version control systems are especially popular with software developers, as it enables them to keep historical versions of source code and project files for access at a later time.

Both distributed and centralized version control systems exist [1]. Centralized version control systems are characterized by a single source repository from which all members check source code files in or out (with proper permissions to do so). Distributed version control systems are decentralized, allowing developers to check-out or clone existing repositories with full rights on that instance of check-out (if the access control allows). Every developer has their own repository and can choose whether they merge

with other repositories or not. Usually, there is an agreed upon ‘central’ repository location that is kept up to date and controlled by a select few.

One popular distributed version control system is Git [5]. Git is open source and handles revisions for repositories of varying sizes. In 2010, Subversion, a centralized version control system, accounted for more than 60% of the market for version-control systems, while Git only accounted for 2.7% of the market share. Redmonk analyst Stephen O’Grady performed a study of software version control systems used in 2013 and found that Git had increased its market share to 28%. Additionally, as of October 2014, there are over 250,000 Git projects, making up 37% of the repositories tracked by Open HUB, an online open source directory [6]. In 2013, GitHub, an online storage site devoted to hosting Git repositories in a centralized and easy to access location, touted over 6,000,000 projects [23]. This includes well-known projects such as Linux or Wordpress and also projects developed by large organizations, such as Google and Facebook. A main reason for the increase in usage of Git is the growing desire for software developers to transition from centralized to decentralized version control systems. Git’s usage of a distributed version control system allows for increased parallel work, reducing inefficiencies. Figure 1 gives a visual depiction of the market share split for current version control systems.

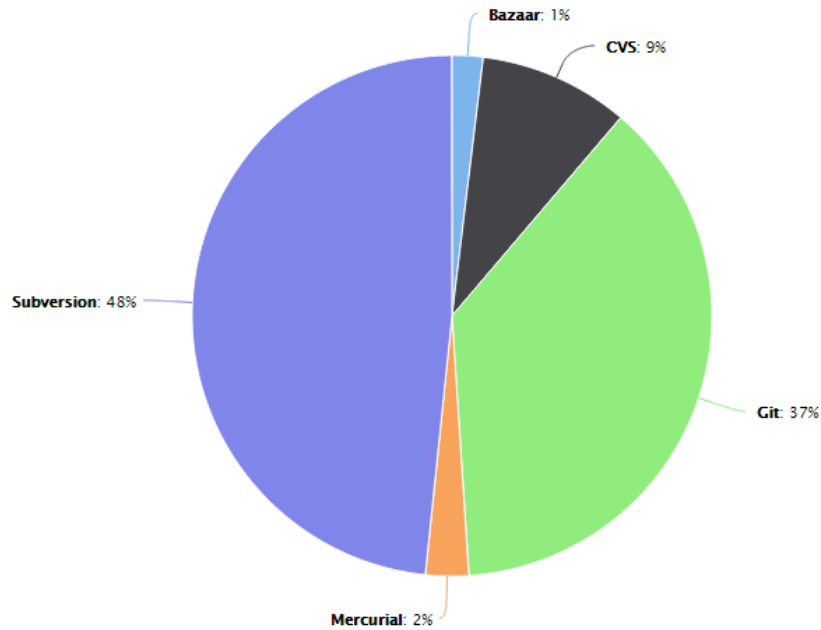


Figure 1. Percentage share for projects tracked by Open HUB [6]

Distributed version control systems are becoming increasingly popular as software projects are often developed in physically distributed work environments as global software engineering is growing [4]. Software version control systems are an addition to the development environment. Desired traits are to be lightweight, efficient, and easy to use. The popular ones have the option to store just the changes between files, called deltas, or compress the separate versions of files, depending on how the system works. This ensures that the repository does not grow in proportion to the sum of all of the sizes of all of the different versions of the files. This works well for unencrypted repositories, which is the environment in which the majority of distributed version

control systems are used, but does not work well on encrypted repositories, as the popular version control systems were not created to handle encrypted data.

Cloud computing offers highly efficient and scalable hardware and software resources, allowing enterprises to save money on computing expenses [13]. These efficiencies come with added security risk, as the cloud computing environment is accessible through the internet [24, 25]. If an enterprise works with sensitive data and wants to utilize the cloud, they must take a look into security. This is an area of little concern to most distributed version control systems used today.

Problem Statement

Git does not encrypt source code files, which potentially limits its use in sensitive projects - for example, many Department of Defense (DoD) projects. Sensitive projects require that repository contents are restricted to a select access group. While unencrypted version control works with sensitive projects if every computer is in a secure area, it is not possible to leverage non-secure storage mediums outside of the secure area. With the rapid adaptation of cloud computing today, if a repository can be securely stored in a cloud computing environment, it provides efficiencies that eliminate the need to create a separate secure network and host the data. The entire repository can be encrypted and transferred, however, this requires a large amount of overhead for small file changes as the entire contents of the repository must be encrypted and transferred, not just the specific files that have changed.

As an example, consider several developers working on the Linux kernel in a distributed environment. These developers are using a Git repository hosted by a cloud

service provider, and securing Git by zipping the repository and encrypting the zip file.

The Linux kernel is roughly 140 MB in size. In order to work on the repository in a secure manner, the steps are as follows:

1. A developer downloads the zip file from the cloud storage server (140 MB of bandwidth usage).
2. The developer decrypts and then uncompresses the zip file.
3. The developer checks out the unencrypted repository on his or her local machine.
4. The developer changes one character in one file.
5. The developer then adds the file and commits the change to the repository.
6. The developer zips the repository and then encrypts the zip file.
7. The developer then uploads the zip file to the cloud (140 MB of bandwidth usage).

This example of a one character change in a file requires 280 MB of bandwidth usage, plus an extra 140 MB of storage in the cloud, as both zip files must be stored on the cloud provider's storage services as different versions of the Linux kernel. If the Git server is used directly, then the user is able to save on bandwidth usage and also only add a fraction of the additional storage to the cloud. Aside from the wasted bandwidth, time, and monetary costs associated with large file uploads and data storage, the seamless functionality of Git is lost if the files have to be zipped and then transferred. A traditional Git repository provides detailed interaction whether it resides on a user's personal computer or an accessible server. When zipped, the repository cannot be easily examined

or mirrored to users. They have to download the zip file and then unzip before the data can be handled as a Git repository. This altogether takes away from the purpose of an efficient and easy to use distributed version control system. While the features of Git still exist once the decryption of the zip file and unzip of the repository is complete, Git becomes burdensome when used in this manner.

Organization

This thesis is intended to have an audience of computer programmers, especially those who are familiar with Git. It is relevant, however, for software engineers, computer engineers, and the information technology community as a whole, as it presents many ideas and concepts that are prevalent in today's rapidly growing cloud computing environment, mainly in regards to security and performance.

The remaining chapters of this thesis present background information relevant to this project and needed in order to fully comprehend all of the research ideas that this thesis encompasses. Chapter II reviews the basic concepts of cryptography and security principles before describing the specific security desired and why it is required for this research. The chapter then provides a more detailed look at the origins of version control systems and the functionality and internal structure of Git. Internal Git structure is necessary to understand when discussing the implementation of encryption within Git and recognizing performance trade-offs. Chapter II then discusses how version control systems are used today. Specifically, software engineering papers researching typical version control uses are discussed. Referenced papers provide background information for characterizing the usage of Git, which is discussed in a later section. Lastly, the

chapter discusses existing Git encryption implementations and the pros and cons of each from the multiple perspectives of security, functionality, and performance of the application.

Chapter III describes the methodology used in this research to demonstrate a usable secure Git implementation. This chapter consists of three phases:

1. Secure Git Baseline Phase: test existing Git encryption implementations to baseline their performance against unencrypted Git.
2. Git Characterization Phase: mine real-world Git repositories in order to provide realistic emulation of Git repositories over time to provide realistic testing and demonstrate that the proposed secure Git implementation works with typical Git repositories.
3. Secure Git Improvement Phase: develop and test a secure Git implementation that improves upon the previously tested versions from Phase One.

Chapter III focuses on the methodology for accomplishing each phase. This includes describing the process for each phase as well as describing the testing environment, to include the variables of the test, the test setup, and expected results.

Chapters IV presents and analyzes the results of each of the phases described in the previous chapters in detail. This chapter analyzes the tests and provides technical analysis of the results through a series of experimental statistics and graphs, in order to show the whole picture of what each test means to this research.

Lastly, Chapter V concludes this research. This final chapter summarizes this research, how it benefits the software engineering, computer engineering, and

information technology community, and suggests future research opportunities relating to this topic.

II. Literature Review

Chapter Overview

The Literature Review chapter provides necessary background information to better understand the research accomplished in this thesis. Related research is also discussed and explained in reference to contribution of the research presented. This chapter begins with a review of the basic concepts of cryptography and security principles. It then goes on to describe different cryptography protocols which are needed to attain the specific security requirements desired for a secure Git implementation upon which the goals of this research hinge. The chapter then provides a more detailed look at the origins of version control, the functionality, and internal structure of Git.

Internal Git structure is conceptually vital to understand when discussing the implementation of encryption within Git. Specifically, recognizing performance trade-offs when operating in a secure manner rather than unencrypted are reviewed. Next, the chapter discusses the need for a secure Git implementation and research into this area. This includes theoretical research as well as examining existing Git encryption implementations and the pros and cons of each in terms of security, functionality, and performance. Lastly, Chapter II discusses software engineering research characterizing version control systems and specific techniques for mining Git data. This is necessary to understand Git usage in order to properly gauge how the secure Git implementation chosen will perform in realistic software development environments.

Cryptography

Cryptography is the art of keeping messages secure, or hidden, from anyone who is not supposed to have access. Cryptography dates back to ancient Egypt in 2000 B.C. [26]. The well-known Caesar Cipher was used in ancient Rome. In this protocol, the letters of the alphabet are all shifted to encrypt a message, and shifted again to their original message in the decryption process. There are four popular goals often used in cryptography [27]:

1. Confidentiality: The ciphertext of a message gives no information about the plaintext of that message.
2. Integrity: The receiver of the message can verify that the message has not been modified in transit and is indeed authentic and not a fake message that an intruder has substituted.
3. Authentication: The receiver of the message can ascertain its origin, preventing an intruder from masquerading and faking identity.
4. Nonrepudiation: A sender cannot deny that they sent a message if they truly sent it.

There are many different protocols in cryptography and the results of using them differ in terms of security and performance. Increasing security tends to increase overhead and decrease performance. Some protocols are tested and assumed secure until a vulnerability is discovered. Oftentimes, the protocol itself is secure, but the implementation in code is not, thus it is always advisable to use vetted cryptography libraries when implementing cryptography in a program [28].

Two of the main types of ciphers are stream and block [26]. Stream ciphers encrypt bits individually by adding (logical XOR in practice) a key stream bit to each plaintext bit. Block ciphers segment the plaintext into equal sized blocks and encrypt these blocks of bits in via a specific algorithm. The two prevalent block encryption algorithms are Data Encryption Standard (DES) and Advanced Encryption Standard (AES). AES is faster than DES and provides options for higher bit levels of security [28].

A simple block encryption cipher is Electronic Code Book (ECB) [26, 27]. This encryption scheme divides up the plaintext into equal size blocks and then encrypts each block, resulting in an equal size ciphertext output. One of the main problems with ECB is that unless a key is changes, a plaintext block will always encrypt to the same ciphertext. This allows attackers to use common occurring plaintext headers or footers, at the beginning or end of messages, and cryptanalysis techniques to decode the message. Alternatively, if an attacker is able to recover matching plaintext and ciphertext, they can match up the ciphertext blocks to other ciphertext messages encrypted with the same key and discover the plaintext corresponding to those ciphertext blocks. Aside from weakness of confidentiality, ECB mode offers no integrity protection. A malicious attacker can rearrange ciphertext blocks and the recipient is still able to decrypt the ciphertext. The plaintext may not make sense, but the recipient is left to wonder if the message has been altered or not. Consequently, more defensive block encryption schemes that overcome these problems are available, but with a performance cost.

One of these schemes is counter (CTR) mode encryption [27, 28]. This mode essentially turns the block cipher into a stream cipher, with a one-time pad encryption,

meaning that the ordering of stream bits added to the plaintext bits to encrypt them is unique and will not be re-used again. The counter can be derived by any function guaranteed to not repeat for as many increments as required by the security settings. In this scheme, an Initialization Vector and counter are used to ensure that each plaintext block that is encrypted differs, even if the plaintext blocks are identical. An Initialization Vector (IV), often called a nonce, is a random value that is input to a block cipher in order to provide randomness to the cipher, in case similar data is encrypted with the same key multiple times. Figures 2 and 3 illustrate the differences between ECB and CTR mode block encryption:

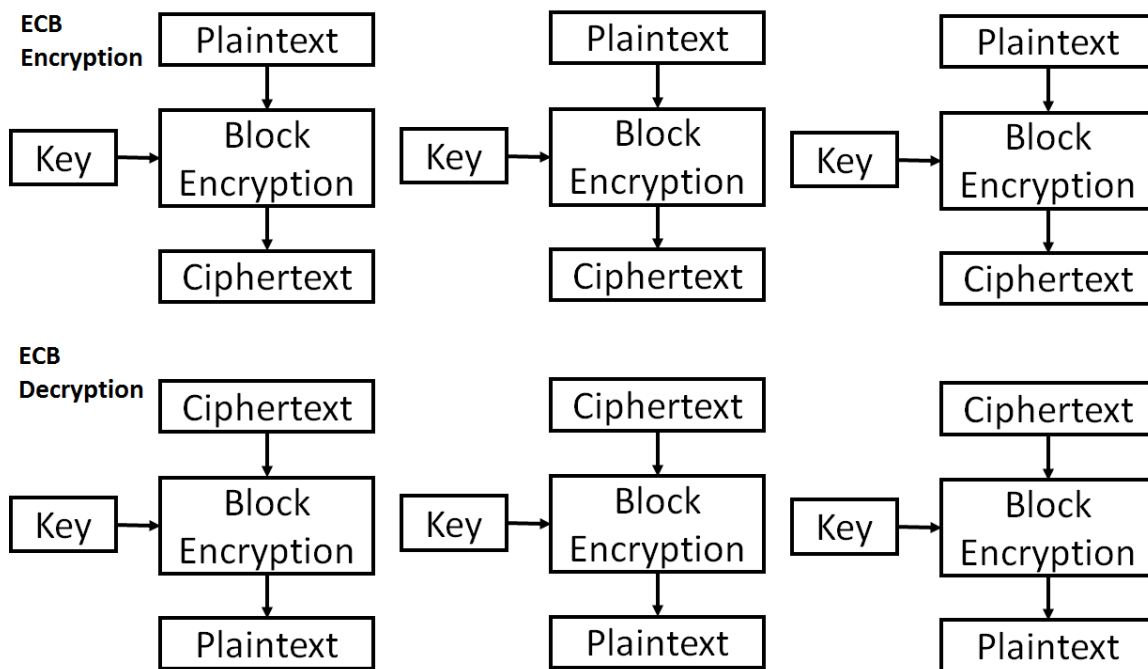


Figure 2. ECB Mode Encryption

ECB mode divides the plaintext up into equal-size blocks and then encrypts them one-by-one into equal size ciphertext blocks. The decryption process is the reverse of the encryption process, using the same key.

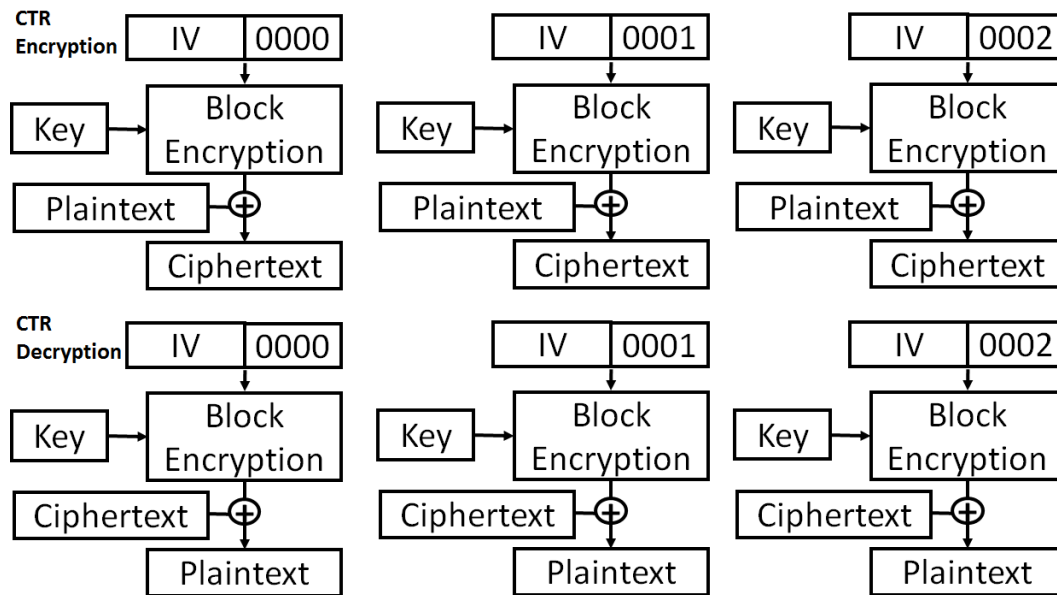


Figure 3. CTR Mode Encryption

CTR mode encrypts an IV and sequence number in order to provide randomness to the plaintext. The result of this encryption then acts as a one-time pad in stream cipher encryption mode, XORing the plaintext with the output of the encryption algorithm in order to calculate the ciphertext. The decryption process is the reverse. Note that in this method, the decryption IV must match the encryption IV. This IV does not need to be secret, but must not repeat, otherwise the one-time pad is not be unique.

Message authentication codes (MACs) and message digests provide integrity protection [26-28]. They assure that any manipulations of a message in transit are

detected. A good message digest is obtained via a strong hash function, a one-way function that inputs an arbitrarily long number of bytes and outputs a fixed length random digest. Strong means that the digest is necessarily random and has a low probability of having a value that collides with another hash function of a different value. MACs take integrity protection a step further by not providing any information about the plaintext. This is because a MAC can only be generated with the proper key, and thus a malicious attacker has no insight into what file generated the MAC. Because the key is used, authentication is proven by use of a MAC. A common way to provide a MAC is to hash the plaintext and encrypt this hash. There are more advanced protocols that combine the previously discussed goals of cryptography into a single block cipher. These are discussed next.

For the purposes of Git encryption, a system that always produced the same ciphertext by using the same key and IV for a given plaintext is desired. Additionally, a scheme that uses AES block cipher is the chosen algorithm based on security robustness and speed. The block cipher offers the option of seamlessly having integrity and confidentiality built into the algorithm [28, 29]. Rogaway and Shrimpton define Synthetic Initialization Vector (SIV) as a scheme that “deterministically turns a key, a header, and a message into a ciphertext.” [29] It takes the name because the Initialization Vector is synthetically created via data within the plaintext. SIV encryption is later defined in RFC 5297 [30] and uses a Pseudo-Random Function (PRF) called String to Vector (S2V) and AES-CTR mode block encryption. SIV encryption builds upon Galois-Counter Mode (GCM), which also provides deterministic authenticated encryption [31]. Operating at

high speeds with low overhead, similarities to SIV encryption exist, as GCM mode is often combined with AES-CTR mode but differs in its hashing. GCM uses the GHASH function, not the S2V function. GCM requires an IV to be input, rather than synthetically generated, which can cause issues in security. This research, however, generates the IV by using a 128-bit secure random number generator, providing 128 bits of randomness to the IV.

GCM is widely accepted and certified by NIST [31]. GCM encrypts using the specific block cipher mode – in this case 128-bit AES CTR mode encryption. GCM outputs ciphertext and an authentication tag that is appended to the ciphertext, enabling the receiver to decrypt and compare the authentication tag generated during decryption to the authentication tag appended to the ciphertext. This allows the receiver to verify that the message was not altered by matching the authentication tags. If they differ, then the integrity of the message is compromised. The Figure 4 shows the high-level block diagram of encryption and decryption process of GCM:

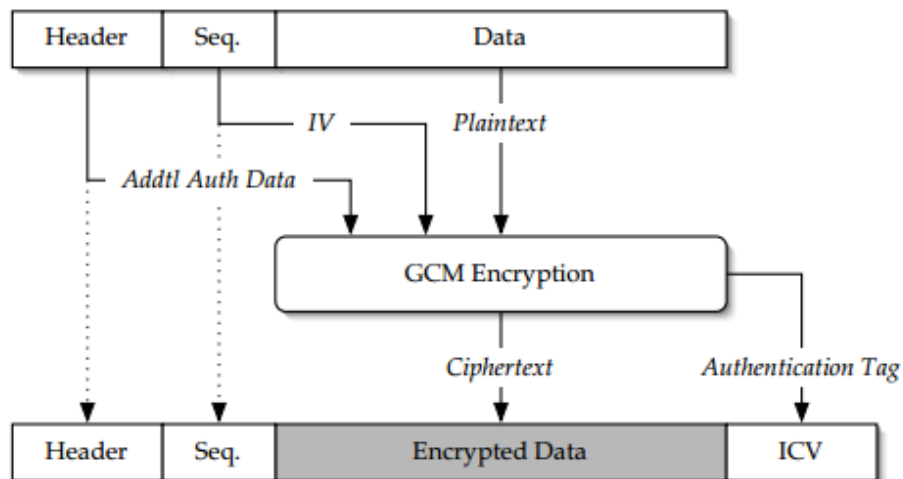


Figure 4. GCM High Level Encryption [31]

The plaintext, a unique IV, and any additional authentication data desired (not used in this research) is passed to the GCM encryption function, where the data is encrypted and the authentication tag is generated. The output of GCM encryption yields ciphertext and appends the authentication tag to the ciphertext. The decryption process is shown in Figure 5:

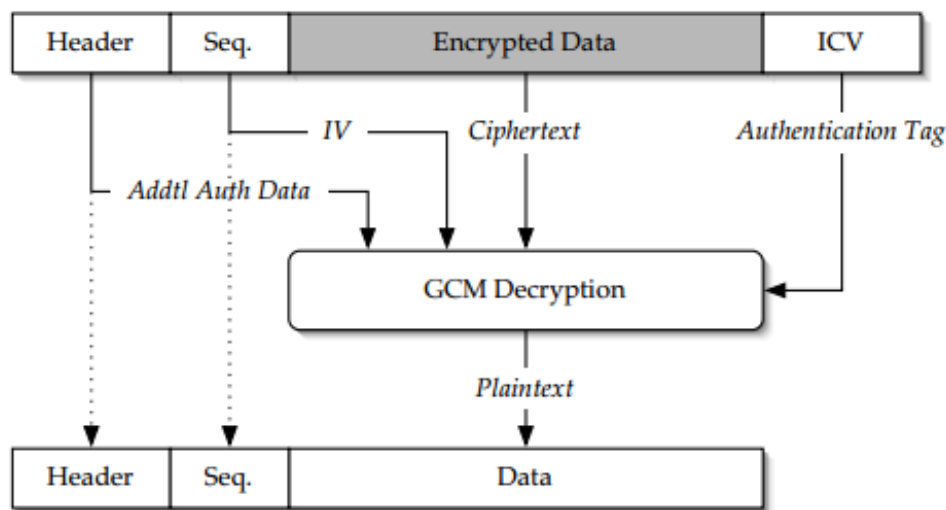


Figure 5. GCM High Level Decryption [31]

The encrypted data and the IV are passed to the GCM decryption function, where the ciphertext is decrypted into plaintext data and the authentication tag used to determine if the data has been altered.

The appropriate security depends on the scenario for usage of the protocols. With Git encryption, confidentiality is the main concern – in that if a third-party were to overtake the cloud provider where the Git repositories are stored, they would not be able to glean any information regarding the plaintext. Integrity protection is a secondary goal,

because the software developers should be able to know if the data has been altered or not. SIV encryption is currently one of the best performers concerning deterministic authenticated encryption, in that the S2V function almost entirely guarantees there will be no nonce reuse at even a higher rate than the NIST approved GCM mode [29]. Even so, there are faster methods of obtaining confidentiality and integrity. Encrypting via AES-CTR mode and then hashing the encrypted data all in a multi-threaded parallel application is faster than both SIV and GCM. In a single-threaded application, however, without taking advantage of parallelization with custom code modification to cryptography libraries, GCM and AES-CTR mode with a secure hash have similar performance.

Version Control

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 are the ability to backup and restore files, synchronization of files, undoing changes, tracking ownership, branching, and merging changes [32]. All of these combined promote efficient software development and support the processes associated with software configuration management.

Some file systems have version control built in, such as Google File System and Bigtable [20, 21]. 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 [32]. Subversion is currently a very popular open source version control system. These are both examples of centralized version control systems [1].

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) [33]. 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.

Addressing scalability and work flow limitations, decentralized version control systems have been developed [4, 33]. These include systems such as Git, Mercurial, Bazaar, and BitKeeper (as stated in the introduction chapter, Git is the most popular of these). These decentralized version control systems 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. This branch is deemed ‘central’ and is stored in an easy to access location and what most developers work off of. Some projects may have several principle branches. Distributed version control systems also provide multiple backups in case of 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 [34]. 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 [1]. 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. Figure 6 shows an example of Git storage:

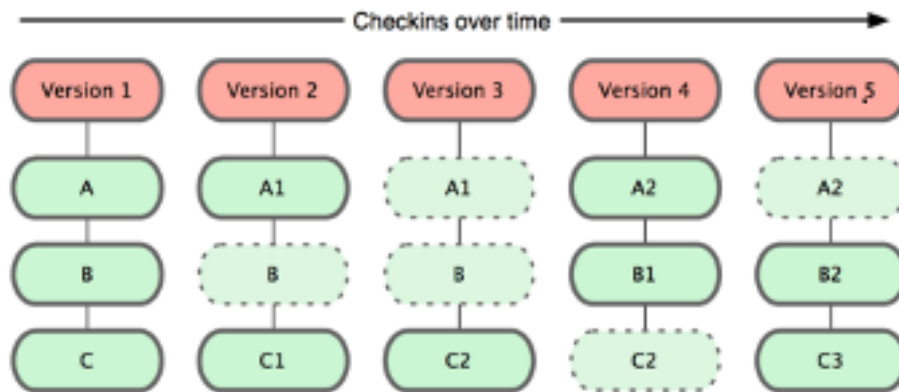


Figure 6, Git Storage [1]

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 arbitrarily long input and a pseudorandom and fixed length output, when checked in, items are stored by hash reference instead of file name [1]. The data object is called a blob and stores the contents of a file. Git also implements tree objects, commit objects, and tag objects [3]. Tree objects serve as a directory, referencing other trees and/or blobs with reference pointers. Commit objects link a physical state of a tree with a description of the commit. Commit descriptions are defined by: the name of the tree object, parent node(s) representing the

previous steps in the history of the project, an author who wrote the change, a committer who took the commit action, and a comment description of the commit. Lastly, the tag object is not often used and is a method in which an individual may ‘tag’ an object with a message. An example of this is to provide their signature as a tag. The objects can be easier understood visually. Figure 7 shows the visual organization of the three main Git objects (commit, tree, and blob), their internal structure, and how they reference each other (note that the data, such as the hash reference, is abbreviated).

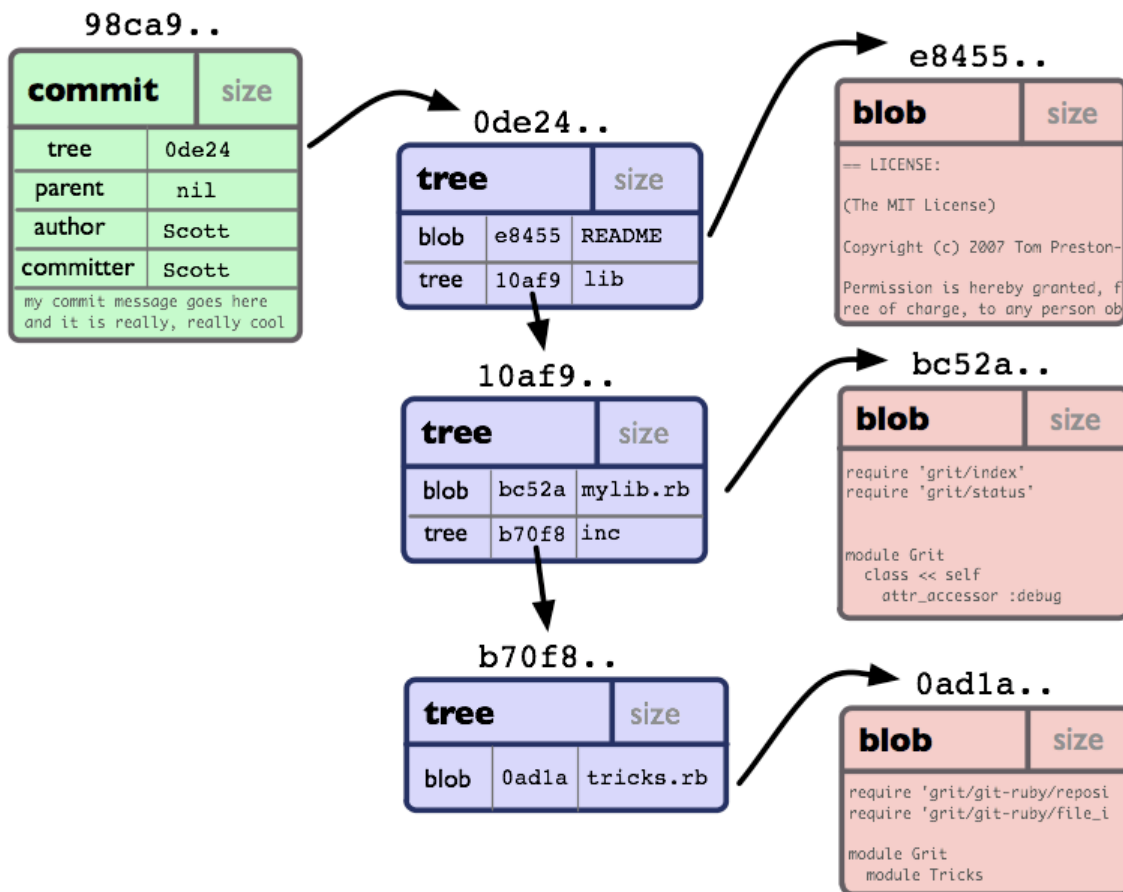


Figure 7. Git Objects [3]

Git compresses the contents of files with zlib, a software library for data compression [1, 3]. This makes the files by default compressed within the blob. Since Git stores snapshots of each file that changed, however, multiple copies of the same file increase the size of the repository linearly. It turns out that while the references are to snapshots containing entire files, Git is able to store only the deltas between the objects. The initial form of object storage is called loose object format. Git occasionally packs several of these objects into a single binary file, called a packfile. The blobs by default are not compressed. Git internals run a routine maintenance on the objects. This maintenance is automatically run when Git deems it has too many loose objects around, or the garbage collection command can be run within Git to pack the files. Additionally, when objects are pushed to a repository, garbage collection is run to compress the objects before the push is executed. Git's garbage collection uses a set of heuristics to find similar objects and the deltas between them. Garbage collection then compresses the repository by storing the object and the deltas between different versions. This is stored as data in the packfile. The packfile is in a binary format and has a main header, as well as headers for each individual object, which describe the size and type of objects. Garbage collection also creates an index file that contains the offsets for each object. The index file references the packfile so that the deltas can be unrolled between different versions for examination. The packfile is critical to the efficient storage of Git. The index file and packfile formats are shown in Figure 8:

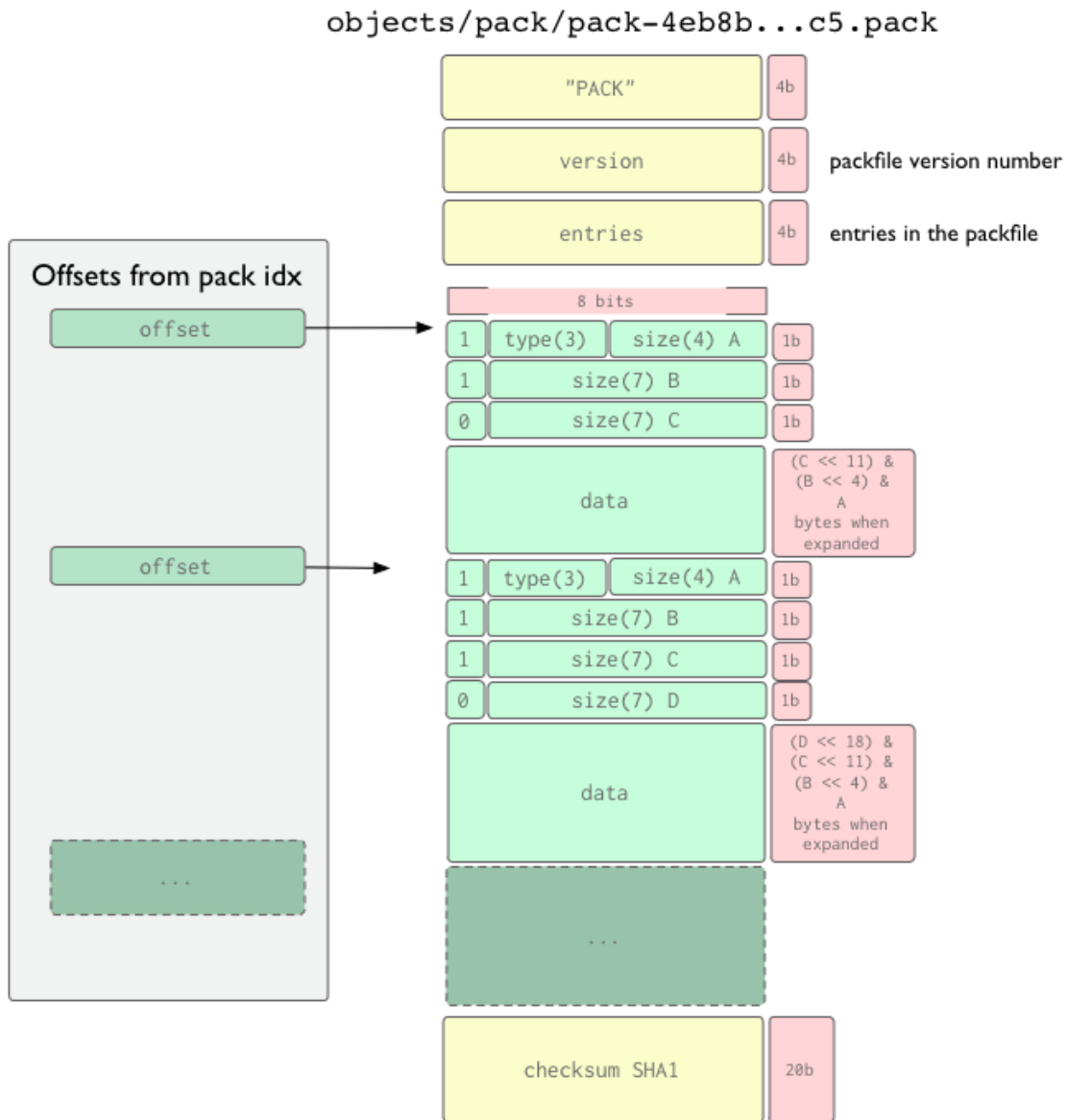


Figure 8. Git Packfile and Index Example [3]

Git is by default set up to handle ASCII files, however, filter functions are built into Git [1, 3]. These are the smudge and clean filters and they allow data to be modified as it travels to and from the repository. By editing the .gitattributes file, one can configure certain types of files to pass through the filters, which are scripted code or programs that

are referenced in the Git configuration file. Many Git filters currently exist, such as for binary files or word documents, along with the functionality to create custom filters. Once the filter is chosen, it must be linked to git config settings. The smudge filter is run on Git checkout, shown in Figure 9:

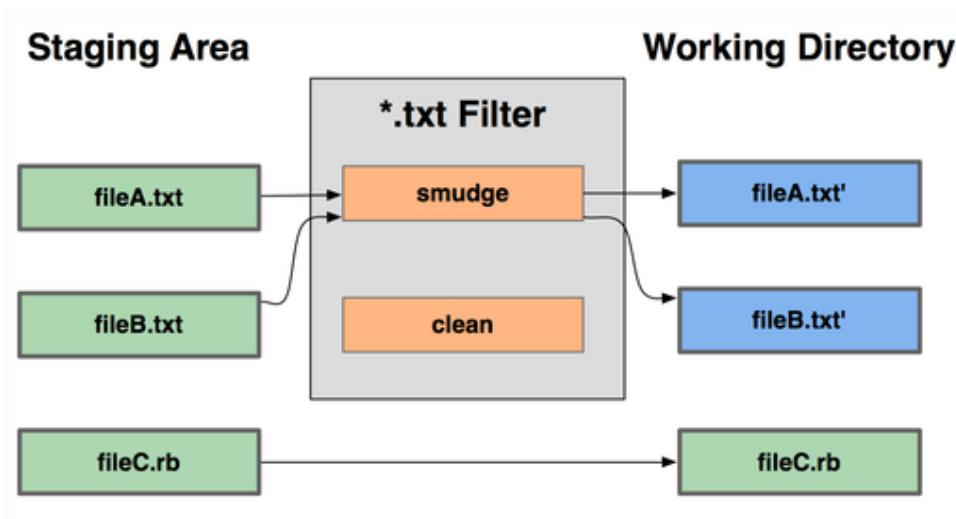


Figure 9. Git Smudge Filter [1]

The clean filter is run on Git add, shown in Figure 10:

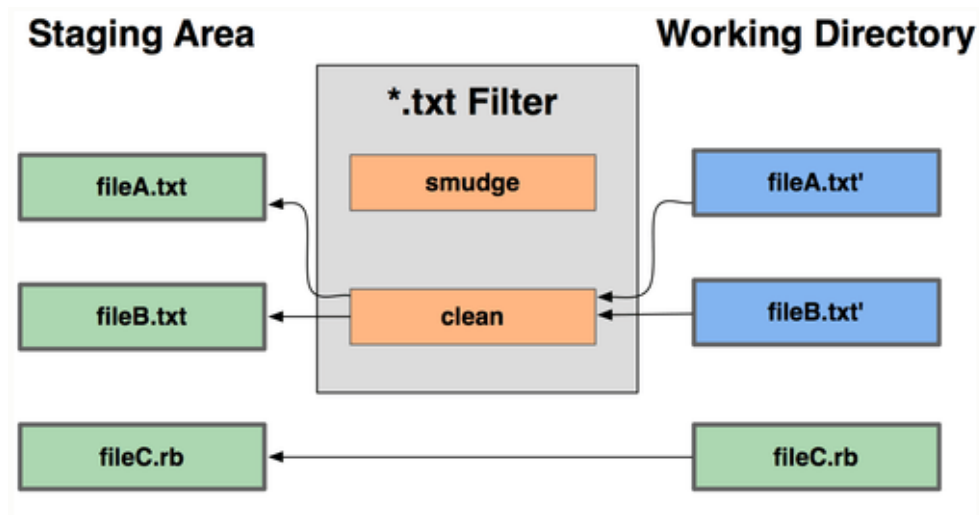


Figure 10. Git Clean Filter [1]

Research Background

Although little published literature is found concerning a secure Git implementation, there is a plethora of published research on securing the public cloud [15, 35-37], deterministic authenticated cryptography [29, 30], other version control systems [32-34], and work which uses Git as a file-system [38, 39]. There are some papers regarding the need for transparent Git encryption and another on how it should be done [40, 41]. Robinson goes further to implement GitBAC: Git-Based Access Control in his paper, “GitBAC: Flexible Access Control for Non-Modular Concerns” [42]. This paper is an ideal stepping stone for this research. The referenced paper has similar goals but does not fall in line with the functionality, infrastructure, and security goals of this research. The paper describes GitBAC as a proxy between the user and the Git-Server, controlling access based on an access control list. In this setup, both the proxy and the Git server have to exist on protected and trusted resources, as the Git repository is not encrypted. This research has the goal of storing the Git repository on any cloud computing environment in order to take advantage of the economical and computing performance of cloud computing [13]. A diagram of GitBAC is shown in Figure 11:

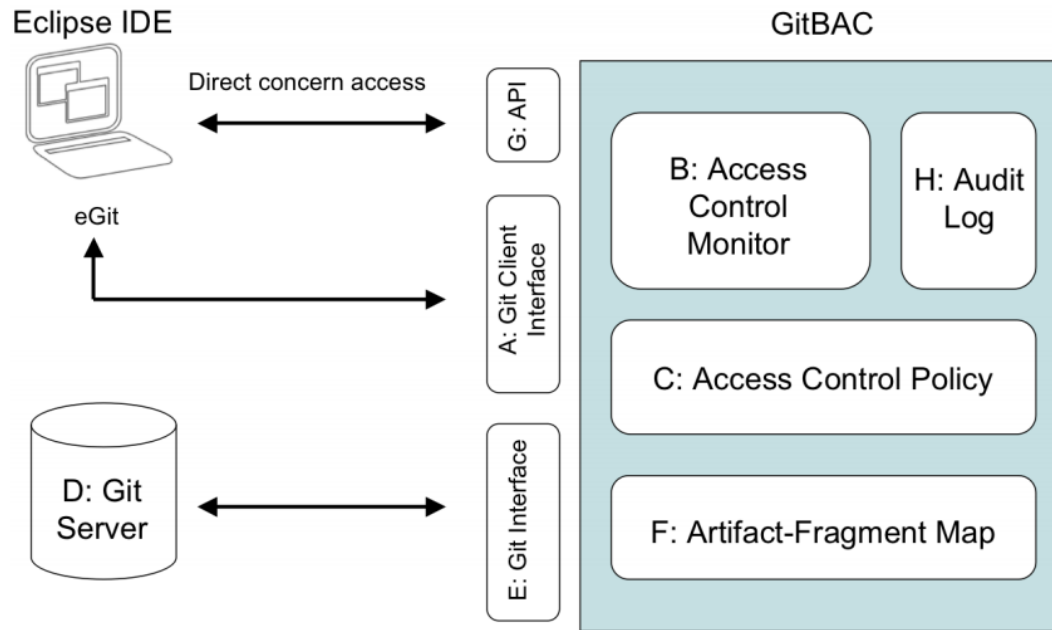


Figure 11. GitBAC Diagram [42]

There are three different open source Git-encryption schemes that can easily be found through web search: Gcrypt, Git-encrypt, and Git-crypt [43-45]. These implementations appear to be developed by individual hobbyists and lack much information or detail with regard to verification and validation testing. Gcrypt uses GNU Privacy Guard (GPG) to encrypt a remote repository and access it. It is listed as a development version, and was omitted from thorough testing for the reason that it is still a work in progress. The other two, Git-encrypt and Git-crypt are both complete projects and are evaluated in this research. Both use Git's smudge and clean filters to allow the custom code to intercept and edit data as it interfaces with the repository. The two implementations encrypt the data when adding it to the repository and decrypt the data as it is checked out of the repository.

Git-encrypt is a series of scripts written by Woody Gilk [43]. Git-crypt is a C++ library written by Andrew Ayer [44]. Due to the nature of command line scripts, Git-encrypt is much slower than Git-crypt's implementation. Git-encrypt calls OpenSSL cryptography library 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, as discussed earlier in this chapter. This type of encryption is chosen to examine a range of encryption techniques, as the Git-crypt implementation is more secure. However, Git-crypt test results show that it does not allow for the same repository size compression as Git-encrypt. Referring to Git-encrypt, a user can change the cipher mode to any OpenSSL scheme, such as cipher block chaining. One problem with alternative block modes is that there is no flexibility for selection of IVs. The implemented IV is not random, which results in insecure encryption for all schemes requiring a random initialization vector. Additionally, there is no MAC implementation to verify the integrity of the message.

Lastly, the user has the option of picking a password and a salt [43], a random value that is appended to a password. The added functionality of a salt provides more security to the password. If the user chooses, the password or salt can be set to random. If the user inputs both of these variables, then there is no check to ensure that they are indistinguishable from uniformly random. A user could choose two words found in a dictionary and thus an attacker would have an advantage in breaking the combination by brute force techniques of common word combinations in an attack. If the user decides to select the random choice implemented in Git-encrypt, the code calls the Linux

/dev/random function. As a robust function, Linux /dev/random is proven to not be indistinguishable from uniformly random, and not secure [46].

Git-crypt is faster than Git-encrypt, since it is written in C++ [44]. It also provides more cryptography functionality, but without cryptography settings options. The algorithm uses AES-256 counter mode with an IV derived from SHA-1-HMAC hash (this is nearly identical to the GCM mode discussed earlier, with subtle differences). This scheme of hashing the message and then encrypting it provides a random IV 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 IV, then deterministic authenticated encryption is achieved. Recall from the beginning of this chapter that this means that the same plaintext will yield the same ciphertext since the IV is derived from the plaintext via a set function rather than an incrementing counter or timestamp [35]. Recall from the cryptography section that this encryption uses two separate keys, one for the hash and one for the AES-CTR encryption. Integrity is provided as the decrypted message is passed to the hash function to check the output IV versus what was received. If the output IV and the received IV match, it is determined the message was sent from someone with a correct key and has not been altered [28].

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 [28]. Deriving the IV from the hash of the message also determines (with very high probability) the same IV will not be reused to encrypt a different message.

In Git-crypt, the key is derived using the `RAND_bytes_openssl` function [44]. 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.

There have been complaints about using filters to encrypt Git on the basis that it takes away from the lightweight and efficient design [47]. 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 the decision must be made to make sure the benefits outweigh the costs. Having described the security aspects of these two open source implementations, this thesis examines performance compared to unencrypted Git and also introduces a new implementation, named Git Virtual Vault, that improves upon them with regards to performance. The results are documented and provided so that any user has all of the information necessary to make their own decision.

In the subsequent chapters discussing methodology and results, the analysis shows that the time it takes to use the clean and smudge filters in order to encrypt and decrypt data is not substantial enough to turn away potential users. The size increase of the repository, however, will be enough to turn away potential users in some cases. In the case of a very small change in the plaintext, the Git-crypt repository grows linearly. This is because the change in the plaintext alters the IV, which in turn alters all of the resulting ciphertext blocks. The result is that a very small change in the plaintext results in a copy of the whole repository being stored. Since very little of the ciphertext is identical, the robust garbage collection within Git is unable to compress efficiently, resulting in a large

repository size. This is not the case with the testing of Git-encrypt, since the implementation tested uses ECB block cipher encryption. This encryption only changes the ciphertext blocks corresponding to the changed plaintext blocks. But, as discussed earlier, this is not a secure method of operation.

The inefficiency of compression using Git-crypt brings further research into whether or not the IV must always be unique, or if the IV could be based on the hash of the filename, or something more constant, for example. While this would indeed introduce the same compression qualities of ECB mode, it would destroy the security of the protocol. Recall from the cryptography section earlier in this chapter that the resulting IV is used as input to the block cipher encryption function, which has the output xor'd with the data. Thus this portion acts as a stream cipher. Since it is a simple xor, using the same 'key' to xor twice results in what is known as a two-time pad, which is insecure [26, 27]. Counter mode in equation form is essentially:

$$C = P \text{ xor } CTR(key, IV)$$

CTR is the AES counter mode with input of the symmetrical key and IV. If two different plaintexts, say P1 and P2, are encrypted with the say Key and IV pair of values,

then the result is:

$$C1 = P1 \text{ xor } CTR(key, IV)$$

$$C2 = P2 \text{ xor } CTR(key, IV)$$

Since any value xor'd with itself will cancel out, the CTR(key,IV) output will cancel out, thus allowing that:

$$C1 \text{ xor } C2 = (P1 \text{ xor } CTR(key, IV)) \text{ xor } (P2 \text{ xor } CTR(key, IV))$$

After cancelling the $\text{CTR}(\text{key}, \text{IV})$ identical function, we get:

$$C1 \text{ xor } C2 = P1 \text{ xor } P2$$

Now the attacker can use common techniques of frequency of English characters, or headers or footers for code to guess the proper plaintext and crack the whole repository.

There is no valid way to securely use counter mode with HMAC, as is done in Git-crypt, with a reuse of the IV [44]. This is the shortfall that is examined and attempted to overcome in the new Git encryption implementation. Git filters are very useful for modification of individual files as they are checked out from a repository or added to a repository [1, 3]. Git filters, however, modify the data one file at a time, before the data is added to the repository as an object. With this order, the filters operate on the data before it can be compressed with the rest of the data in an object. If there is a way to add the data to the repository and then encrypt objects, this would help solve the problem of the size growth of the repository. Since Git is open source, there are a few options to modify it in this manner. One is to analyze the code of Git and modify it. Another, and more viable option, is to engage with more expansive libraries for Git, to use the benefits of Git but also take advantage of needed features not found within.

Two external Git libraries are JGit and libgit2 [48, 49]. These two libraries provide an Application Programming Interface (API) for various languages to directly work with Git repositories and functions. They essentially provide a back-end Git infrastructure, allowing for the developer to create a new front end, fitted to the purpose of their application.

JGit is a lightweight pure Java library implementing Git [48]. It is open source, developed by Shawn Pearce, and licensed by Eclipse, a popular Integrated Development Environment (IDE). JGit has very few dependencies, making it suitable for embedding in Java applications. JGit was originally created with the goal of providing an Eclipse plugin for working with Git but grew to expand Git functionality and offer many additional tools for Git repositories. There are several developers working on JGit, with new functionality being added often. JGit has the advantage of full integration of Git, and additional functionality beyond traditional Git, within a Java object oriented design [50].

Libgit2 is a pure C implementation of Git core commands [49]. It allows developers to write custom Git applications in any language supporting these C bindings. It is also an open source project like JGit and provides similar functionality. The full libraries of commands are available via the C++ implementation, but libgit2 also has less inclusive language bindings developed for languages such as Ruby, Python, Perl, PHP, and C#, to name a few.

Both libraries are analyzed in detail and some of the main developers contacted via e-mail. The decision of this research is to use JGit for Git modification because of the portable nature of Java, sponsorship by Eclipse, more thorough documentation, and more commands and functionality, when compared to libgit2. JGit allows one to implement Java encryption libraries at the transport level, either locally or remotely. This allows objects to be encrypted as they are pushed to or pulled from a remote or local repository. This functionality allows the new Git encryption implementation, named Git Virtual Vault, to take advantage of all of the efficient and lightweight features of Git, including

the garbage collection function, and still maintain a secure repository on an unsecure cloud.

The final item related to version control that is researched in this thesis is the topic of how distributed version control systems, particularly Git, are used today by various entities. This research came about because it is important to know who the general audience of a product will use it, so that usage habits can be learned and analyzed in order to provide a product that meets the average well-rounded user's needs, with testing that a user can relate to.

There are several general studies regarding the natures of commits. One in particular in which authors Hattori and Lanza seek to quantify the size of small and large commits within version control systems [51]. Alali, Kagdi, and Maletic [52] dig into what a typical commit represents, in trying to characterize open source software repositories. In their research, the authors study nine open source software systems to uncover characteristics of how developers use commit commands in version control systems. They find that roughly 75% of commits are small and the messages that are included with the commits can be correlated to the size categories of the commits. In addition to the size of commits, Kolassa, Riehle, and Salim [53] explore the commit frequency distribution of open source projects in order to further understand the software development process. Other research is analyzed dealing with characterization of version control systems, amassing version control system repositories into a census type study, and also software engineering practices of open source projects [23, 54-56]. These are all

interesting software engineering studies, but the research is found to be too high level for the goals of this thesis.

For details regarding developing a Git modification application for an audience and how consumers of that application are going to use it, specific random and targeted data is needed about the actual number of files and also size of files of each commit. Also, language specifics are desired. Gousios and Spinellis [57] explore GitHub, a popular project hosting, mirroring, and collaboration platform for projects using Git. In their research, they explore the GitHub extensive REST API, which enables researchers to gather specific information from the public repositories hosted on GitHub. This research does not go to the low-level desired in this current thesis, but provides a look into a tool that is usable.

Further research finds that there are websites dedicated to mining GitHub data. This is beneficial because these sites provide up to date data and as outlined in the introductory chapter, version control system usage is rapidly changing. One such site is GitHub Archive [58]. This site began in 2012 and archives GitHub repository events, such as pushes and pulls from the repository. Each archive contains a JSON encoded stream of these GitHub events, allowing processing in any language. The dataset is also available via Google's BigQuery, and can be accessed online via queries over the dataset in a matter of seconds. While this archive does not encompass the entire data history of GitHub, it does provide a very fast and user-friendly way of mining GitHub data without the query frequency restriction limits imposed by GitHub. A combination of data mining using this GitHub archive and also the GitHub API [59] provides the tools necessary for

characterization of Git usage. This characterization comes in terms of commit habits, push habits, repository size, and language differences.

Summary

Git is a popular distributed version control system. As computing moves from local and secure to offsite and cloud computing centers, security concerns arise. This chapter explored security to provide a basis for understanding why a specific protocol is chosen. It went on to provide a more detailed look at the origins of version control, and the functionality and internal structure of Git, so that the reader has the knowledge necessary to understand the Git encryption implementation. The chapter then discussed various Git encryption implementations and concluded with software engineering research characterizing usage of Git. Chapter III discusses the methodology necessary for analyzing and demonstrating the proposed secure Git implementation solution.

III. Methodology

Chapter Overview

The previous chapter gave background information relevant to this research and then characterized the security of different existing Git encryption implementations, based on the cryptography protocols used. The goal of this thesis is to find or develop a viable secure Git implementation that can be used while working with sensitive data in unsecure environments. If successful, this will allow organizations that work with sensitive data to take advantage of today's efficient cloud computing environments and use Git in a secure distributed manner, without degrading performance [13, 25]. This research methodology describes the tests to analyze existing Git encryption implementations by comparing their performance relative to unencrypted Git, as well as to each other. Proper methodology proves the hypothesis that encryption of a Git repository is possible without overly degrading the performance of Git and losing the functionality of Git.

As discussed in Chapter I, distributed version control systems are rapidly gaining popularity among software developers and Git has become the most popular distributed version control system [6]. Git was created for open-source, non-secure environments, but its usability and performance is desired by those who work with sensitive data. There is no formal research or methodology characterizing or analyzing any Git encryption methods. This research fills a large hole in answering the question of whether Git repositories can be encrypted to be used in a secure manner for sensitive projects. The hypothesis of this research is that this is possible. This research contends that Git can be

used with encryption while still maintaining relevant performance and functionality that enables it to be efficient for real-world projects.

In order to show the validity of this hypothesis, a three-phased approach test methodology is used. Phase One consists of testing existing Git encryption implementations and base lining their performance through analysis and characterization. This phase is called the Secure Git Baseline Phase. The analysis of this phase is accomplished through a series of three broad, worst-case type scenario, controlled tests. Each test measures a unique performance metric. Two existing Git encryption implementations are tested and analyzed: Git-encrypt, a series of scripts written by Woody Gilk [43], and Git-crypt, a C++ program written by Andrew Ayer [44]. They are analyzed and compared to unencrypted Git in terms of functionality and performance.

Phase Two consists of characterizing Git usage habits in order to provide realistic and accurate information to make the decision as to when a secure Git implementation becomes unusable for practical purposes, from a performance standpoint. Having this information allows for determining that Git encryption works under realistic scenarios. This phase is referred to as the Git Characterization Phase. As discussed in Chapter II, high-level overview research has been performed in the area of characterizing version control systems [52-57], but this research does not provide enough detail to fully understand the intricacies of how software developers use Git. More detail is needed to emulate Git usage in detailed terms of commit frequency, commit size, how many files are changed per commit, how many lines in each file are changed, and the variance by languages. This is new research and contribution to the software engineering community.

With this information, a proper Git usage emulator can be created in order to simulate Git usage for a repository over a particular amount of time. Having an accurate simulation provides higher confidence that Git encryption can be used for a variety of different projects, as different settings can be tested and results analyzed. In order to retrieve real-world Git repository data, GitHub repositories are mined via the GitHub Archive [58] and the GitHub API [59].

Phase Three consists of developing an improved secure Git implementation that addresses the shortfalls of the previous Git encryption methods from Phase One. This phase is referred to as the Secure Git Improvement Phase. This new implementation was briefly described in Chapter II and retains the rigorous security that the cryptography protocols used in Git-crypt provide but overcomes some of the performance shortfalls. This new secure Git implementation is aptly named the Git Virtual Vault (or GV2, pronounced G-V-squared). There are many types of vaults, but in general a vault is a secure container in which items of value are stored. The vault can only be opened by using the proper key, which is only given to those with controlled access to the vault. This new secure Git implementation is a virtual vault, meaning it is a secure and enclosed virtual location, such as used in cloud computing. The items of value stored in this virtual vault are sensitive Git source code files. These items are only valuable if they are obtained in plaintext. Thus, the vault is made secure by encrypting all of these sensitive files and providing integrity protection so that those who do have access to the files know if someone has been altering them. From this discussion, this new secure implementation

is aptly named GV2. GV2 is the final product of this thesis research and demonstrates a valid secure Git implementation.

Phase One: Secure Git Baseline Phase

The initial phase of this thesis methodology consists of testing existing Git-encryption implementations and obtaining a baseline of their performance through analysis and characterization. The testing analyzes performance 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 unencrypted Git and two existing Git encryption implementations: Git-encrypt [43] and Git-crypt [44].

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 is found by looking through the popular repositories on GitHub, an online storage and sharing area for Git repositories [60]. It is developed by Schneiderandre and is a collection of animation examples of Apple iOS applications. The size characteristics of each are shown in the Table 1:

Table 1. Phase One 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

Phase One, Experiment I: 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.

Phase One, Experiment II: 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.

Phase One, Experiment III: 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 versus waiting until the last iteration as with the Linux Kernel test. Differences in the compression of the data based upon 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.

These three initial tests baseline the performance of existing Git encryption implementations in terms of how long simple commands take to run, the size increase in a repository when the contents are encrypted, and also the size growth increase in a repository when files within the repository are edited. Additionally, this test examines the garbage collection routine of Git, which is very important to performance. The results of these tests are fully examined in the following chapter. The initial premise before testing is that the performance will see a large increase in both time and size, as encryption takes time and it is hard to compress encrypted data. These both turn out to be accurate assumptions and the large size increase of the repositories, described in the experimental results section, show the need for the improved Git encryption implementation.

Phase Two: Git Characterization Phase

The second phase of this methodology consists of characterizing Git usage habits from a software engineering perspective. This is done to provide realistic and accurate information used to show that Git encryption works under realistic real-world scenarios. This phase is aptly named the Git Characterization Phase and researches information necessary to emulate Git usage in terms of commit frequency, commit size, how many files are changed per commit, how many lines in each file are changed, and the variance by programming languages. The process of mining Git data consists of using two venues:

GitHub is mined via the GitHub Archive [39] for top level information and also using the GitHub API [40] for lower level details.

GitHub Archive can be accessed multiple ways. In this mining session, Google BigQuery tool [61] is used to mine GitHub with simple database queries. BigQuery is an analytical tool allowing for interactive analysis of large datasets via BigQuery's SQL syntax.

For mining purposes, C++, Java, Javascript, PHP, and Python are chosen as they have varying syntax programming paradigms and represent some of the more popular languages both presently and over the past decade. The first experiment with this data is mining a set of repositories of each language and comparing push data to distinguish activity differences in the repositories. The entire base of repositories does not need to be analyzed in order to draw a conclusion. The API for BigQuery contains a hash function which is used in this script to select random repositories. The script consists of selecting repositories according to language, then ensuring they are recent and have been created within the last two years (for relevancy purposes), then randomly selecting a repository from the results, then ensuring the repository includes data and is not an empty initialized repository. Lastly, the repositories must be somewhat active (not a one-time creation and push, etc.). In order to ensure this, a minimum number of pushes of 30 is chosen. This equates to roughly two updates a month under the two year recent repository selection constraint. In a project needing security, any less than this would allow a system with inefficiencies to be sufficient and the efficiency targets pursued in this study would be

unnecessary. An example of the BigQuery environment and Java language query is shown in Figure 12:

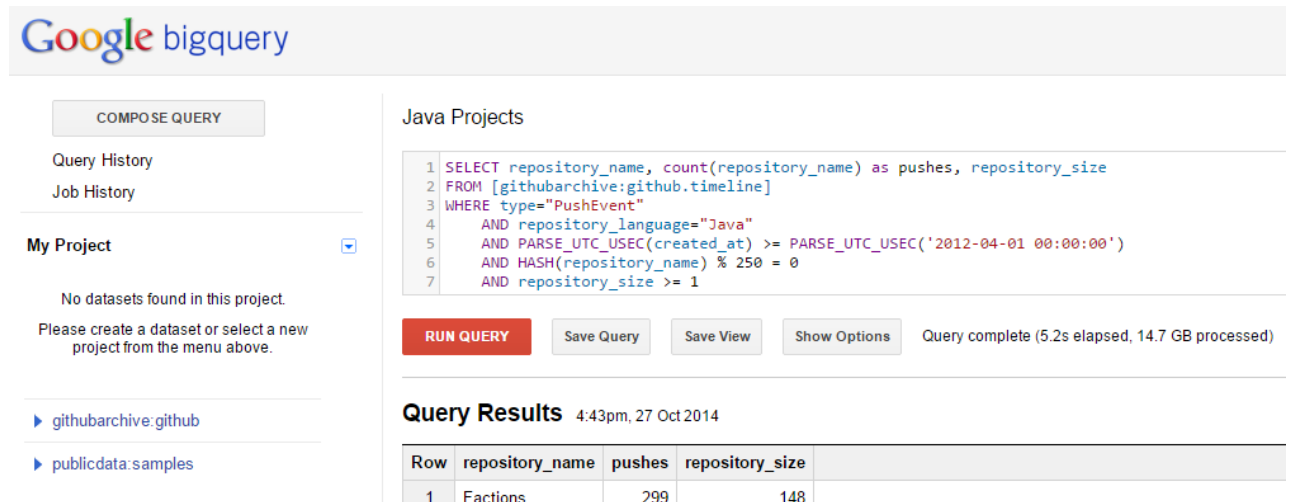
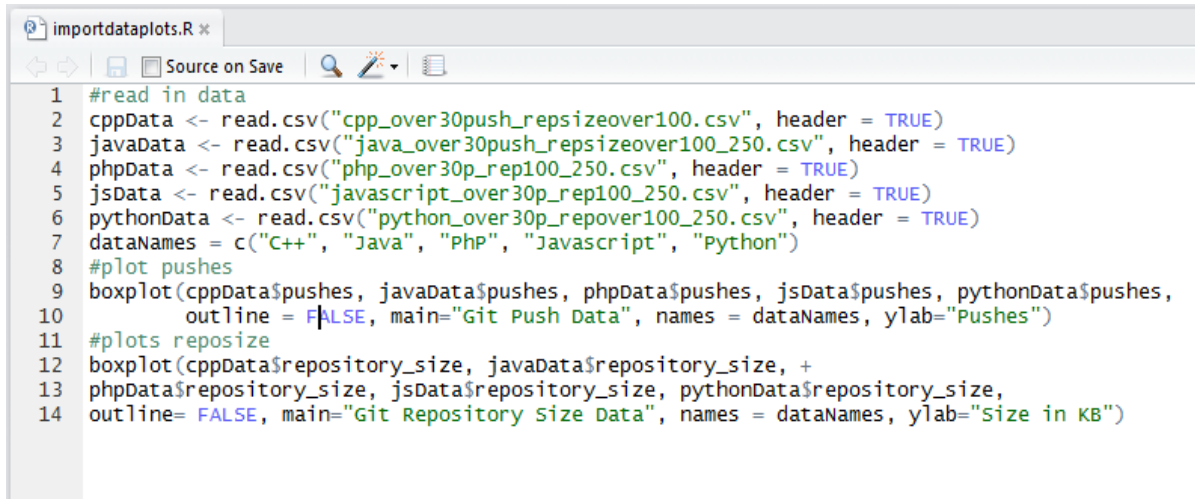


Figure 12. BigQuery Java Query

From the example query above, one can see the easy to use interface. The results are shown in the web browser and then downloaded to a .csv file. Next the results of the five queries are imported into the RStudio integrated development environment and analyzed. The push data is first analyzed and plotted to determine frequency of usage differences in the languages. Then the sizes of the repositories are compared to determine if language makes a large difference in terms of size. The R Script run in RStudio is shown in Figure 13:

The image shows a screenshot of the RStudio interface. The top pane displays the source editor with R code. The code reads five CSV files into data frames: cppData, javaData, phpData, jsData, and pythonData. It then creates two boxplots: one for 'Pushes' and one for 'Repository Size' in KB, comparing the five languages. The bottom pane is currently empty.

```
1 #read in data
2 cppData <- read.csv("cpp_over30push_repsizeover100.csv", header = TRUE)
3 javaData <- read.csv("java_over30push_repsizeover100_250.csv", header = TRUE)
4 phpData <- read.csv("php_over30p_rep100_250.csv", header = TRUE)
5 jsData <- read.csv("javascript_over30p_rep100_250.csv", header = TRUE)
6 pythonData <- read.csv("python_over30p_repsizeover100_250.csv", header = TRUE)
7 dataNames = c("C++", "Java", "PHP", "Javascript", "Python")
8 #plot pushes
9 boxplot(cppData$pushes, javaData$pushes, phpData$pushes, jsData$pushes, pythonData$pushes,
10         outline = FALSE, main="Git Push Data", names = dataNames, ylab="Pushes")
11 #plots reposize
12 boxplot(cppData$repository_size, javaData$repository_size, +
13         phpData$repository_size, jsData$repository_size, pythonData$repository_size,
14         outline= FALSE, main="Git Repository Size Data", names = dataNames, ylab="Size in KB")
```

Figure 13. RStudio repository push and size data

The results of the push data collection and analysis are shown in Figure 14 (note that the thick line on each bar represents the mean of the data for each language and the box encompassing each language field represents the 95% confidence interval for each language). The push data collection from the five different languages is very similar, as the 95% confidence intervals overlap. The average total pushes for each repository over the past year is around 40.

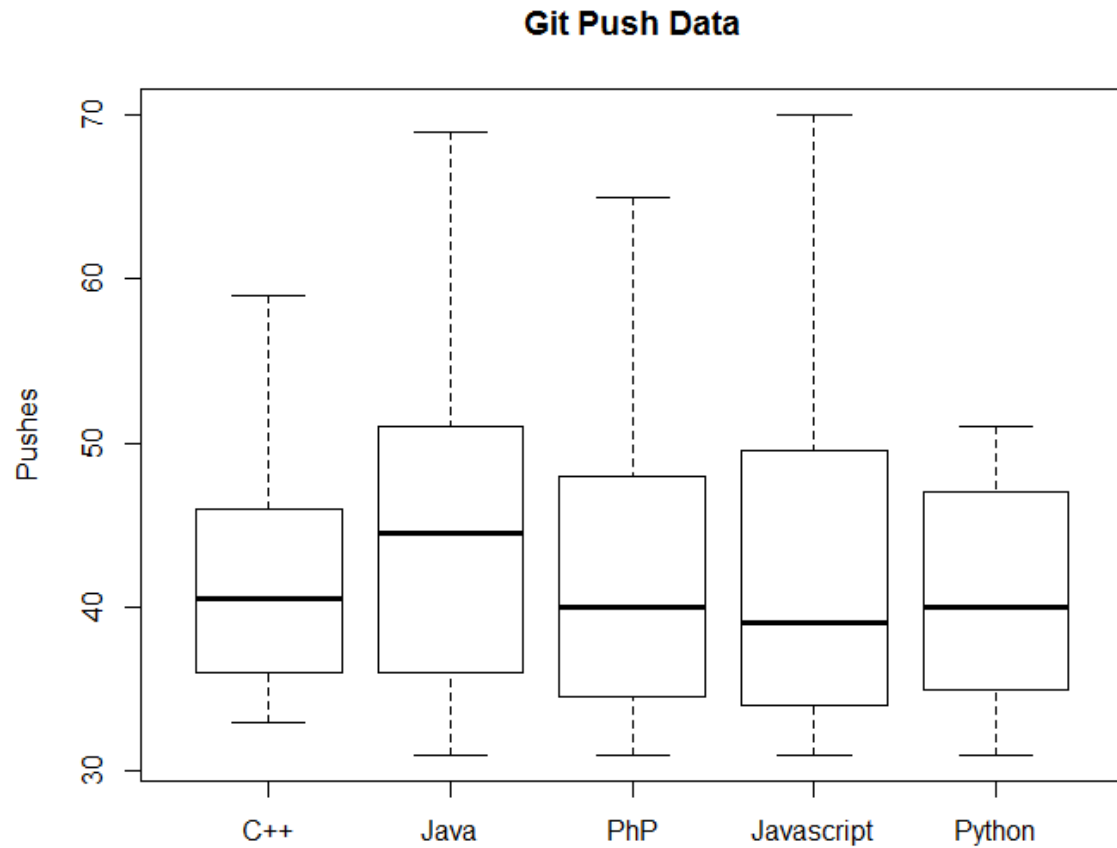


Figure 14. Git Push Data

The repository size data, shown in Figure 15, is much different than the repository push data. The size of the repositories that contain Java, PhP, Javascript, and Python source code are all relatively the same, but C++ language repositories are nearly twice the average size of the others and the 95% confidence interval extends much higher in size quantity:

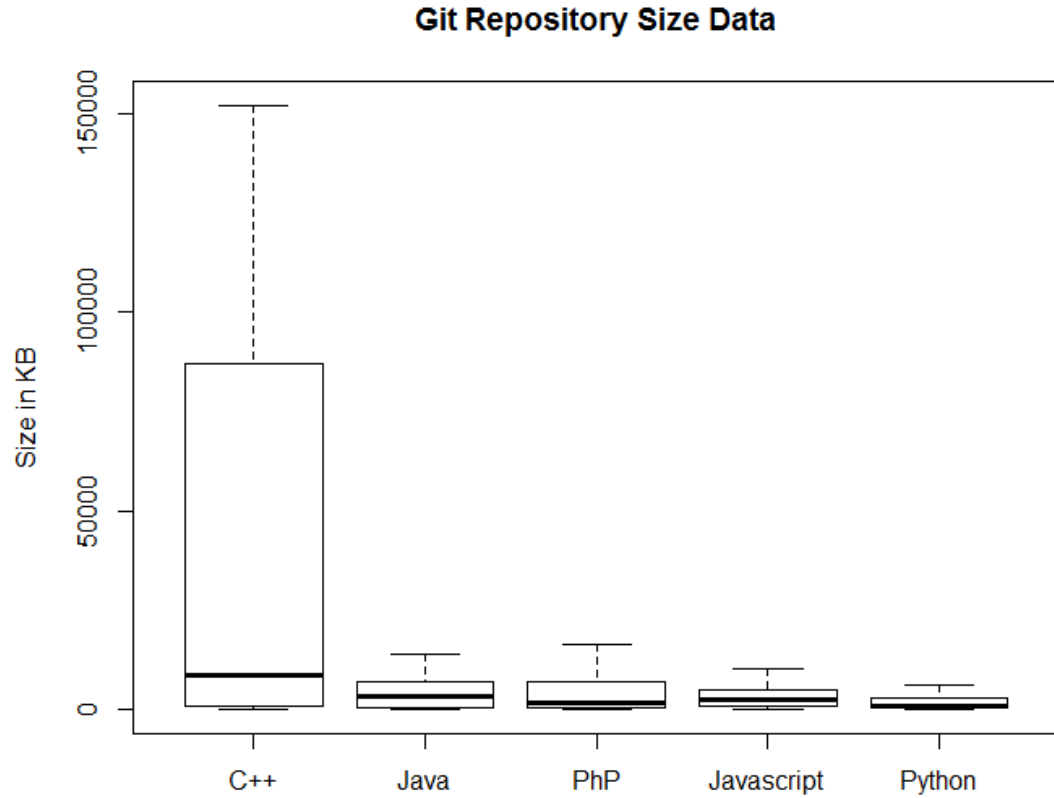


Figure 15. Git Repository Size Data

After these two high level characteristics of the languages are analyzed, a subset of random repositories is chosen in order to test at a more detailed level using the GitHub API. 15 samples are chosen: 3 from each of the 5 chosen languages and spanning 3 different repository size categories of small, medium, and large. Small is defined as less than 10 MB, Medium is between 10-100 MB and large is greater than 100 MB in size. The resulting repositories chosen are shown in order of small to large in Table 2. The name, language, size in KB, and number of pushes in last two years is shown:

Table 2. GitHub Random Repository Selection

Name	Language:	Size (KB):	Pushes:
Devnull	PHP	4028	51
ComputerNetworks	Python	133	51
Restfiddle	Java	1372	38
Intouch2	JavaScript	1992	54
CoolProp	C++	9926	51
EZ Publish	PHP	38201	60
1PICNIC	Python	30856	51
TetraWord	Java	14023	50
Syra	JavaScript	15792	42
FF2-Alpha	C++	29921	35
Openvault	PHP	125569	32
Tendenci	Python	168627	49
Eucalyptus	Java	145201	46
Boost32Boost	JavaScript	122692	2
Nme	C++	223925	38

Additionally, 15 repositories from GitHub's popular trending monthly category are also chosen as examples of popular repositories. Many of these popular repositories are hosted by well-known companies, such as Google or Facebook, and represent enterprise-like software, or at least software that is being worked on by many developers due to the popularity, rather than the random selection that could be enterprise or a solo

hobbyist. These selections are limited to the list that GitHub provides and not every size value for each language is represented. The frequency of commits in these repositories are higher as they are chosen from the popular repositories. Testing these determines if any differences between these and the random selection exists in terms of commit size.

The selection is shown in Table 3:

Table 3. GitHub Monthly Trending Repository Selection

Name	Language:	Size (KB):	Sponsor:
Typecho	PHP	12212	Typecho
PHPMailer	PHP	4886	PHPMailer
Google-api-php-client	PHP	8586	Google
Physical-web	Java	4763	Google
Iosched	Java	21624	Google
Spring-framework	Java	257910	Spring-projects
Fetch	JavaScript	434	GitHub
React	JavaScript	81551	Facebook
Meteor	JavaScript	156780	Meteor
Reddit	Python	50746	Reddit
iPython	Python	126665	iPython
Tornado	Python	19756	TornadoWeb
Mongo	C++	250912	MongoDB
Hhvm	C++	312159	Facebook
Atom-Shell	C++	18328	Atom

In the same manner as the random repositories, these 15 popular selected repositories are tested in more detail using GitHub API [59]. The repository statistic API is chosen for this task, as it provides an API that allows fetching of data that GitHub uses for visualizing different types of repository activity. The main data needed for characterization is commit frequency, number of commits, and size of commits. To pull the number of additions and deletions, the code frequency command is used. This command returns the weekly number of additions and deletions pushed to a repository over the past year (additions are lines added and deletions are lines deleted). The format for this is the first number is the start of the week, as a Unix timestamp, and the second and third numbers are the additions and deletions. Each week is separated by brackets. To retrieve the number of commits on a repository, GitHub API offers a participation command, showing the total commit counts for the past year.

Rather than use individually formed HTTP GET requests, Curl program is used [62]. Curl is a free, open source software that consists of a command line tool and library for transferring data via URL syntax, supporting multiple protocols. It is used in thousands of applications for its ease of use, which is the reason why it was chosen to be used in this project. As examples, the code frequency and participation requests in Curl are:

```
curl -i https://api.github.com/repos/:owner/:repo/stats/participation
```

```
curl -i https://api.github.com/repos/:owner/:repo/stats/code_frequency
```


The output of these requests is JavaScript Object Notation (JSON) format, a simple key-value pair format, and is stored to a text file that can later be parsed and converted to a csv file for easy reading into RStudio for analysis.

Every software project on GitHub is not going to be constantly worked on every week, or have data pushed to it on a weekly basis. By separating the commit and data statistics by week, one can choose to only look at weeks in which work was done and then extrapolate over a year, or look for the full year as an average of the entire work done on the software project. Having the weekly commit frequency and the weekly lines of code additions and subtractions data belonging to those commits gives the necessary information to emulate software repositories. Pulling this data from multiple languages and sizes allows one to see if there are any large differences in the way repositories are worked on, based on size or language. Results of repository commit and participation rates are shown and analyzed in the Chapter IV.

Phase Three: Secure Git Improvement Phase

The final phase of this methodology consists of testing an improved Git encryption implementation, GV2, developed by the author of this thesis. The testing methodology is the same as Phase One - in terms of CPU time, size, and functionality. The testing consists of comparing unencrypted Git performance to encrypted Git performance. GV2 was briefly described in Chapter II and provides improved authenticated encryption standards of Git-crypt but overcomes some of the performance shortfalls, mainly the massive Git repository size increases over series of commits. This

testing methodology phase shows that GV2 performs better than the previous Git encryption implementations in key performance areas of size and CPU time.

GV2 must have practicality for use determined according to the standard usage of Git provided by the Phase Two Git Characterization Study. The Secure Git Improvement Phase is the final phase of this thesis research and produces a product that demonstrates an efficient and secure Git implementation. In the end, it is up to the end-user to determine validity of software tool functionality and performance, according to their purpose, but with the characterization study research of mining real-world Git repositories, the end-user is able to fit their organization's software development uses into the test and determine usefulness of a secure Git implementation under their work habits.

Chapter II briefly describes two external Git libraries: JGit and libgit2 [48, 49]. JGit is chosen as the library to use for this Phase of research because of its increased functionality over libgit2 and portability of Java. Java is inherently slower than C because it is an interpreted language built upon a virtual machine known as the Java Virtual Machine [63-65]. Java code is compiled into bytecode that is run on Java Virtual Machine, which makes it slower than code compiled directly into binary sets of native instructions, such as C or C++. In recent years, however, Java has begun to bridge the performance gap with native languages.

JGit allows combinations of traditional Git to be used in parallel with JGit, helping to overcome this speed performance deficiency. Switching from traditional Git to JGit commands, and vice versa, is transparent to the Git repository. In addition to a large

API, JGit offers a command line interface. JGit was designed in a modular fashion and allows for code to hook into almost every aspect of the internal structure of Git.

JGit contains five different levels, or layers, of usability and functionality for interaction [66]. Level 0 is simply calling the executable JGit program. Commands at this level are called from the command line through a shell script or by executing the .jar file. Level 1 consists of embedding JGit into an existing Java process by using the JGit program's Main class, *org.eclipse.jgit.pgm.Main*, and invoking the Main method. This is similar to level 0 but has the advantage that a new Java Virtual Machine does not need to be created every time JGit is invoked, allowing for quicker execution of multiple commands.

The middle layer, level 2, involves using the JGit's Git class to wrap a Git repository and provide a set of porcelain commands. Porcelain commands are defined by Chacon as verb commands doing low-level work on a Git repository [2]. The lower layers of Git functions that the porcelain commands works on is called the Git plumbing. JGit Level 2 is one of the more popular levels that JGit contains, according to JGit users, because it provides simplicity and flexibility with the option to use debugging tools within an IDE [66].

Level 3 consists of the option to build porcelain commands by invoking instances of the JGit Repository class. This class allows developers to obtain more specific Git repository information, such as Git references, or to open specific branches and traverse them in the repository. The level 3 layer provides JGit users who are so inclined to add or modify the level 2 commands, but they are limited by the read-only nature of the

Repository class. They can combine this with RevWalk in order to iterate over commits, and essentially walk the whole repository. If one wants to modify a repository, the final layer, level 4 comes in. Level 4 improves upon level 3 by adding read and write access and allows JGit users to get objects into and out of a repository via the ObjectInserter and ObjectReader classes.

The levels of JGit functionality previously discussed provide a wide-range of uses, depending on the level at which one wants to access and modify a Git repository. These levels are the typical way in which developers use JGit, with level 2 being the most popular method [66]. All of these levels, however, deal with using JGit as it currently exists. There are more ways to intertwine specific JGit commands through code, as well as modifying JGit functions or adding new ones so that commands work uniquely to how the developer wants them to. JGit is open source and readily available for download and personal modification or community improvement via GitHub [48].

Recall from Chapter II that traditional Git uses filters to modify data in transit to or from the repository [1]. Both Git-encrypt and Git-crypt use these smudge and clean filters to encrypt data as it passes to the Git repository and decrypt as it comes from the Git repository [43, 44]. Because the data is encrypted before it is added to the repository, the Git garbage collection delta compression routines are ineffective, as they are not programmed to operate on encrypted data.

JGit has options for data modification using methods other than the smudge and clean filters [48]. JGit contains a Transport package that handles the transport network layer functionality of the repository. This layer deals with remote repositories, such as

repositories that reside in a cloud storage provider - the goal of this project. Within the JGit Transport package, a WalkEncryption class allows for encrypting or decrypting objects in the form of input and output streams of data being transferred at the transport level. This is monumental functionality for the goals of this project because it means that the Git repository is able to perform its built-in functions on an unencrypted repository and then compress the data via the garbage collection routine before uploading an encrypted stream to a local network or remote cloud service provider. Since the encryption occurs on the client side, the data is secure after it leaves the client, allowing for secure repositories to exist on untrusted cloud service providers.

The current implementation of the JGit WalkEncryption class has a password based key derived using a locally stored password in a password based encryption (PBE) algorithm with an MD5 hash to generate the key. Once the key is generated, DES encryption is used. JGit contains an AmazonS3 class that uses JetS3t (pronounced “jet-set”) to ease the functionality of operations on objects between JGit and AmazonS3 [67]. JetS3t is an open source Java library toolkit that contains functionality to link projects to cloud service providers such as AmazonS3, Amazon CloudFront, and Google Storage Services.

The PBE algorithm with MD5 and DES is not within the cryptography goals for this research that are outlined in Chapter II. A password chosen by the user is not securely random [27]. Additionally, AES-CTR mode encryption is desired for this project. Java Development Kit does not inherently contain AES-CTR mode functionality. The solution for this is to integrate Bouncy Castle, a library of Java cryptography

functionality [68]. Bouncy Castle version 1.51 is used. 128-bit AES-CTR mode is used for data encryption and decryption. AES-CTR mode encryption relies on the property that the same initialization vector is not used to encrypt more than once. For this, the Java Secure random class is used [69]. This class provided a cryptographically strong random number generator with compliance to statistical randomness tests specified in FIPS 140-2, Security Requirements for Cryptographic Modules [70]. Additionally, the output sequences are cryptographically strong, following the description of RFC 1750, Randomness Recommendations for Security [71]. For integrity protection, GCM, as described in Chapter II, is used in conjunction with the AES-CTR mode encryption, providing authenticated encryption [31]. The decryption process of GCM either outputs the plaintext or fails, indicating that the data is not authentic. Thus, if the data is modified by an unauthorized user while in the cloud, the user is alerted by an exception when trying to decrypt.

The size overhead of using GCM encryption is shown in Figure 16. Note that the plaintext data being encrypted has already gone through Git Garbage Collection, thus allowing for efficient compression of the plaintext data. Encryption before GC, as is the case with any encryption scheme that uses Git filters, does not allow for this compression.

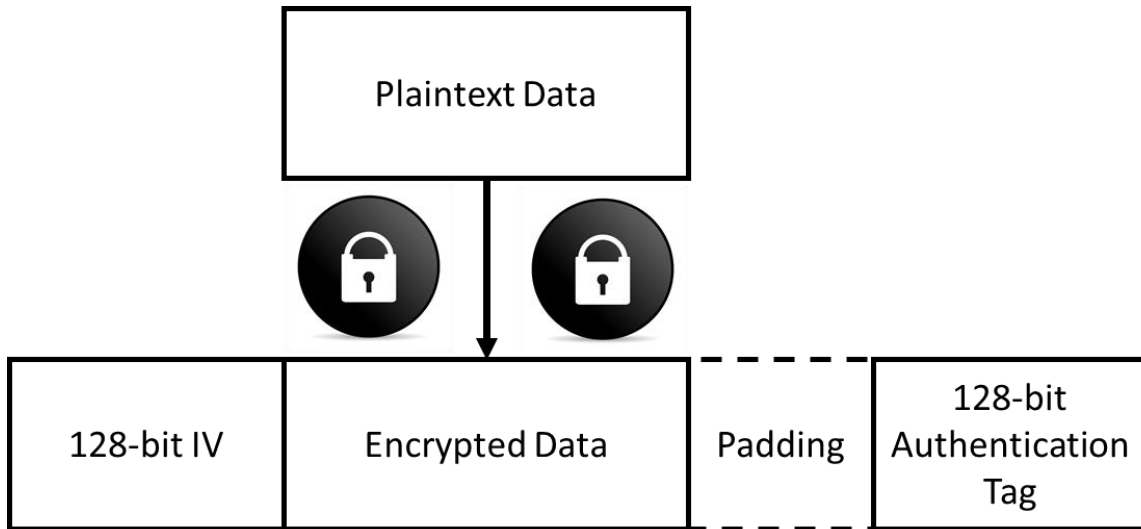


Figure 16. GV2 Repository Size Increase Diagram

Figure 16 shows the added components of overhead with the GCM encryption method used in GV2. First, the 128-bit IV is generated and pre-pended to the data in transit. Note that the IV is not encrypted – the only rule for it is that it is unique. The data is then encrypted (and optionally padded to fill out to exactly a block on the end). Lastly, the GCM authentication tag is appended to the end of the data. The maximum size of padding is 127 bits, for a total of 128 bits + 127 bits + 128 bits = 383 bits. This is less than 48 bytes of data. This again is minimal, as the popular repositories from Phase Two ranged from 434 KB to the hundreds of megabytes. Even just a 1% increase in size for the 434 KB Fetch program is 4.34 KB, or 4340 bytes. With Git-crypt, the best case size increase in terms of performance is 350%, with the Git Program. This size increase only gets worse as it has a linear increase and after five iterations of editing every file, the size increase in the repository is well over 1000%, compared to unencrypted Git.

The Git repository contains snapshots of the files that are being tracked by Git and under version control. The primary purpose of securing Git is to secure these files, which are stored as objects in Git, by providing confidentiality and integrity protection. This is what Git-crypt and GV2 do. Additionally, the remainder of the Git directory is examined to ensure that no other information is leaked. The .git directory folder stores the administrative portions of a Git repository, such as the configuration of the Git repository and the locations of branches and commits. [72]. The structure of a Git repository looks complex, but is easy to understand. Not all Git repository structures are identical – every .git structure is unique to the repository where it resides. Figure 17 shows a sample .git directory:

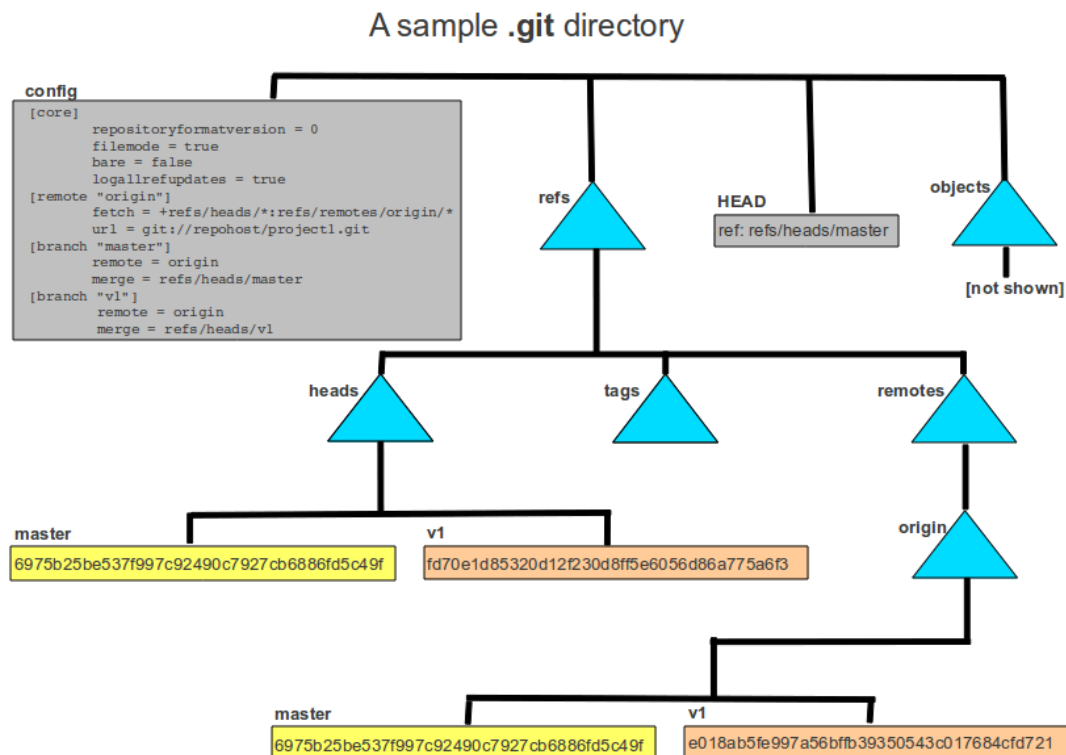


Figure 17. Sample .git Directory [72]

The sample .git directory structure in Figure 17 is a typical example of a Git repository structure. Table 4 explains what the files (or directories) and values associated with them mean:

Table 4. Sample .git Directory Explained [72]

File/Directory	Meaning
Config	The configuration file for the local git repository
HEAD	Lists a file to read that is the current HEAD branch: git branch will show the branch that HEAD refers to as the current branch.
Refs	Everything under the refs directory is references to a commit for a branch of some type (either a local branch or a remote tracking branch).
refs/heads	Files in the refs/heads directory are branch names. For example a file named refs/heads/master means a branch named master exists in the local repository. The contents of the file is the hash of the most recent commit on that branch.
refs/heads/master	The most recent commit on this branch (master).
refs/heads/v1	The most recent commit on this branch (v1)
refs/remotes	Everything under the refs/remotes directory is references to a commit for a remote-tracking branch.
refs/remotes/origin	The remote tracking branches for the remote repository origin are stored in this directory.
refs/remotes/origin/master	Note the hash is the same as in refs/heads/master which means the user has merged the commits from this remote-tracking branch into the user's master branch.

The Git Config file and the Git references stand out. Recall from Chapter II that Git references are SHA-1 hashes [1]. These hashes are easily computed and can leak information about a repository. For example, if a file contains the text “Hello World” and is stored in the repository, a malicious user may know that “Hello World” is common source file contents and have a pre-computed SHA-1 hash for “Hello World.” This hash

could be stored along with several other pre-computed hashes, in what is known as a rainbow table [73]. The rainbow table is then compared to the references in the .git directory and if a match appears, then the plaintext corresponding to the reference becomes known based on the plaintext that generated the reference in the rainbow table. The above example is clearly fictional, but imagine if instead of “Hello World,” the files were actually a set of encryption libraries with a known vulnerability. In this case, the information leaked is critical to an attacker and something that the software developer does not want exposed.

To defend against a rainbow attack vulnerably, the references must be securely hashed by using a keyed hash algorithm. This could have an effect upon the functionality and performance of Git. GV2 encrypts the .git directory files through the same GCM AES-CTR mode encryption functions that encrypt the repository file data objects. This solves part of the issue, but the filenames of some of the .git directory files contain references. The structure also gives away that the directory is a Git repository, which the owner may want to keep secret as well. In order to secure this, the filenames are run through a keyed hash algorithm, HMAC, prior to storage on the cloud in Amazon’s S3 storage.

To test this new Secure Git implementation, a baseline is first set. This baseline consists of testing the popular Git program repository. The tests are similar to Phase One:

1. A test to compare the speed of the new secure Git implementation compared to unencrypted Git.

2. A test to compare the size of the repository using the new secure Git implementation with the size of the repository using unencrypted Git.
3. A test to compare size growth of the repository as files are modified and committed to the repository.
4. Lastly, a test to compare JGit speed performance to Git.

The results of these tests provide enough data to extrapolate and determine roughly the overall performance of encryption using GV2. It is concluded that there are no further tests needed, as discussed in the analysis section in Chapter IV for Phase Three.

IV. Analysis and Results

Chapter Overview

This chapter presents results and analysis of the testing described in the methodology section. The methodology testing described three phases:

1. Secure Git Baseline Phase: baseline current Git encryption implementation performance
2. Git Characterization Phase: characterize Git usage in real-world environments
3. Secure Git Improvement Phase: develop a new improved Git encryption implementation

The results and analysis from the first phase of research is presented and the results compared to unencrypted Git. Next, the results of the Git Characterization Phase are discussed, with analysis of the implications as to the speed and size tradeoffs that are acceptable with real-world Git repositories. The third phase results are discussed and GV2 is compared to unencrypted Git and also to the Git encryption implementations tested in Phase One. GV2 is shown to be capable of providing a useful secure Git program for a wide range of audiences, which provides a much needed research community development. Finally, the chapter concludes with an overall analysis of the whole range of testing.

Phase One: Secure Git Baseline Phase

The methodology for this phase of testing is described in the previous chapter. This phase consists of testing existing Git-encryption implementations and obtaining a baseline of their performance compared to unencrypted Git. The testing analyzes

performance in terms of CPU time, size, and functionality. Three distinct real-world Git repositories are tested: the Linux Kernel, the Git program source code, and Popping program. They are tested in three distinct test scenarios, described in the previous chapter. The results show how each test compares in performance between unencrypted Git and the two Git encryption implementations: Git-encrypt [43] and Git-crypt [44]. The first test analyses the time it takes for all of the files in a repository to be added and committed to a new repository. This test measures the time it takes each encryption program to use filters to encrypt the files, compared to unencrypted Git, and is run over a series of 10 iterations.

Phase One, Experiment I: Adding all files to the initial repository

The first test subject is the Linux Kernel. The performance time for adding all of the files to an empty repository, then committing them, then repeating 10 times is shown in Figure 18. The difference in speed is due to the filters being used in the Git encryption implementations. These encryption filters slow 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. This time increase is a one-time penalty and is proportional to the amount of data added, as the data encryption determines the speed penalty compared to unencrypted Git.

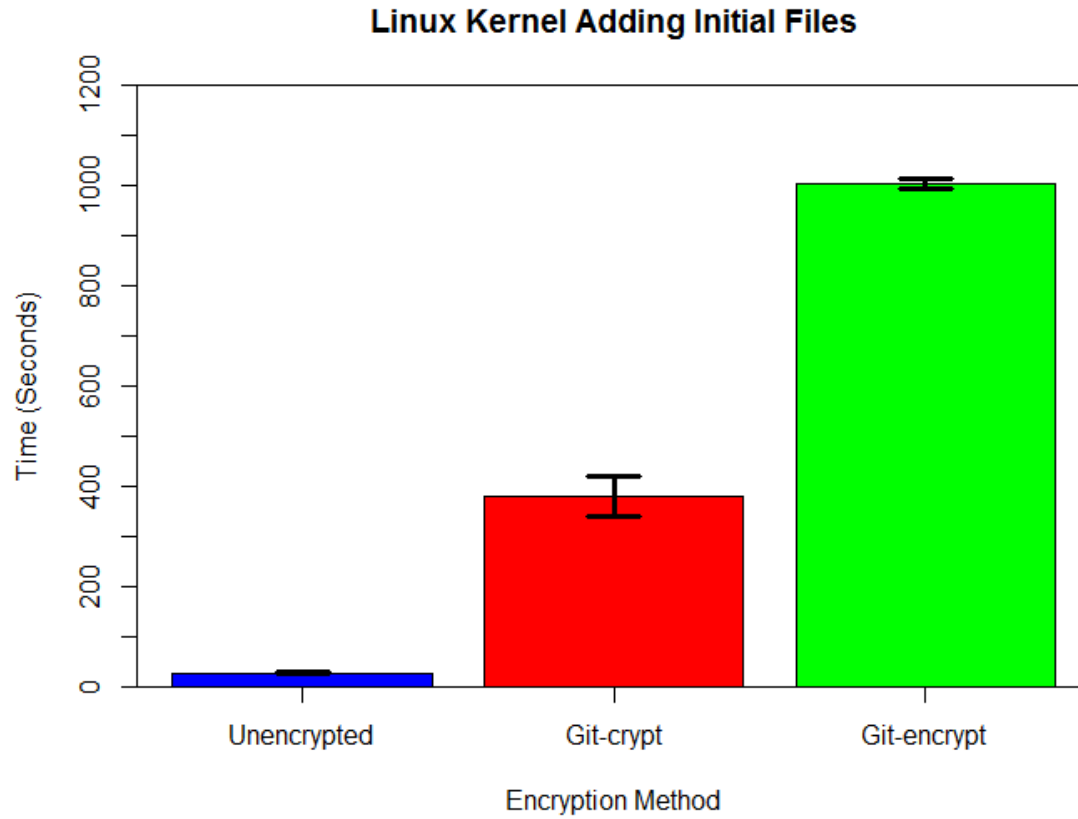


Figure 18. Phase One, Experiment I: Linux Kernel

The exact CPU time averages for each of the three implementations is shown in Table 5:

Table 5. Phase One, Experiment I: 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 for Experiment I is the Git Program. The results of Experiment I using this program are shown in the Figure 19:

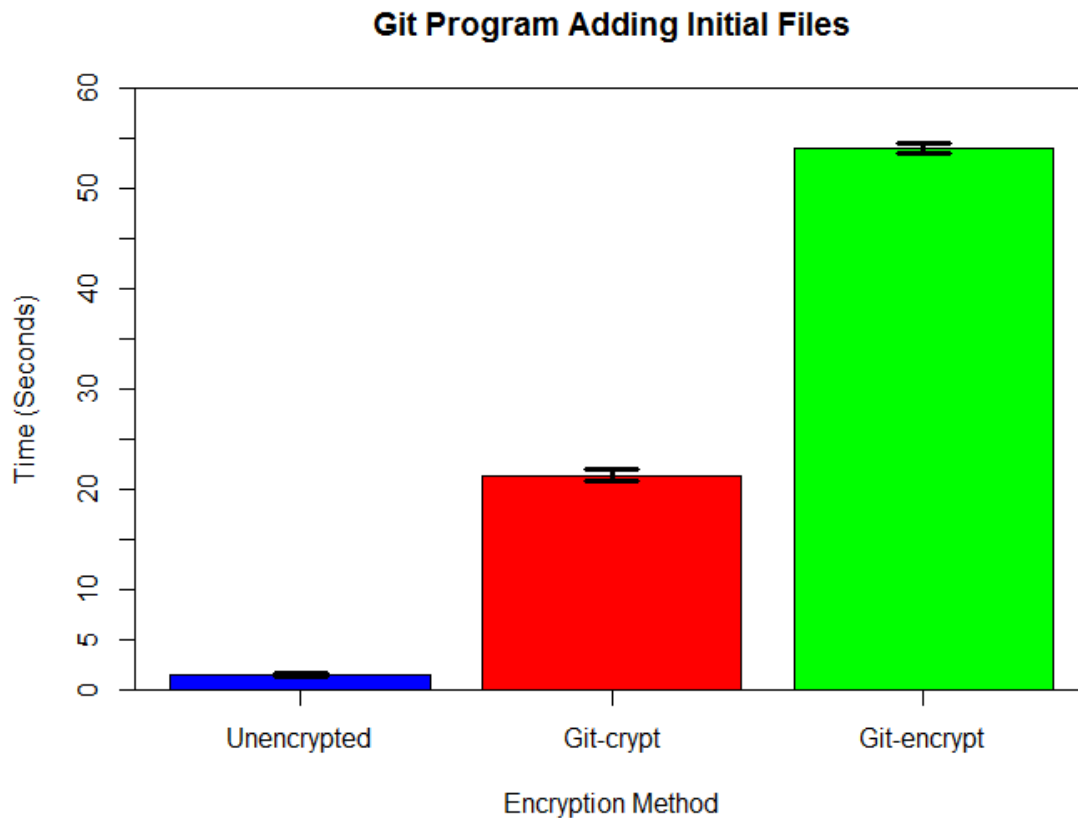


Figure 19. Phase One, Experiment I: Git Program

The time increase ratio for Git encryption compared to unencrypted Git for the Git Program is roughly the same as 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. The Git program is a much smaller project in terms of size and number of objects when compared to the Linux kernel and the time Experiment I takes reflects that. The averages in time for each of the three implementations are shown in the Table 6 :

Table 6. Phase One, Experiment I: 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 20

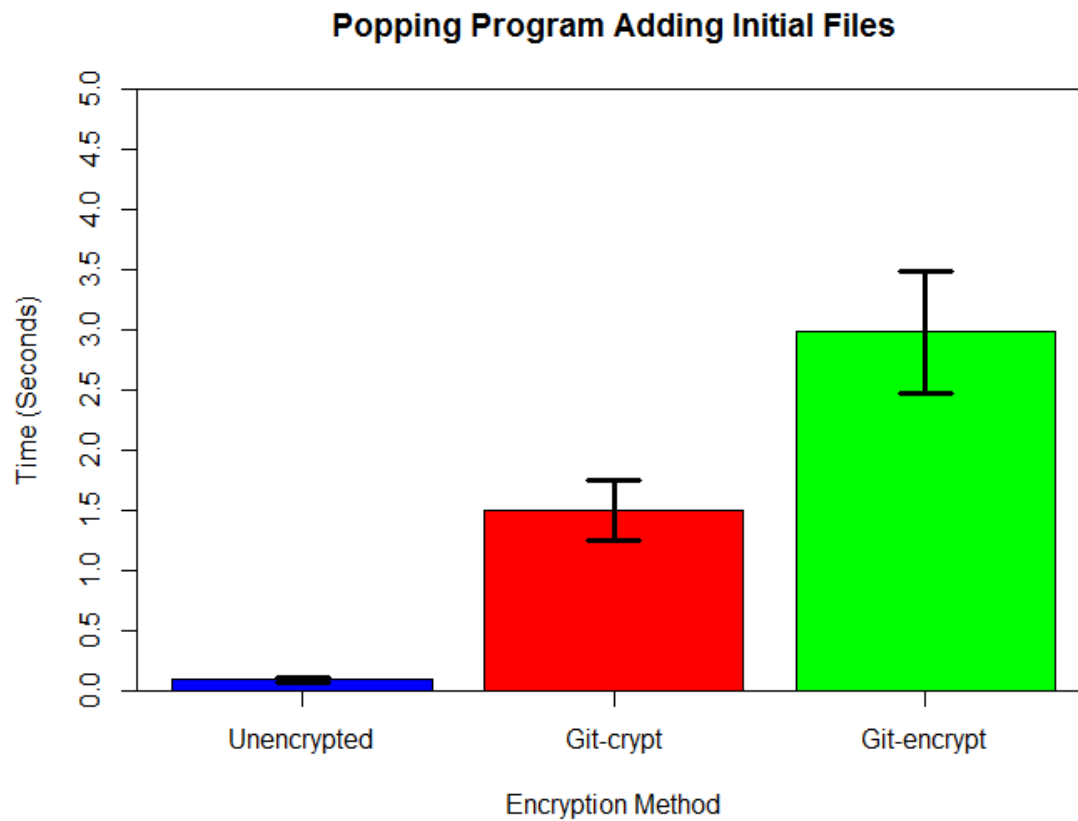


Figure 20. Phase One, Experiment I: Popping Program

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 still very similar to the previous test subjects. The exact averages in time for each of the three implementations are shown in Table 7:

Table 7. Phase One, Experiment I: 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

Phase One, Experiment II: Initial size comparison

Experiment II measures the size directly after the process of adding and committing all files to the repository, as is done in Experiment I. This size is compared between all three Git implementations and then Garbage Collection is run and the sizes compared again. In terms of size for the Linux kernel, 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 the Git garbage collection (GC) algorithm is unable to efficiently delta-compress and combine certain parts of the blob that have similar plaintext but very different ciphertext. The graph of results is shown in Figure 21:

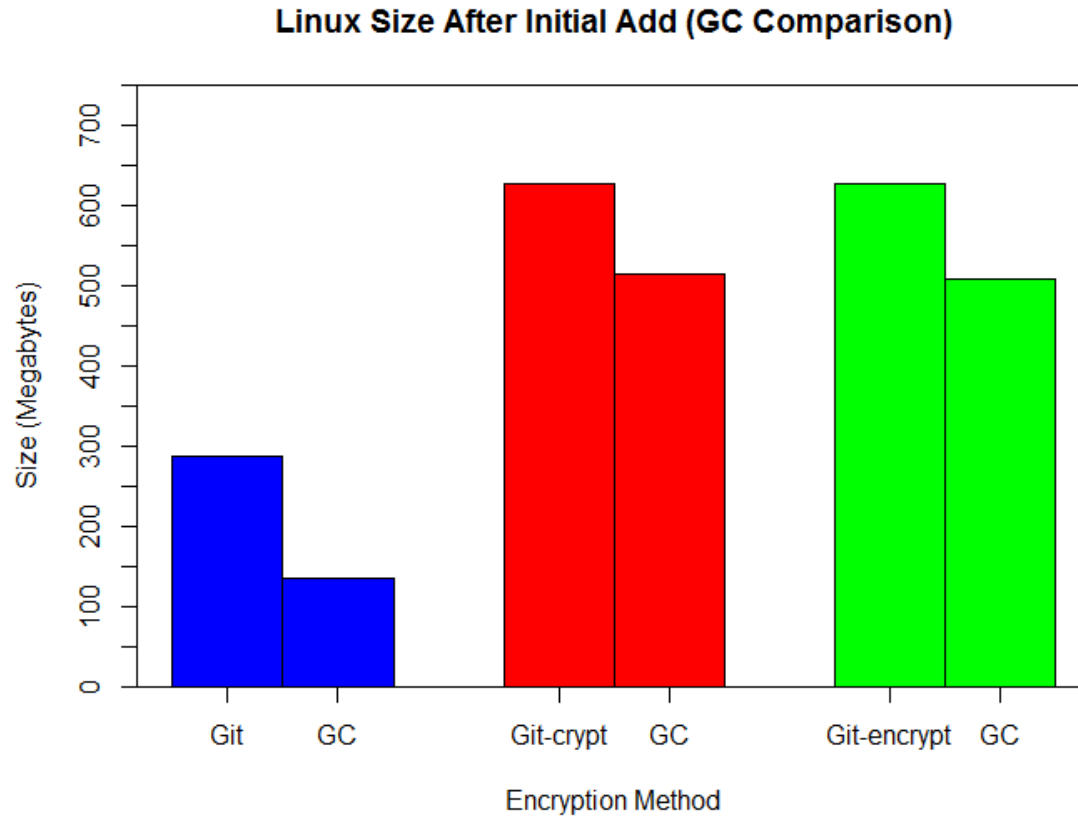


Figure 21. Phase One, Experiment II: Linux Kernel

The size of each implementation directly after the commit and after garbage collection is run is shown in the Table 8:

Table 8. Phase One, Experiment II: 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

For the test of the Git Program, 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 inability of Git garbage collection to efficiently delta compress. The graph is shown in the Figure 22:

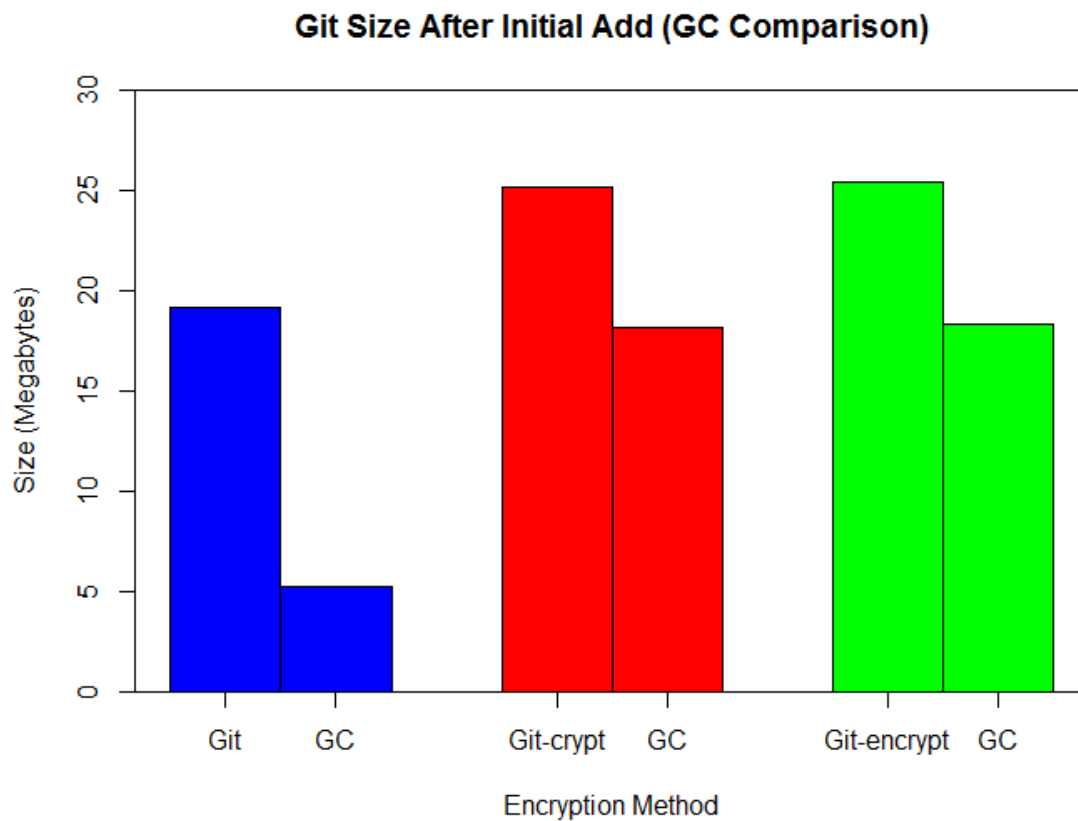


Figure 22. Phase One, Experiment II: Git Program

The size of each implementation directly after the commit and after garbage collection is run is shown in Table 9:

Table 9. Phase One, Experiment II: 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 test yields a repository 80% the size of both encrypted versions. When garbage collection is run, the unencrypted repository is reduced by 75%, whereas the repository sizes when using the encrypted implementations of Git are reduced by 20%. This is the same type of reduction as the Git Program. The graph of results is shown in Figure 23:

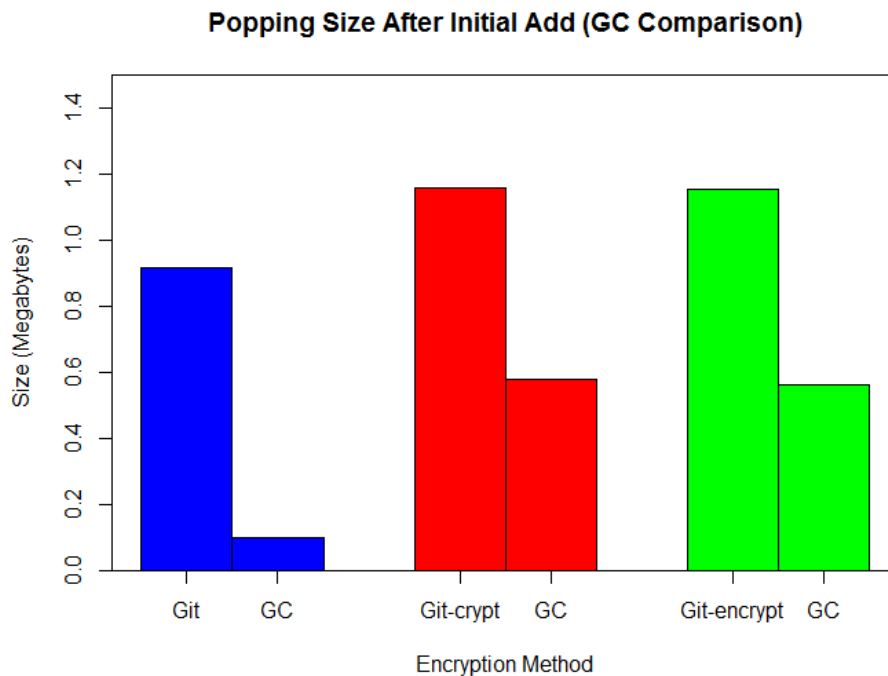


Figure 23. Phase One, Experiment II: Popping Program

The size of each implementation directly after the commit and after garbage collection is run is shown in %able 10:

Table 10. Phase One, Experiment II: 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

The size of the repositories using Git encryption implementations is worse than with unencrypted Git. The initial size is worse because Git uses zlib to compress data in the repository and encrypted data is more random than unencrypted data and cannot be compressed at the same level. The size after garbage collection is much worse for the encrypted versions and the efficiencies of garbage collection depend on being able to delta compress files. As a result, much of the redundant plaintext data must be stored as ciphertext, because the ciphertext cannot be compressed. The next experiment takes a further look into the size increase performance penalty for Git encryption implementations.

Phase One, Experiment III: Size growth with file modifications

The final experiment is performed by editing every data file in the repository by appending “hello” to the end of the file and then staging all the modified files by adding them and then committing them to the repository. This process of editing the files and then committing them is repeated five times.

In the case of the Linux kernel, the growth in size is measured as this process is repeated five times. Garbage collection is not run until the fifth iteration. The repositories all grow at the same rate of adding 287 MB of data every iteration. This is because there is no delta compression due to garbage collection, so every new file snapshot is stored in the repository. After the fifth iteration, garbage collection is run and the results of the repository after one iteration and after the fifth (with garbage collection) are shown in Figure 24:

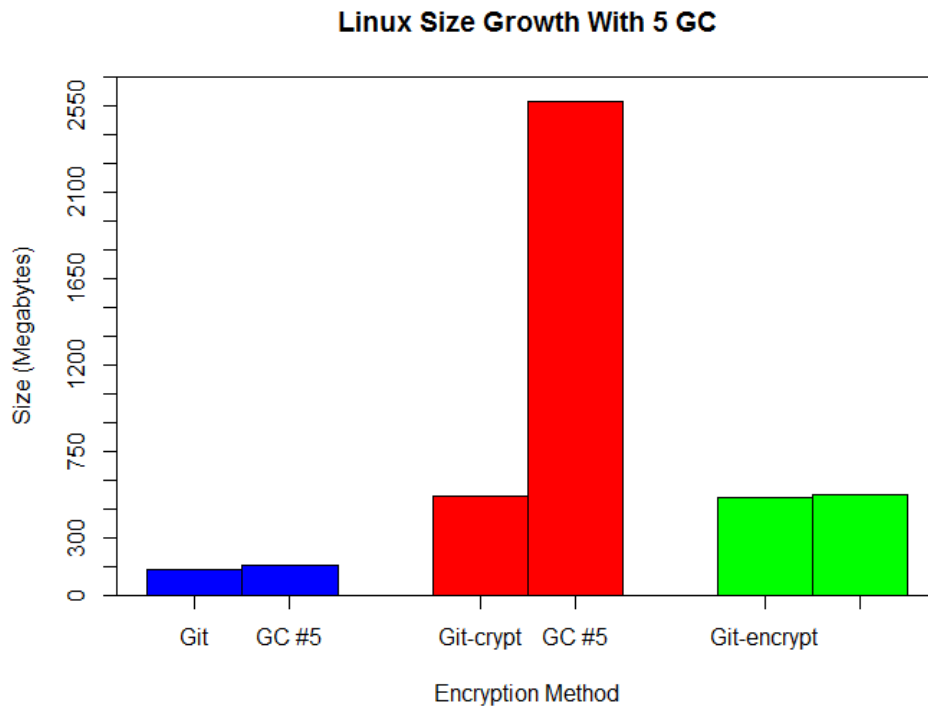


Figure 24. Phase One, Experiment III: Linux Kernel

The results show that the size of the Git repository using Git-crypt grows very fast because of failure to be condensed as with Experiment II. The unencrypted and Git-

encrypt repository grow at a much slower rate because Git-encrypt uses ECB mode, allowing for efficient delta compression. The exact sizes are shown in the Table 11:

Table 11. Phase One, Experiment III: Linux Kernel

Type	Size after fifth iteration garbage collection:
No Encryption	152 MB
Using Git-crypt	2577 MB
Using Git-encrypt	525 MB

The Git Program is the second test subject. In the same manner as the Linux Kernel, the Git program grows linearly for Git-crypt, and does not grow at a high rate for the other two implementations. The difference in this test is that the garbage collector is run after each iteration. This is to determine if it makes a difference when garbage collection is run, or how often. It is found that how often the garbage collection is run does not affect the compression performance. The results of this test are shown in Figure 25:

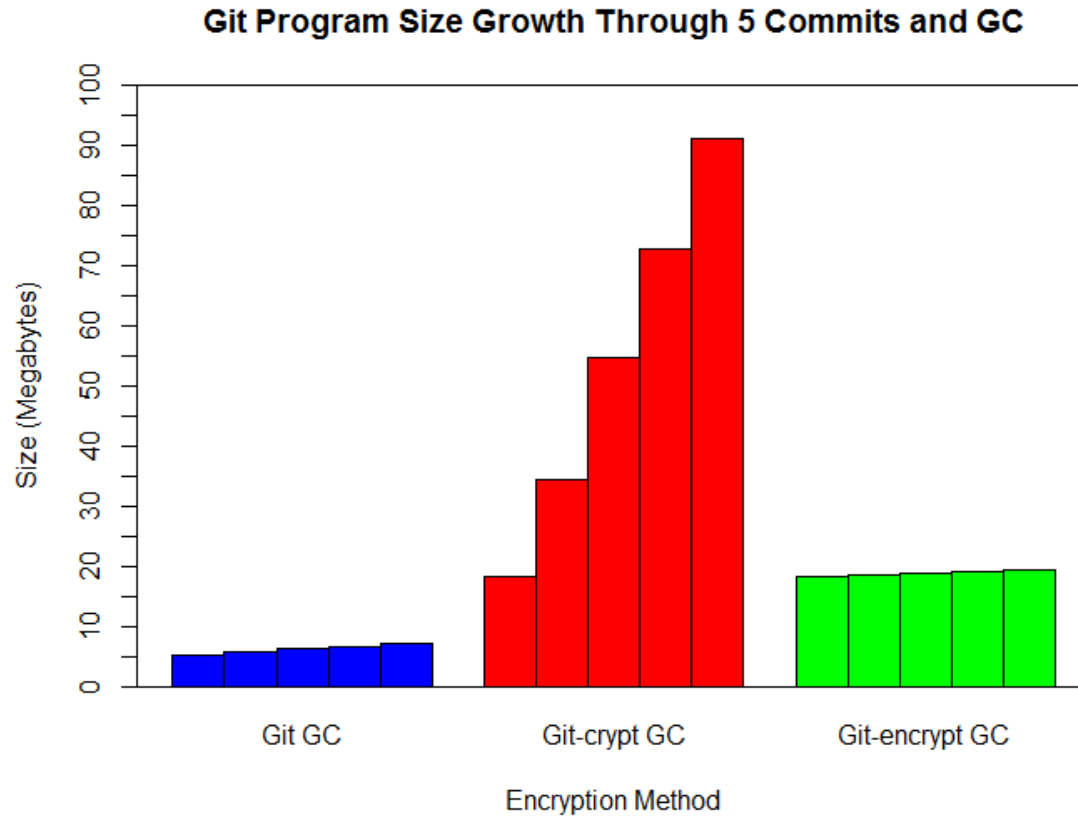


Figure 25. Phase One, Experiment III: Git Program

It is easy to see that the size of Git-crypt grows very fast in a linear manner relative to the other Git implementations. This result is the same as the Linux Kernel test. The exact values for each iteration after garbage collection are shown in the Table 12:

Table 12. Phase One, Experiment III: 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 is the last test subject and performs similarly to the Git program. It grows linearly using Git-crypt, and does not significantly grow in size for the other two implementations. This test is again run using the garbage collection routine after each iteration. The results of this test are shown in Figure 26:

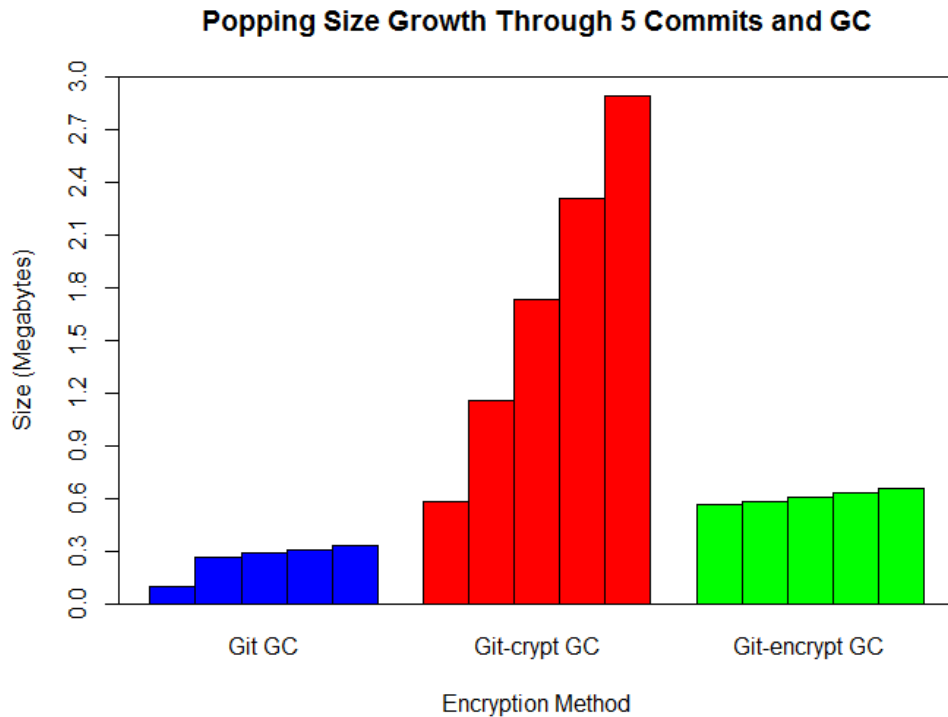


Figure 26. Phase One, Experiment III: Popping Program

Again, this graph reflects the same pattern as the Git Program test, showing that 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 after each garbage collection iteration are shown in Table 13:

Table 13. Phase One, Experiment III: Git 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 14. This table shows the baseline performance of Git without encryption is shown as both a normalized baseline value of 1 and also the actual value resulting from the corresponding test. Each of the two Git encryption methods are shown in the corresponding columns for performance comparison as a ratio for each of the experiments and test sizes:

Table 14. Phase One: Summary of Experiments

Type of Test (size category)	No Encryption (baseline) / Actual	Git-Crypt	Git-Encrypt
Speed ratio of repository init (Small)	1.00 / 0.085 seconds	17.66	35.06
(Medium)	1.00 / 1.47 seconds	14.53	36.76
(Large)	1.00 / 26.2 seconds	14.51	38.32
Size ratio of initial repository (Small)	1.00 / 0.1 MB	5.8	5.6
(Medium)	1.00 / 5.2 MB	3.5	3.52
(Large)	1.00 / 135 MB	3.81	3.76
Size ratio of growth with file changes after 5 iterations (Small)	1.00 / 0.329 MB	8.78	1.98
(Medium)	1.00 / 7.2 MB	12.65	2.69
(Large)	1.00 / 152 MB	16.95	3.45

Phase One examined two existing methods for securing Git repositories, Git-encrypt and Git-crypt, and compared their performance relative to unencrypted Git. They are tested side-by-side to unencrypted Git through a series of three tests. These tests examine the performance impact in terms of time, size, size growth with file modifications, and functionality of initializing and populating a repository, compressing a repository through garbage collection, modifying and 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. This is a constant time increase and attributed to the time it takes for files to be encrypted.

The size increase for the Git-encryption implementations in adding files to the repository is larger than unencrypted Git by a factor ranging from 3 to 9. This size increase occurred because Git compresses data objects in the repository with zlib. The randomness of the encrypted data does not allow for as much compression as with unencrypted data. These initial encrypted repositories also suffer from inefficient delta compression, causing the percentage decrease in size after Git garbage collection to be at a worse performance than with unencrypted Git.

The size growth of encrypted Git implementations compared to unencrypted Git is tested by editing every file in the repository and then adding and then committing the repository files. This process is repeated five times. Garbage collection is run after the fifth iteration for the Linux kernel and after each iteration for the other two test subjects. The size increase is similar for unencrypted Git and Git-encrypt and the size increase is large and linear 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 of that file 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.

A system is needed to provide a secure implementation for Git in order 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 efficient and tested method for this to be accomplished 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. Even though these tests represent the worst case scenario, Git-encrypt has an alarmingly high growth rate of the repository. This causes the size of the repository to quickly get out of hand for many software development environments. Git-encrypt provides better size performance, but at a high cost to security. GV2 reduces this growth rate to provide for better usability.

Phase Two: Git Characterization Phase

The methodology for this phase of testing is described in the previous chapter. The Git Characterization Phase consists of characterizing typical real-world Git usage habits in order to provide realistic and accurate tests to show that Git encryption works under realistic scenarios. The research of this phase is used to provide accurate information with which decisions can be made as to how well an encrypted Git implementation must perform in terms of size and speed compared to unencrypted Git. This phase is started with the intent to provide an emulator of realistic Git usage to use to test the Git encryption implementations and see how they perform under realistic situations ranging over multiple years. The reality shows, however, that the usage habits for any popular repository contains too many commits and pushes to be efficient under

the previous Git encryption implementations. GV2 performs so closely to unencrypted Git, that the emulation is not needed. To show the conclusions, the mined real-world Git projects are shown in Table 15 alongside their language, commits, and lines of code additions and subtractions per commit:

Table 15. Random Repository Commit Statistics

Name	Language	Commits/Week	Added/Commit (Lines of Code)	Removed/Commit (Lines of Code)
Devnull	PHP	14.6	501.2	(278.4)
ComputerNetworks	Python	23.5	50.9	(28.0)
Restfiddle	Java	22.0	215.3	(64.21)
Intouch2	JavaScript	25.75	459.2	(150.4)
CoolProp	C++	59.2	150.9	(90.3)
EZ Publish	PHP	39.36	116.1	(56.9)
1PICNIC	Python	42.8	14.5	(6.65)
TetraWord	Java	23.4	2149.5	(41.5)
Syra	JavaScript	14.7	261.0	(46.7)
FF2-Alpha	C++	19.36	103.5	(43.2)
Openvault	PHP	24.0	33.8	(34.3)
Tendenci	Python	19.5	2707.9	(1417.2)
Eucalyptus	Java	38.4	291.2	(171.9)
Boost32Boost	JavaScript	22.1	45.1	(30.9)
Nme	C++	9.4	566.7	(557.1)

The results of the random repository analysis are visually shown in Figure 27. The figure shows the number of commits to each repository per week based on language. The boxes represent the 95% confidence intervals and the thick lines in the boxes show the average of the commit data.

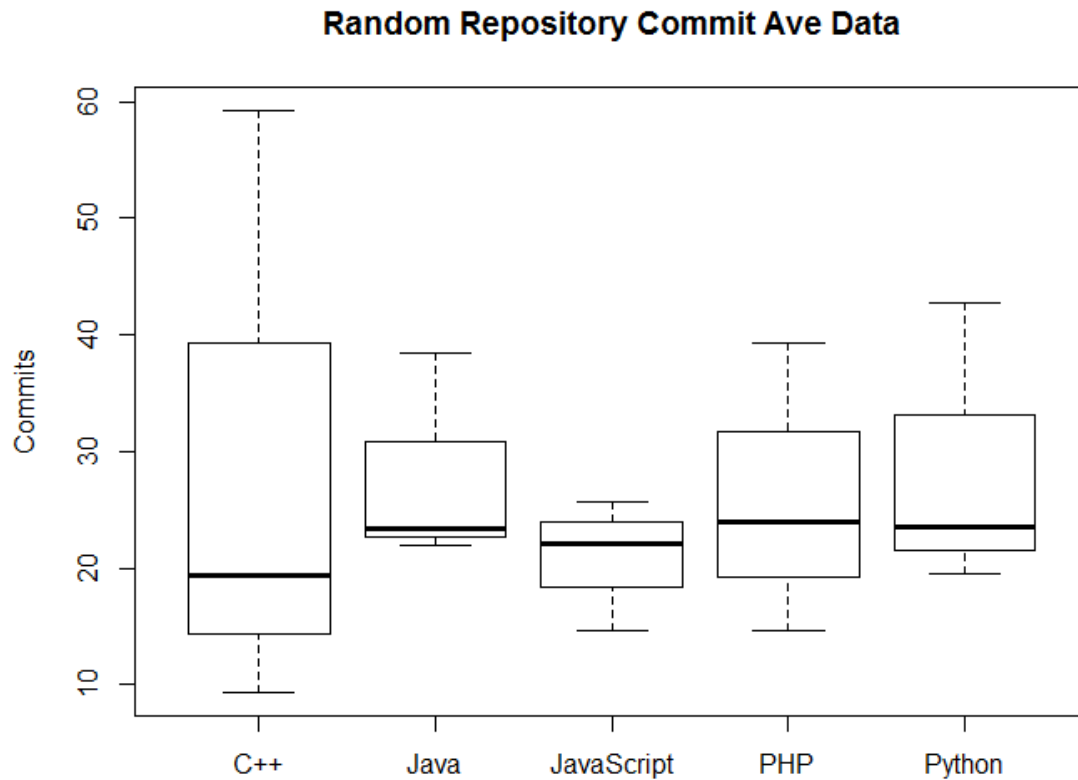


Figure 27. Random Repository Commit Average Data

The commit data is statistically similar within the 95% confidence interval that that habits for commits per week are not influenced by language of the project. The trending repository projects are shown in Table 16 alongside their language, commits, and lines of code additions and subtractions per commit. These trending repositories are

more accurate representations of what types of projects a company or Government organization would be working with:

Table 16. Trending Repository Commit Statistics

Name	Language	Commits/Week	Added/Commit (Lines of Code)	Removed/Commit (Lines of Code)
Typecho	PHP	40.1	169.9	173.1
PHPMailer	PHP	6.17	14.9	13.3
Google-api-php-client	PHP	17.3	187.2	92.1
Physical-web	Java	65.75	47.9	27.7
Iosched	Java	4.5	1273.5	583.3
Spring-framework	Java	40.0	178.5	127.9
Fetch	JavaScript	18.25	16.5	5.2
React	JavaScript	40.7	44.8	35.7
Meteor	JavaScript	123.3	120.0	105.4
Reddit	Python	24.7	43.7	18.1
iPython	Python	91.2	72.0	60.0
Tornado	Python	11.8	26.7	20.8
Mongo	C++	78.6	670.3	284.5
Hhvm	C++	110.5	527.2	429.0
Atom-Shell	C++	32.0	60.83	43.1

The results of the trending repository analysis are visually shown in Figure 28 below. Figure 28 shows the number of commits to a repository per week based on

language. The boxes represent the 95% confidence intervals and the thick lines in the boxes show the average of the commit data.

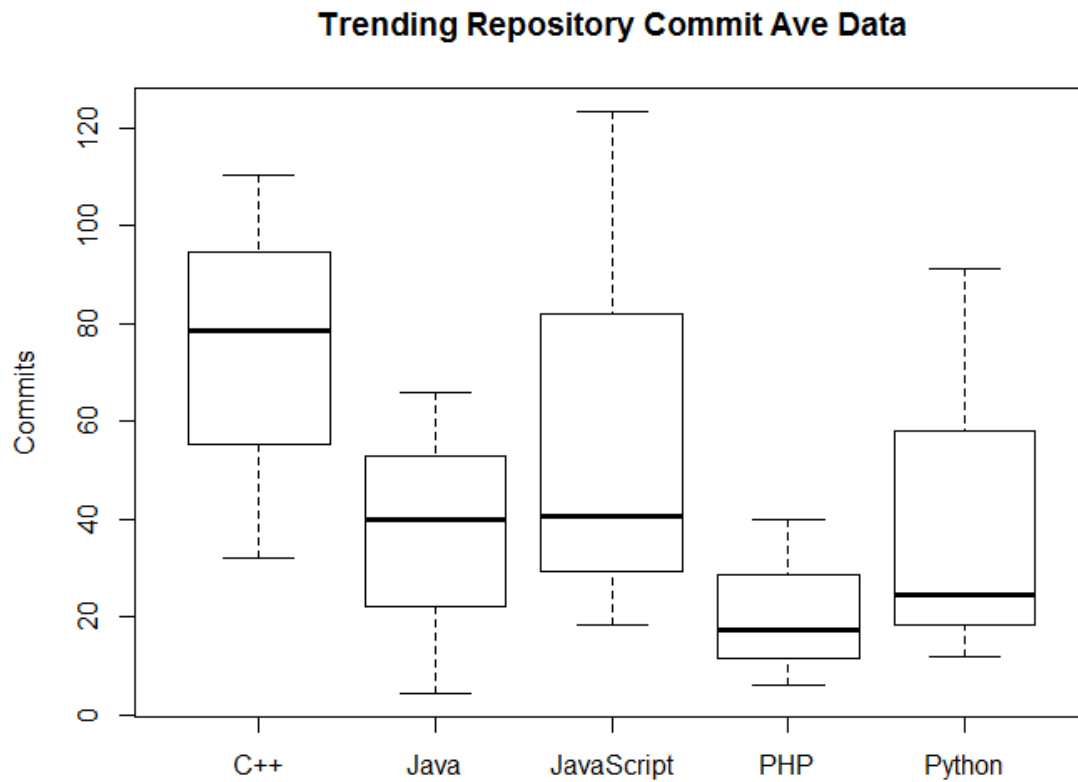


Figure 28. Trending Repository Commit Average Data

The commit data is statistically similar within the 95% confidence intervals to conclude that habits for commits per week are not influenced by language of the project. The average size of lines of code added per commit is shown in Figure 29:

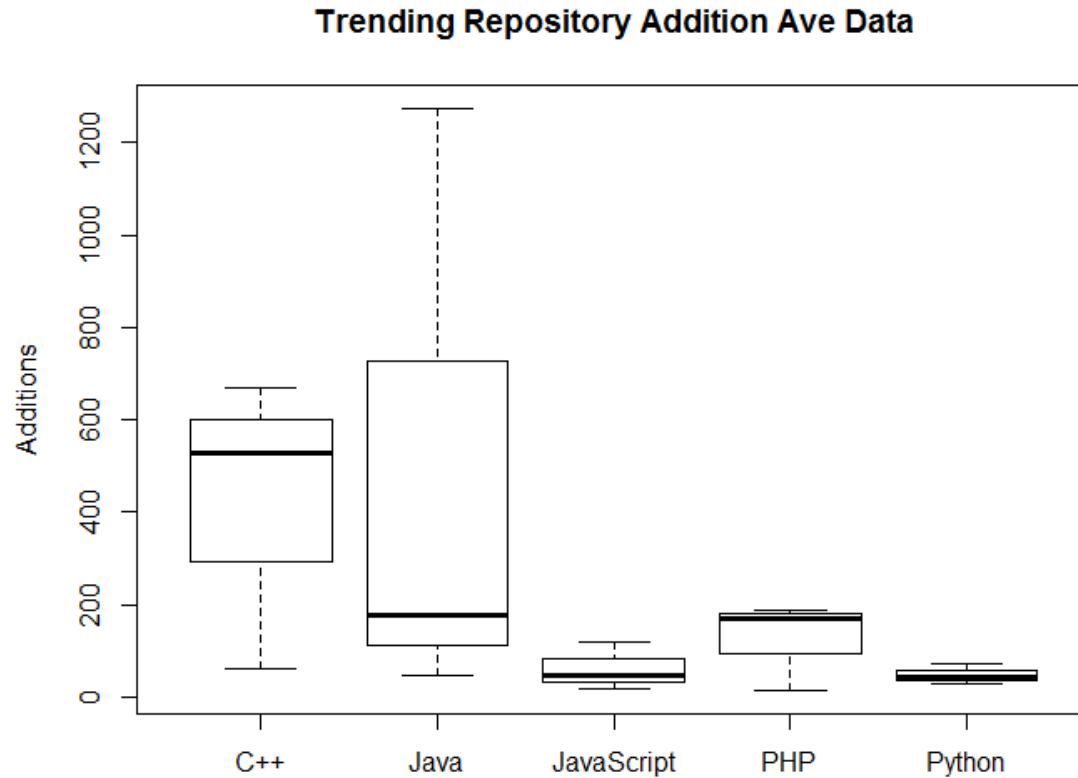


Figure 29. Trending Repository Line of Code Average Addition / Commit

Both C++ and Java contain larger code additions than JavaScript, PHP, and Python. This is attributable to C++ and Java having a higher level of verbosity compared to JavaScript, PHP, and Python. Similar statistics are expected for the subtractions, shown in Figure 30:

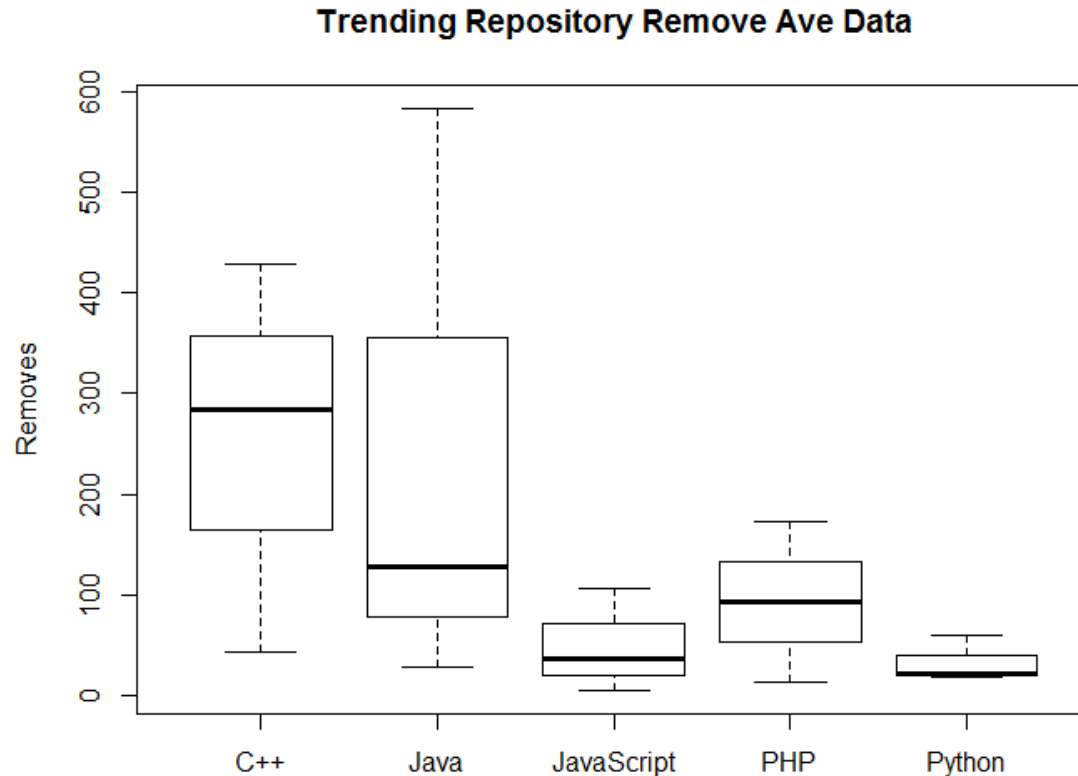


Figure 30. Trending Repository Line of Code Average Removal / Commit

As seen in the additions, both C++ and Java contain larger code removals than JavaScript, PHP, and Python. Again, this is attributable to C++ and Java having a higher level of verbosity compared to JavaScript, PHP, and Python..

If an emulator for Git were needed, this data could be used, but this data provides enough evidence to support that the linear growth of the size of Git-crypt occurring during code edits and commits will not suffice to support efficient secure Git encryption. This is because the Git Garbage Collection routine is run after encryption. Phase Three develops an improved Git encryption method, GV2, that encrypts data after Git Garbage

Collection and negates the need for an emulator to test, as the size growth and time expense are minimal penalties and predictable constants.

Phase Three: Secure Git Improvement Phase

This final phase of analysis consists of reviewing the results of the Secure Git Improvement Phase. The methodology for this stage is described in Chapter III. GV2 is developed by the author of this thesis and performs Git Garbage Collection prior to encryption. GV2 uses JGit as a base Git software and utilizes other third party libraries to aid in encryption and deployment to third-party Amazon S3 storage. GV2 addresses the performance shortfalls of the previously analyzed existing Git encryption implementations – most notably the performance shortfall of the linear repository size growth of Git-crypt. By carefully integrating these libraries and applying new code to JGit, GV2 is able to provide a usable secure Git implementation with full functionality and performance similar to unencrypted Git.

The testing performed on GV2 is in similar nature to Phase One of this research and is tested and recorded in terms of CPU time, size, and functionality. The performance compared to previous Git encryption implementations is better with regards to all three key performance metrics. As stated, the most critical performance shortfall addressed is the linear size growth of Git-crypt, when files are modified, the changes added to the repository, and then the changes are committed. This size growth causes Git-crypt to be unusable during typical Git usage. Because the reason for this growth is caused by the Git filters encrypting data before it is compressed and garbage collected, a total redesign of Git-crypt is necessary to fix this issue. This is one of the main decision factors to develop

an improved version, rather than continue testing and/or modifying the previous Git-crypt.

GV2 is briefly described in Chapter III and provides improved security standards from Git-crypt: the encryption methods used provide the same authenticated encryption but the .git directory files are also encrypted. GV2 uses GCM with AES-CTR mode encryption.

The test subjects and methodology process for comparison of unencrypted JGit to GV2 is identical to Phase One. The Git program is tested first as it represents a typical repository, based on the Phase Two study (the Popping program and Linux Kernel are on the extreme ends of the size spectrum). The first test is a speed test and this test is run over a series of 10 iterations using no encryption, the JGit built in DES encryption option, and GV2. The results show that the time increase of GV2 compared with unencrypted JGit is minimal. The majority of the time for a push to Amazon S3 is in the network transmission and propagation delay. The average time increase from unencrypted JGit to GV2 for adding all of the files of the Git program, committing them to a new repository, and then pushing them to S3 is 0.258 seconds, or an increase of 2.4%, over the average of the 10 tests trials. This test is run over a series of 10 iterations to provide consistency of results. DES encryption has slightly better performance, but no integrity protection and weaker encryption properties. The time results are shown in Figure 31:

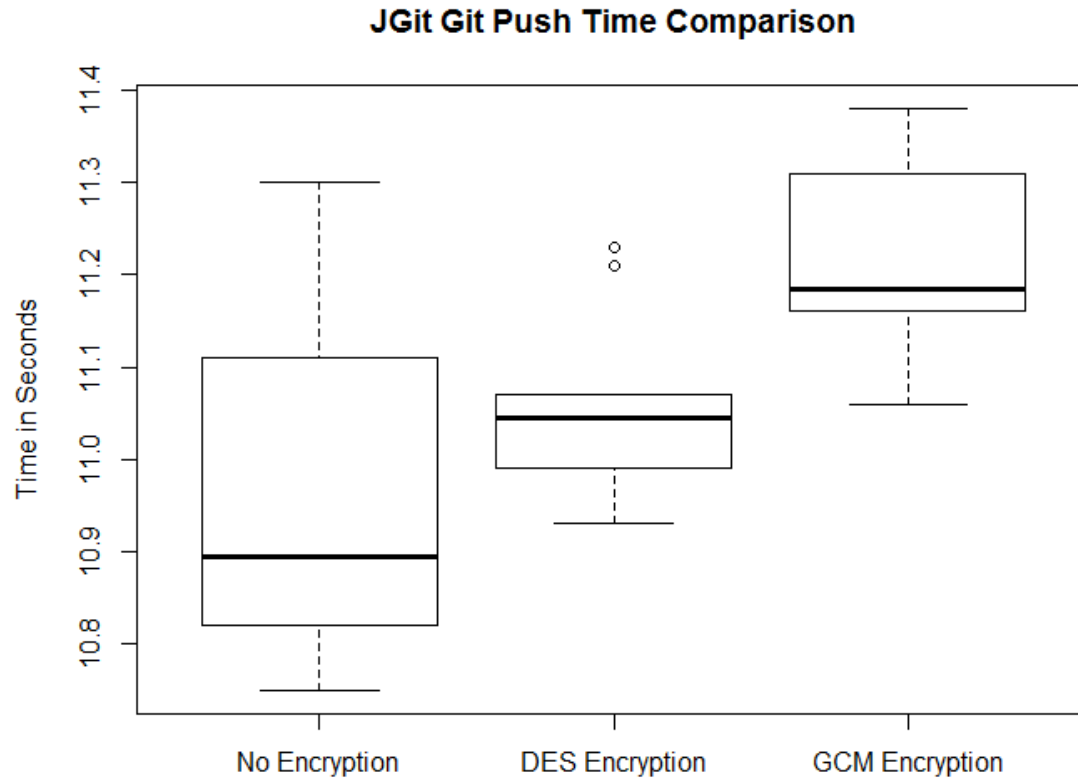


Figure 31. JGit Git Program Push Time Comparison

The results in the figure above show staggering support for the case to use GV2. The time increase for an encrypted push to Amazon S3 is minimal and negligible. The speed is decisively better than that of even Git-crypt. The reason for this is that Git-crypt encrypts every individual file before adding it to the repository. This requires encryption setup time and overhead for many files. GV2 first adds and commits all of the files and then runs GCM encryption in a streaming manner on the packfile produced from garbage

collection, prior pushing the repository to Amazon S3 storage. The order in which the encryption is done drastically lowers the overhead required.

The second test measures the size of the GV2 encrypted Git repository compared to unencrypted Git. Again, this size increase is also minimal for GV2. This again is attributed to the order in which the encryption happens. This implementation takes advantage of Git's intended design of being fast and efficient [1]; most notably Git garbage collection. Git garbage collection results in a data file, called the packfile, and an index file that contains the information necessary to decompress the packfile. By encrypting in this manner, the only size increase is in the encryption overhead, and also the padding, if it is used.

This linear size growth from Git-crypt testing does not exist in GV2. This is because the data can be compressed locally by Git garbage collection when it is unencrypted, and that is the process that this implementation uses. Since additional data added to the repository is either encrypted or decrypted after garbage collection, there is not an excess of extra data from different versions, as is the case with Git-crypt. Thus subsequent versions of repositories grow and shrink in the same manner that an unencrypted Git repository does. The only increase in size is the added overhead for encryption, as stated earlier.

The previous discussion of the size performance of GV2 is theoretical. In order to demonstrate it, the same size test from Phase One of this research is used. This test process consists of initializing an empty repository, adding all of the Git files to the repository, committing them, and then pushing this data to Amazon S3 storage. This is

done using unencrypted Git as well GV2. Since the data overhead for encryption is consistent, the test is done with one program, again the Git program. The contents of the .git directory files are encrypted, unlike in Git-crypt. This and the size comparison are shown in the Figures 32 and Figure 33:

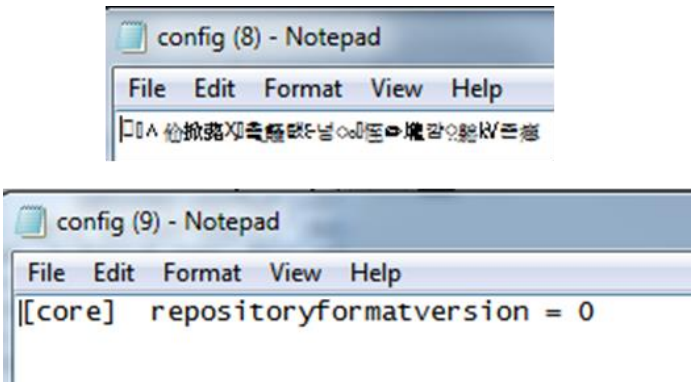


Figure 32. Git Encrypted and Unencrypted Config File Comparison

All Buckets / afitgitbucket / gcmtestofGitProg			
	Name	Storage Class	Size
<input type="checkbox"/>	HEAD	Standard	39 bytes
<input type="checkbox"/>	config	Standard	52 bytes
<input type="checkbox"/>	info	--	--
<input type="checkbox"/>	objects	--	--
<input type="checkbox"/>	refs	--	--
All Buckets / afitgitbucket / noncrypt11			
	Name	Storage Class	Size
<input type="checkbox"/>	HEAD	Standard	23 bytes
<input type="checkbox"/>	config	Standard	36 bytes
<input type="checkbox"/>	info	--	--
<input type="checkbox"/>	objects	--	--
<input type="checkbox"/>	refs	--	--

Figure 33. Git Encrypted and Unencrypted Structure Size Comparison

Figure 32 shows the contents of an encrypted and unencrypted Git config file. Figure 33 shows the encrypted and unencrypted contents of Git structure of the identical

Git repositories. Note that the sizes of the encrypted files are exactly 16 bytes larger than the sizes of the unencrypted files. This is the authentication tag. If padding is included, there would be additional size and the encrypted files would be multiples of 16 bytes, but for this purpose seeing the exact 16 byte size increase is important, so padding is not implemented in this portion of the analysis. The entire contents, including filenames can be encrypted as well, but then there would just be a blob of data, which makes analysis impossible because one would need to know the unencrypted Git structure in order to navigate the encrypted. This is because the names are encrypted using HMAC, which is a one-way function. Since every .git directory is unique, one would have to guess it along with the filenames and then HMAC, which is time consuming. Even so, if an attacker without access knows that the directory is a Git directory, they still have no ability to glean any bits of information about the data. The attacker also cannot modify data without alerting the integrity check. An example ‘scrambled blob’ directory is shown in Figure 34:

[All Buckets](#) / [afitgitbucket](#) / [scrambledGit](#)

	Name	Storage Class	Size
<input type="checkbox"/>	5lX6Z1zXWuKbe3bvxlscSRzs1w=	Standard	75 bytes
<input type="checkbox"/>	8xv-SlirRfM6DKpfSvFZzWeVh18=	Standard	52 bytes
<input type="checkbox"/>	Hj1lneGklc5GdcrZhH9lliKrPvU=	Standard	57 bytes
<input type="checkbox"/>	KrNfFrbnAasXCtldQ2vbRD3LiE0=	Standard	69 bytes
<input type="checkbox"/>	NvhbFvt5KWTwhGnuWYXr106qtw=	Standard	5 MB
<input type="checkbox"/>	nX1F2vjxdjGTydQ9nHlzl-hnHPE=	Standard	74.6 KB
<input type="checkbox"/>	nbiq2MOAfdL7aqw40alqIngfpSI=	Standard	39 bytes

Figure 34. Scrambled Blob .git Directory

Ensuring the 16 byte data increase for unencrypted Git compared to GV2 is consistent, the entire contents of each Amazon S3 storage bucket is recorded and analyzed. Amazon S3 stores all files as key-value pairs and every encrypted file is exactly 16 bytes larger than the unencrypted version. This is due to the data overhead of GCM in having an authentication tag. The largest of the files stored is the packfile, in which the Git content data objects are stored. The packfile is generated during Git Garbage Collection and the size growth of this binary file is the key to efficient Git encryption. A closer look at the packfile is shown in Figure 35:

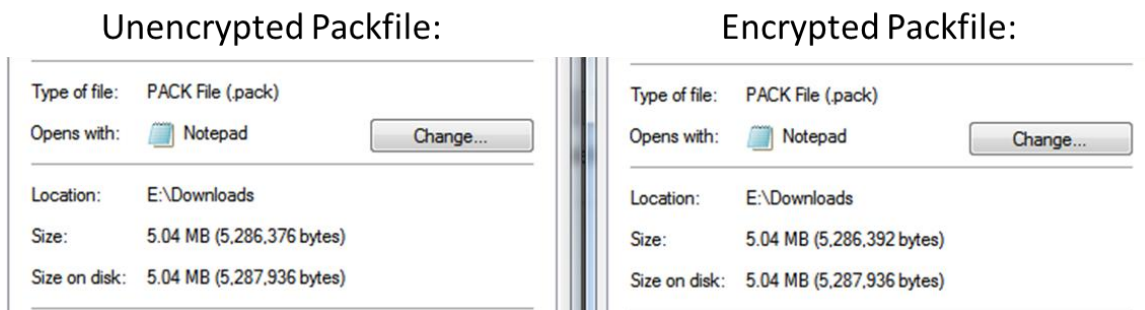


Figure 35. Encrypted and Unencrypted Packfile Comparison

These packfiles are both roughly 5 MB in size. As expected from the previous results, the encrypted packfile is just slightly larger than the unencrypted, because of the appended 16 byte authentication tag. The filename, unencrypted size, encrypted size, and difference for all of the files stored on Amazon S3 for this particular Git repository are summarized in Table 17 (note that not all of the files shown in a potential Git repository are included, just what is needed for this particular repository to be stored on Amazon S3):

Table 17. GV2 Git Program Size Comparison

File:	Unencrypted Size:	Encrypted Size:	Difference	% Difference
Head	23	39	16	41%
Config	36	52	16	31%
Refs	59	75	16	21%
Master Ref	41	57	16	28%
Packs Info	53	69	16	23%
Index File	76364	76380	16	0.02%
Packfile	5286376	5286392	16	0.0003%
Total	5362952	5363064	112	0.002%

The total size increase is made up of solely overhead: 112 bytes in total or 0.002% increase from unencrypted Git. This equates to a 0.002% size increase. Additionally, because garbage collection is run prior to encryption, changes do not grow linearly, the size increase of encrypted repositories maintains a very small overhead of 16 bytes per file. In addition to this, the initialization vectors are also stored on Amazon S3, as well as some other management data. JetS3t library handles this data overhead and maintains it. Other than the IVs, which are 16 bytes each, this is data that is standard to Amazon S3 storage and is also less than a percent of overhead.

There may still be concerns with using JGit in the terms of speed and functionality. C and C++ are generally faster than interpreted languages, such as Java [63-65]. Because Java is converted to bytecode, which is run on the Java Virtual Machine

running on the native operating system, it is inherently slower than native C or C++, although the performance of Java is steadily improving. As with every decision, however, performance tradeoffs should be analyzed prior to making a final decision. JGit provides the same and even extended functionality when compared to Git [48]. JGit also provides several different layers of interaction with a Git repository, from command line to within a program [66]. Most importantly for this research, JGit provides a base program to be modified as specific points within a Git repository to produce GV2. GV2 includes additional libraries and code that is added to stitch them together and increase functionality in the right place in JGit. There are currently no options for hooking into traditional Git to encrypt data after garbage collection. All of the data modifications for traditional Git exist in the form of a filter, as the data is first added to the repository. Still, it is beneficial to analyze the speed differences between JGit and Git as it relates to this research.

In order to determine the performance differences between JGit and Git, as relating to this research, the test methodology of Phase One, Experiment I is used with both JGit and Git. Recall that this test consists of initializing an empty repository, adding all files from a specific program to the repository, and then committing those files to the repository. This process is repeated a total of 10 iterations for an ample sample size to draw statistical conclusions.

The results are written in Table 18 and displayed in the Figures 36-38:

Table 18. Git and JGit Adding Initial Files Time Comparison

Program:	Ave CPU Time:(Std Dev)		
	Git:	JGit	Difference: (% increase)
Linux Kernel	26.2 seconds (0.95)	47.1 seconds (1.56)	20.9 seconds (80%)
Git Program	1.47 seconds (0.189)	3.02 seconds (0.09)	1.55 seconds (105%)
Popping Program	0.085 seconds (0.019)	1.066 (0.04)	0.981 seconds (1154%)

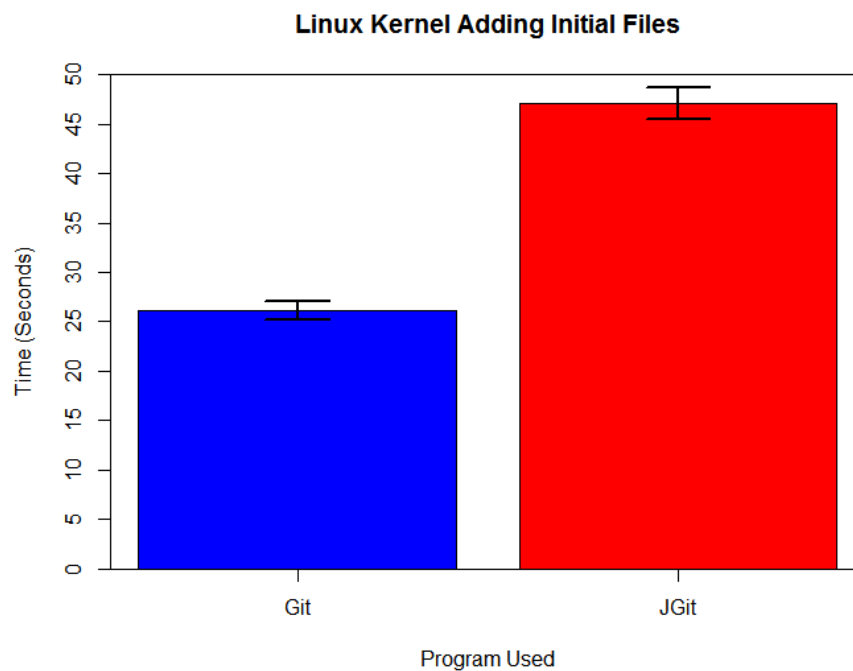


Figure 36. Linux Kernel Git and JGit Adding Initial Files Time Comparison

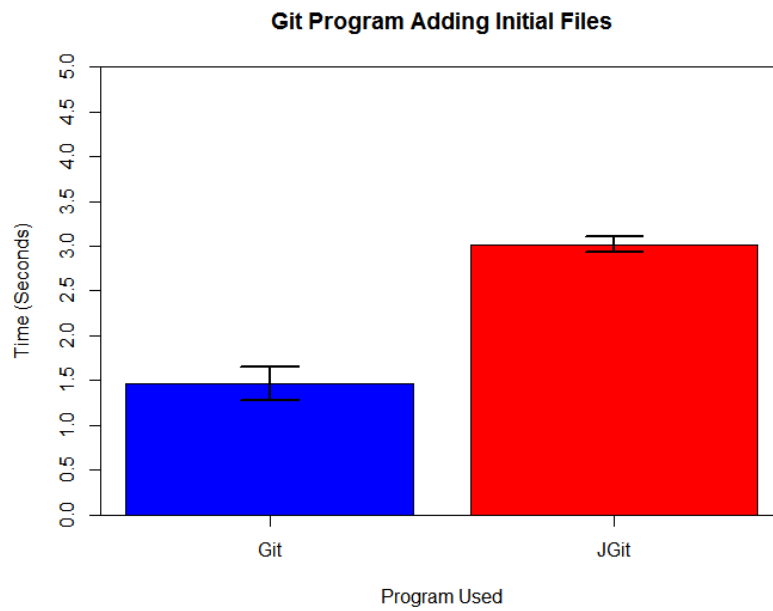


Figure 37. Git Program Git and JGit Adding Initial Files Time Comparison

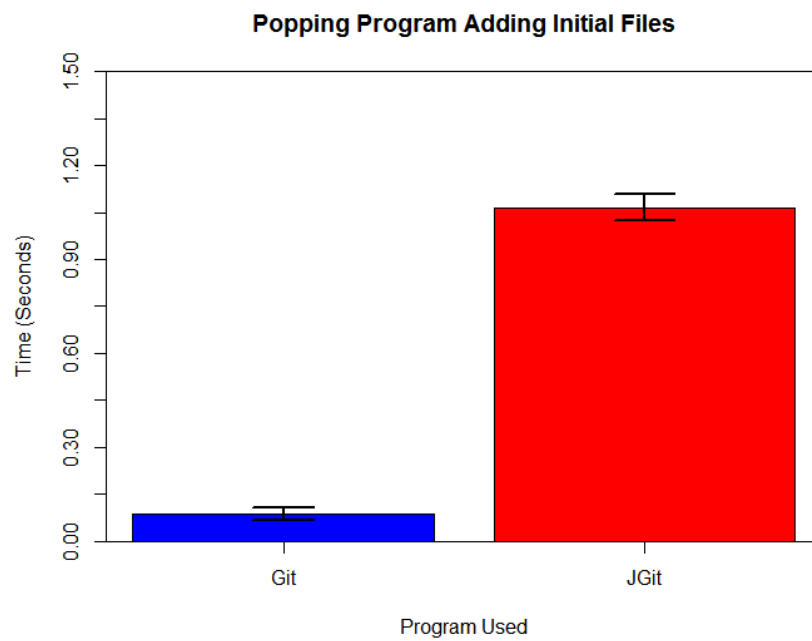


Figure 38. Popping Program Git and JGit Adding Initial Files Time Comparison

JGit takes longer to perform the functions of adding and committing files to a repository when compared to Git. While the percentage increase is high, it is a constant penalty and the time for even a project as large as the Linux Kernel is still less than a minute, which is ideal in most software development environment scenarios. Because JGit is built to emulate Git using the Java language, it operates on traditional Git repository structures [48]. This means that the commands can be intertwined. Thus, if working entirely from the command line, or a program that has interaction with both Git and JGit programs, one can use traditional Git to add and commit files and avoid the time increase in the performance of JGit. When time to encrypt and push to a remote repository in a secure manner, one can use GV2 via the command line interface.

This section presented the results and analysis of Phase Three of this research. Phase Three is built upon the shortcomings of Git encryption implementations that are base lined in Phase One of this research. The design of GV2 takes into account the typical usage of Git, which is researched in Phase Two. Finally, this section presents a new and secure way of encrypting a Git repository by improving upon the open source Java version of Git, JGit. This implementation, named GV2, provides high confidentiality and integrity protection with full functionality and only minimal performance degradation. It does this by adding external libraries and adding Java code functionality into JGit to use GCM AES-CTR mode encryption for confidentiality, processing an authentication code check to ensure integrity protection, and do this all after Git garbage collection in order to manage Git repository size.

GV2 is a vast improvement upon the previously analyzed open source Git encryption implementations: Git-crypt [44] and Git-encrypt [43]. The security qualifications of GV2 and Git-crypt are similar and both in line with the cryptography goals of this thesis research. Performance-wise, GV2 is superior to Git-crypt. Table 19 and Table 20 summarize the performance improvement of GV2 compared to Git-crypt, with the baseline of unencrypted Git, in terms of speed adding initial files to a repository and size of the repository when encrypted:

Table 19. GV2 Speed Performance Compared to Git-crypt In Adding Initial Files

Program	Ave CPU Time:(Std Dev)		
Using Git:	Git:	JGit	Git-crypt:
Linux Kernel	2.62 seconds (0.95)	47.1 seconds (1.56)	380.1 seconds (8.46)
Git Program	1.47 seconds (0.189)	3.02 seconds (0.09)	21.36 seconds (0.456)
Popping Program	0.085 seconds (0.019)	1.066 (0.04)	1.501 seconds (0.506)

Table 20. GV2 Encrypted Repository Size Compared to Git-crypt

Program	Repository Size After Garbage Collection:		
Using Git:	Git:	JGit	Git-crypt:
Git Program	5.2 MB	5.2 MB + 112 Bytes	18.2 MB
Git Program after 5 iterations of file modifications:	7.2 MB	7.2 MB + 112 Bytes	91.1 MB
Performance:	Baseline	Less than 1% larger	1265% larger

V. Conclusions and Recommendations

Chapter Overview

This chapter reviews the research objectives presented in Chapter I and determines if they have been met through the testing methodology and analysis of results of this research. The chapter continues to address the significance of this research to various interested communities. Following this statement, the chapter makes a recommendation for action. Finally, this chapter concludes with recommendations for further research that can be done in this area to better improve using Git in a secure manner.

Conclusions of Research

Recall that the objective of this research is to develop and demonstrate a modification to Git allowing it to serve as a fully functional and secure distributed version control system for sensitive projects. The security goals of this research are to apply confidentiality protection, defined as read-only access protection, and integrity protection, defined as protection against malicious altering of data, to Git repositories. These security goal must be met in a fully functional manner, meaning that all of the Git commands work the same on the secure Git repository as they would a traditional unencrypted Git repository.

This research clearly demonstrates that GV2, developed in Phase Three of this research, overcomes the performance shortfalls of existing Git encryption implementations by encrypting at the lower transport level after Git garbage collection is

run. GV2 accomplishes the security goals described above in a fully functional manner with hardly any negative performance effects compared to unencrypted Git.

Significance of Research

As stated in Chapter I, distributed version control systems, particularly Git, have been rapidly increasing in popularity among software developers in recent years [4]. The problem exists when these organizations have sensitive data that they want to use with Git in an unsecure environment. To secure an environment, especially over the internet, involves high levels of cost. As the name implies, GV2 provides Git with a Virtual Vault in a remote location, such as on an Amazon Cloud, using their Amazon S3 storage service. GV2 provides new documented functionality and performance to the research community. This is new research that has high interest from the Department of Defense and other organizations who want to run applications using a third party cloud service provider but also want to maintain confidentiality and integrity of their application data. In the future, many traditional applications will be modified to support this same type of security in an unsecure environment in an efficient manner that is transparent to the user.

Recommendations for Action

This research suggests that GV2, developed using JGit, is used by those who desire to use Git in an efficient, transparent, and secure manner over a third-party cloud provider. This research specifically implements Git encryption using Amazon S3 storage cloud services, as Amazon provides high performance at affordable prices. This implementation can be modified to work with other preferred providers, if necessary.

While users can create and store their own keys, it is recommended that they use a Public Key Infrastructure (PKI) system for secure transport of keys.

Recommendations for Future Research

This research presents significant improvements over previous Git encryption implementations that use Git clean and smudge filters. The performance of GV2, developed by the author of this thesis, is nearly identical to unencrypted Git, while maintaining high confidentiality and integrity protection via GCM AES-128 counter mode encryption. This research requires the user work with JGit, rather than traditional Git, which may not be ideal for some. Future research would be to implement the same encryption system at the transport layer in the traditional Git program. Additionally, some may require that the Git directory structure is not known and prefer to have a blob of data on Amazon S3. Further research could look into the problem described in Chapter IV of other users not being able to calculate the HMAC references because of not knowing the names of the remote Git files and directory structure. Other relevant research in this area includes using Git as a secure file system. This would allow organizations to securely store files and track changes in an inexpensive and high performance cloud computing environment. Lastly, further research into other secure cloud applications will further the independence on localized systems and allow for increased productivity at the right cost.

Summary

In an effort to allow for users to work with sensitive data with Git repositories, and keep the data safe on an unsecure cloud, this research investigated potential Git encryption implementations. It consisted of three phases:

1. Phase One: baseline current Git encryption performance
2. Phase Two: Characterizing Git repository usage.
3. Phase Three: Improving upon Git by developing a new secure method.

The third phase is successful in developing GV2 and demonstrating it as a secure Git implementation that can be adopted and used by masses. This is demonstrated by sound methodology and analysis of the results. Finally this research concludes with recommendations for action and future research.

Bibliography

References

- [1] Chacon, S., & Hamano, J. C. (2009). *Pro Git*. Springer.
- [2] Chacon, S. (2014). Git-SCM. Retrieved from <http://git-scm.com>.
- [3] Chacon, S. (2014) *Git Community Book*. Retrieved from <http://schacon.github.io/gitbook>.
- [4] Herbsleb, J. D. (2007). Global software engineering: The future of socio-technical coordination. Paper presented at the *2007 Future of Software Engineering*, 188-198.
- [5] Asay, M. (2014). *Git is giving Subversion a run for its money: What took so long?* Retrieved from <http://readwrite.com/2014/01/21/git-subversion-developers>.
- [6] (2014). *Compare Repositories*. Retrieved from <https://www.openhub.net/repositories/compare>.
- [7] Schaller, R. (1997). Moore's law: Past, present and future. *Spectrum, IEEE*, 34(6), 52-59.
- [8] Bell, G. (2008). Bell's law for the birth and death of computer classes: A theory of the computer's evolution. *Solid-State Circuits Society Newsletter, IEEE*, 13(4), 8-19.

- [9] Kurose, J. F., & Ross, K. W. (2001). *Computer networking: A top-down approach featuring the internet*. Addison-Wesley Reading.
- [10] Arutyunov, V. (2012). Cloud computing: Its history of development, modern state, and future considerations. *Scientific and Technical Information Processing*, 39(3), 173-178.
- [11] Mell, P., & Grance, T. (2009). The NIST definition of cloud computing. *National Institute of Standards and Technology*, 53(6), 50.
- [12] (2014). *Definition of Cloud Computing*. Retrieved from <http://www.dictionary.reference.com>.
- [13] Marston, S., Li, Z., Bandyopadhyay, S., Zhang, J., & Ghalsasi, A. (2011). Cloud computing—The business perspective. *Decision Support Systems*, 51(1), 176-189.
- [14] Han, Y. (2011). Cloud computing: Case studies and total cost of ownership. *Information Technology and Libraries*, 30(4), 198-206.
- [15] Bacon, J., Eysers, D., Pasquier, T. M., Singh, J., Papagiannis, I., & Pietzuch, P. (2014). Information flow control for secure cloud computing. *Network and Service Management, IEEE Transactions on*, 11(1), 76-89.
- [16] Zhang, H., Jiang, G., Yoshihira, K., & Chen, H. (2014). Proactive Workload Management in Hybrid Cloud Computing. *Network and Service Management, IEEE Transactions on*, 11(1), 90-100.

- [17] Satyanarayanan, M., Kistler, J. J., Kumar, P., Okasaki, M. E., Siegel, E. H., & Steere, D. C. (1990). Coda: A highly available file system for a distributed workstation environment. *Computers, IEEE Transactions on*, 39(4), 447-459.
- [18] Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N., & West, M. J. (1988). Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1), 51-81.
- [19] Shvachko, K., Kuang, H., Radia, S., & Chansler, R. (2010, May). The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on* (pp. 1-10). IEEE.
- [20] Ghemawat, S., Gobioff, H., & Leung, S. T. (2003, October). The Google file system. In *ACM SIGOPS operating systems review* (Vol. 37, No. 5, pp. 29-43). ACM.
- [21] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., & Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2), 4.
- [22] Dean, J., & Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1), 107-113.
- [23]. Carlsson, E. (2013). Mining Git Repositories: An introduction to repository mining (Bachelor's Thesis). Linnaeus University, Kalmar, Sweden. Retrieved from <http://www.diva-portal.org/smash/get/diva2:638844/FULLTEXT01.pdf>

- [24] Feng, D. G., Zhang, M., Zhang, Y., & Xu, Z. (2011). Study on cloud computing security. *Journal of Software*, 22(1), 71-83.
- [25] Bisong, A., & Rahman, M. (2011). An overview of the security concerns in enterprise cloud computing. *arXiv preprint arXiv:1101.5613*.
- [26] Paar, C., & Pelzl, J. (2009). *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media.
- [27] Schneier, B. (2007). *Applied cryptography: protocols, algorithms, and source code in C*. John Wiley & Sons.
- [28] Boneh, D. (2014). *Introduction to Cryptography Class*. Retrieved from <https://class.coursera.org/crypto-preview/lecture>.
- [29] Rogaway, P., & Shrimpton, T. (2006). A provable-security treatment of the key-wrap problem. In *Advances in Cryptology-EUROCRYPT 2006* (pp. 373-390). Springer Berlin Heidelberg.
- [30] Harkins, D. (2008). RFC-5297 Synthetic Initialization Vector (SIV) Authenticated Encryption Using the Advanced Encryption Standard (AES). *Network Working Group*.
- [31] McGrew, D., & Viega, J. (2004). The Galois/counter mode of operation (GCM). *Submission to NIST*. <http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/gcm/gcm-spec.pdf>.

- [32] Ruparelia, N. B. (2010). The history of version control. *ACM SIGSOFT Software Engineering Notes*, 35(1), 5-9.
- [33] De Alwis, B., & Sillito, J. (2009, May). Why are software projects moving from centralized to decentralized version control systems? *Cooperative and Human Aspects on Software Engineering, 2009. CHASE'09. ICSE Workshop on* (pp. 36-39). IEEE.
- [34] Spinellis, D. (2012). Git. *Software, IEEE*, 29(3), 100-101.
- [35] Wakchaure, S. S., & Arora, S. K. (2012). Implementation of a Secure Distributed Storage System. (Unpublished Graduate Project). George Mason University, Fairfax, VA. Retrieved from http://ece.gmu.edu/coursewebpages/ECE/ECE646/F12/project/F12_specifications/Snehil_Simrit.pdf
- [36] Wang, C., Wang, Q., Ren, K., Cao, N., & Lou, W. (2012). Toward secure and dependable storage services in cloud computing. *Services Computing, IEEE Transactions on*, 5(2), 220-232.
- [37] Varadharajan, V., & Tupakula, U. (2014). Security as a service model for cloud environment. *Network and Service Management, IEEE Transactions on*, 11(1), 60-75.
- [38] Lührig, J. P. File synchronization using git-annex assistant. University of Lubeck, Lubeck, Germany. Retrieved from http://media.itm.uni-luebeck.de/teaching/ws2013/sem-cloud-computing/File_synchronization_using_git-annex_assistant.pdf

- [39] (2014). *Git Annex*. Retrieved from <http://Git-annex.branchable.com>.
- [40] Klingelhuber, P., & Mayrhofer, R. (2011, December). Private notes: encrypted XML notes synchronization and sharing with untrusted web services. *Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services* (pp. 254-261). ACM.
- [41] (2014). *Git transparent encryption*. Retrieved from http://syncom.appspot.com/papers/git_encryption.txt.
- [42] Robinson, M., Niu, J., & Shonle, M. (2011, November). GitBAC: Flexible access control for non-modular concerns. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering* (pp. 500-503). IEEE Computer Society.
- [43] Gilk, W. (2014). *Git-encrypt*. Retrieved from <https://Github.com/shadowhand/Git-encrypt>
- [44] Ayer, A. (2014). *Git-crypt*. Retrieved from <https://Github.com/AGWA/Git-crypt>
- [45] (2014). *Git-remote-gcrypt*. Retrieved from <https://github.com/joeyh/git-remote-gcrypt>.
- [46] Schneier, B. (2014). *Insecurities in Linux /dev/random*. Retrieved from https://www.schneier.com/blog/archives/2013/10/insecurities_in.html.

- [47] Hamano, J. (2014) *RE: Transparently encrypt repository contents with GPG*. Retrieved from <http://article.gmane.org/gmane.comp.version-control.git/113221>
- [48] (2014). *JGit Project*. Retrieved from <http://www.eclipse.org/jgit>.
- [49] (2014). *libgit2*. Retrieved from <https://libgit2.github.com>.
- [50] Weber, S., Meyer, B., Nordio, M., & Estler, H. C. (2012). *Automatic Version Control System for Distributed Software Development* (Doctoral dissertation, Eidgenössische Technische Hochschule Zürich, Department of Computer Science, Software Engineering Group).
- [51] Hattori, L. P., & Lanza, M. (2008, September). On the nature of commits. In *Automated Software Engineering-Workshops, 2008. ASE Workshops 2008. 23rd IEEE/ACM International Conference on* (pp. 63-71). IEEE.
- [52] Alali, A., Kagdi, H., & Maletic, J. I. (2008, June). What's a typical commit? A characterization of open source software repositories. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*(pp. 182-191). IEEE.
- [53] Kolassa, C., Riehle, D., & Salim, M. A. (2013, August). The empirical commit frequency distribution of open source projects. In *Proceedings of the 9th International Symposium on Open Collaboration* (p. 18). ACM.

- [54] Kolassa, C., Riehle, D., & Salim, M. A. (2013). A model of the commit size distribution of open source. In *SOFSEM 2013: Theory and Practice of Computer Science* (pp. 52-66). Springer Berlin Heidelberg.
- [55] Hindle, A., German, D. M., & Holt, R. (2008, May). What do large commits tell us?: a taxonomical study of large commits. In *Proceedings of the 2008 international working conference on Mining software repositories* (pp. 99-108). ACM.
- [56] Mockus, A. (2009, May). Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on* (pp. 11-20). IEEE.
- [57] Gousios, G., & Spinellis, D. (2012, June). GHTorrent: Github's data from a firehose. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on* (pp. 12-21). IEEE.
- [58] (2014). *GitHub Archive*. Retrieved from <http://www.githubarchive.org>.
- [59] (2014). *GitHub API Overview*. Retrieved from <https://developer.github.com/v3>.
- [60] Schneider, A. (2015). *Popping*. Retrieved from <https://github.com/schneiderandre/popping>.
- [61] (2014). *Google BigQuery*. Retrieved from <http://cloud.google.com/bigquery>.

- [62] (2014). *CURL*. Retrieved from <http://curl.haxx.se>.
- [63] Gherardi, L., Brugali, D., & Comotti, D. (2012). A java vs. c++ performance evaluation: a 3d modeling benchmark. In *Simulation, Modeling, and Programming for Autonomous Robots* (pp. 161-172). Springer Berlin Heidelberg.
- [64] Georges, A., Buytaert, D., & Eeckhout, L. (2007). Statistically rigorous java performance evaluation. *ACM SIGPLAN Notices*, 42(10), 57-76.
- [65] Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., & Wiedermann, B. (2006, October). The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM SIGPLAN Notices* (Vol. 41, No. 10, pp. 169-190). ACM.
- [66] Blewitt, A. (2013). *Embedding JGit*. Retrieved from <http://alblue.bandlem.com/2013/11/embedding-jgit.html>.
- [67] (2014). *JetS3t*. Retrieved from <http://www.jets3t.org>.
- [68] (2014). *Bouncy Castle*. Retrieved from <https://www.bouncycastle.org/java.html>.
- [69] (2014). *Oracle Java Documentation of SecureRandom Class*. Retrieved from <https://docs.oracle.com/javase/7/docs/api/java/security/SecureRandom.html>.
- [70] FIPS, P. (2001). 140-2: Security requirements for cryptographic modules. *National Institute of Standards and Technology*, 15.

- [71] Eastlake, D. E., Crocker, S. D., & Schiller, J. E. F. F. R. E. Y. (1994). RFC-1750 Randomness Recommendations for Security. *Network Working Group*.
- [72] (2014). *The Git Guys: The .git Directory Explained*. Retrieved from <http://www.gitguys.com/topics/the-git-directory>.
- [73] Teat, C., & Peltsverger, S. (2011, March). The security of cryptographic hashes. In *Proceedings of the 49th Annual Southeast Regional Conference* (pp. 103-108). ACM.

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 074-0188	
<p>The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of the collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.</p> <p>PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.</p>					
1. REPORT DATE (DD-MM-YYYY) 26-03-2015		2. REPORT TYPE Master's Thesis		3. DATES COVERED (From – To) August 2013 – March 2015	
TITLE AND SUBTITLE Git As An Encrypted Distributed Version Control System				5a. CONTRACT NUMBER	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S) Shirey, Russell G., Captain, USAF				5d. PROJECT NUMBER	
				5e. TASK NUMBER	
				5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way, Building 640 WPAFB OH 45433-8865				8. PERFORMING ORGANIZATION REPORT NUMBER AFIT-ENG-MS-15-M-022	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Intentionally left blank				10. SPONSOR/MONITOR'S ACRONYM(S)	
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION/AVAILABILITY STATEMENT DISTRIBUTION STATEMENT A. APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.					
13. SUPPLEMENTARY NOTES This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.					
14. ABSTRACT This thesis develops and presents a secure Git implementation, Git Virtual Vault (GV2), for users of Git to work on sensitive projects with repositories located in unsecure distributed environments, such as in cloud computing. This scenario is common within the Department of Defense, as much work is of a sensitive nature. In order to provide security to Git, additional functionality is added for confidentiality and integrity protection. This thesis examines existing Git encryption implementations and baselines their performance compared to unencrypted Git. Real-world Git repositories are examined to characterize typical Git usage and determine if the existing Git encryption implementations are capable of efficient performance with regards to typical Git usage. This research shows that the existing Git encryption implementations do not provide efficient performance. This research develops an improved secure Git implementation, GV2, with transparent authenticated encryption. The fundamental contribution of this research is developing GV2 to perform Git garbage collection on plaintext data before encrypting the data. The result is a secure Git implementation that is transparent to the user with only a minor performance penalty, compared to unencrypted Git.					
15. SUBJECT TERMS					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT UU	18. NUMBER OF PAGES 126	19a. NAME OF RESPONSIBLE PERSON Dr. Kenneth Hopkinson, AFIT/ENG
a. REPORT U	b. ABSTRACT U	c. THIS PAGE U			19b. TELEPHONE NUMBER (Include area code) (937) 255-3636, ext 4579 (Kenneth.hopkinson@afit.edu)

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18