Eötvös Loránd University
Faculty of Informatics

Investigations on Securing Repositories using Difference Analysis and Patch Applications on Encrypted Files

Seres István András, M.Sc.
Eötvös Loránd University

Tom Mathijs van Essen
Computer Science

Prof.Dr. Andreas Peter
University of Twente

Dr. Serge Autexier
DFKI

Budapest, 2021.

Statement

of thesis submission and originality

I hereby confirm the submission of the Master Thesis Work on the Computer Science MSc course with author and title:

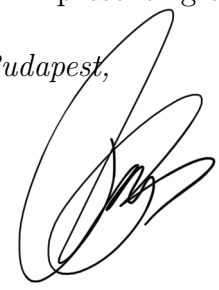
Name of Student:	Tom Mathijs van Essen
Code of Student:	aiyf2k
Title of Thesis:	Investigations on Securing Repositories using Difference Analysis and Patch Applications on Encrypted Files.
Supervisor:	István András Seres

at Eötvös Loránd University, Faculty of Informatics.

In consciousness of my full legal and disciplinary responsibility I hereby claim that the submitted thesis work is my own original intellectual product, the use of referenced literature is done according to the general rules of copyright.
I understand that in the case of thesis works the following acts are considered plagiarism:

- literal quotation without quotation marks and reference;
- citation of content without reference;
- presenting other's published thoughts as own thoughts.

Budapest,



Abstract

Collaborating on code projects is popular and many corporations rely on this for their daily process. Commonly, sharing source code in such a manner is done via repositories that are part of a Version Control System (VCS). Such a system can be the target of an attack or can leak the code in another way. This work investigates the tools needed to create a secure repository.

To enable a secure repository this work introduces the notion of securely composing patch files – sCompose – and proposes approaches to the problem. We introduce the Secure Longest Common Subsequence (LCS) functionality – sLCS – and suggest functions to solve the problem. These approaches to sLCS are not practical in a real-world scenario due to the space complexity. Finally, we introduce the notion of secure difference analysis with the function sDiff and sDiff3.

Acknowledgement

I owe a debt of gratitude to my supervisors, Andreas Peter, Serge Autexier, and István Seres. They provided me with fresh ideas and encouraged me to tackle the challenges that we crossed along the way. When stress threatened to take the best of me, our meetings – or a linked tweet – put my mind at ease. Our discussions helped me gain insight into the world of research and pushed me to keep moving forwards. In particular, I am grateful for the exciting assignment made possible by Andreas and Serge.

Through the past half-year, the world was full of uncertainties. For me, this meant I did not know in which country I would spend the final months of my Master. While this had its aggravating moments, these were fleeting. For this, I have all my friends to thank. Thank you to all my friends in Budapest and the Netherlands for making me feel that where ever I ended up, I would be welcome.

In my crusade to finish my thesis, I appreciated the interest of the members of Dispuut Yorinf and Huize Boslust. In particular, I acknowledge the help and support I received from Vincent Dunning and Mark 't Hart. Discussions with you kept me motivated and helped me look at what I could do instead of what I could not do.

Esther, Peter – mom, dad – Stef; thank you! Not only for the past six months, thank you for the past 23 years and a bit. Sometimes you were a challenge to deal with; most of the time, I was a hassle to manage. Thank you for sticking it out with me. Hopefully, we will have many more years to go!

Finally, I thank all the people I have forgotten to mention by name, my apologies. If you made me smile, distracted me, or inspired me over the last half-year, thank you.

Contents

List of Figures

List of Tables

Acronyms

BBF Basic Block Function. xxii

CVS Concurrent Versions System. xi

FHE Fully Homomorphic Encryption. xvii, xxviii, lix

GCD Greatest Common Divisor. lv

LCS Longest Common Subsequence. i, vii, xi, xii, xv, xvi, xvii, xix, xx, xxiv, xxv, xxvii, xxviii, xxix, xxx, xlix, l, li, liv, lv, lvi, lvii, lviii, lix, lx, lxi, lxiii, lxv, lxvii

LD Levenshtein Distance. xi

OS Operating System. xi

PHE Partial Homomorphic Encryption. xxviii

PKEET Public Key Encryption with Equality Test. lv

SCCS Source Code Control System. xi

VCS Version Control System. i, vi, x, xi, xii, xiii, xxxi, lxv

Chapter 1

Introduction

Traditional repositories allow multiple users to work together on a projects such as source code. These repositories are part of a Version Control System (VCS). These systems consist of patches that describe the changes from one file to the next. The patches are part of commitments. Commitments contain meta-data about the author, a textual description of the changes in the file, and the actual patch. We create such a patch using difference analysis. A property of these patches is that we can compose them. Version Control Systems use this property to squash commitments. This is a feature that helps maintain a repository and its readability. Additionally, we can use difference analysis to combine multiple files into one file. We refer to this process as merging documents, this feature is an important part of Version Control Systems.

Difference analysis of documents is a problem that presented itself during the early development of computer languages and systems. Its applications include comparing lengthy outputs of functions, verifying checksums, and finding the changes from one version of a document to the next. The latter functionality is important in our application. Not only do we wish to determine the difference between two files – this is useful in software updating through patches – there is also a need to combine multiple files. Combining files is often referred to as merging, and it derives its capabilities from difference analysis.

Throughout recent history, many tools for on difference analysis appeared[1][2][3].

[1]FileMerge: `https://developer.apple.com/xcode/features/`
[2]TkDiff: `https://www.unix.com/man-page/linux/1/tkdiff/`
[3]Kompare: `https://apps.kde.org/nl/kompare/`

These tools all serve the same purpose, that of difference analysis between two documents. This thesis follows the work of Autexier [Aut15]. This work introduces a new aspect to difference analysis – namely, similarity instead of equality. The difference analysis in the methods of Autexier resembles the difference analysis as performed by the `diff` function found in Unix Operating System (OS), introduced in the 5^{th} version of said OS. The work of Hunt and McIlroy forms the basis of this function [HM75].

Using the `diff` function, we can create descriptions from one file to another. We can perform the transformation based on a description automatically. We call such a description a patch. Storing these patches leads to a system that keeps track of each iteration of a piece of software. This history of revisions is aptly named the Source Code Control System (SCCS). This leads to VCS such as Concurrent Versions System (CVS), BitKeeper, and Git [FSF08, Bon15, CS14].

The patch created by `diff` contains the deleted lines from the first file, the lines added to the second file, and the changed lines from the first file to the second. Formatting these three different changes in a certain way leaves us with an edit-script. An edit-script describes the steps we need to take to edit one file to match another file. Only registering the changes leaves us with smaller files to transmit compared to sending the whole new file.

The alignment of two documents forms the basis of difference analysis. This alignment allows us to pinpoint the differences between two files efficiently. The alignment method is much like the Levenshtein Distance (LD) [Lev66]. The LD gives the number of deletion, changes, and insertion to transform one string into another. The difference between the LD and the alignment method is that the alignment method does not consider changes of elements, only additions and deletions. This translates to LD by setting the cost for changing higher than the summed cost of an addition and a deletion. This cost division leaves us with a scenario in which the algorithm will never pick changing a character as it is cheaper to delete the old element and insert the new one. This method almost directly resembles the Longest Common Subsequence (LCS) problem. Solving the LCS problem leaves us with one or more sequences present in both the input sequences. These sequences are the most extended sequence present in both the input sequences. Note that there is a difference between substrings and subsequences. Substrings require elements to be subsequent in the sequence. In the case of the LCS, only the relative ordering is

important, not the direct relation.

Once we have the Longest Common Subsequence between two sequences, we can construct a `diff` output. First, we determine which file is the origin file and which file is the target file. The deletions are those values present in the origin file and not in the LCS. The insertions are the elements in the target file but not in the LCS. The retentions are the lines that make up the LCS between the origin and the target. We observe that, given the origin file, we can derive the deletions from the retentions and vice versa. Considering the retentions instead of the omissions allows us to compose patch files. This feature is essential in squashing commitments.

Besides the `diff` function we also introduce the `diff3` function. `diff3` also considers, besides the two input files, a common origin file. This common origin file is the origin for both the other input files. An origin file of a file is the previous version of a file. Sharing an origin file means that two files, a and b, are adaptations of the same document o.

The output of the `diff3` function is the edit-script that describes a transformation of o. We transform o to include the changes in a and b. Note that this function can result in conflicts. If a and b change the same value, we can no longer determine which change is the correct one without making assumptions about the underlying files. The uncertainty about the correct course of action results in a merge conflict. Merge conflicts can present themselves in a repository and require human intervention to solve.

We investigate how we can translate these functions to their secure counter part. We introduce the notion of Secure `diff`, `diff3`, and LCS – sDiff, sDiff3, sLCS – that are functions over files. Furthermore, we introduce Secure Compose – sCompose – a function over patch files. These functionalities combined enable a repository to be secure. We deem a repository secure if it can perform all the tasks of a normal repository without learning about the content of the repository.

1.1 Motivation

Modern applications of difference analysis mostly find their merit in Version Control Systems. Protecting the source code of a project can be a vital aspect of a business, whether to enforce their Copyright or to slow down competitors in developing competing products. However, large corporations often have multiple developers working

on the same source code. Such a scenario lends itself to VCS perfectly. There are many options for such a system; all can have drawbacks.

GitHub allows its customers to hide its repositories from the public for a relatively low price[4]. However, this comes at the cost of the necessity to trust GitHub and, by extensions, all its parent or sibling corporations. Other solutions exist where a company does not necessarily need to trust the provider of the VCS software. GitLab offers hosting plans, but one can also host an instance of the software developed by GitLab on their server[5]. Since GitLab is open source, a company can theoretically trust that its code is safe. However, this requires the corporation to have enough knowledge, time, and resources to host and maintain their server. Hosting a server can get costly.

History knows examples of source code leaking unintentionally [Cim20, War17]. Hackers can also target a server to obtain source code, as happened to CD Projekt Red in February of 2021. This targeted attack left the attackers with the source code of popular video games Witcher 3 and Cyberpunk 2077[Sta21]. In January of the same year, Nissan found themselves the victim of source code leaking [Cim21]. However, this was not a targeted attack. It was a misconfigured Git server. By using a standard username and password, their repositories were open to the public. These situations illustrate how we can gain security by protecting centrally stored data.

1.2 Research Questions and Objective

This thesis investigates how we can create technologies that allow the existence of secure repositories. These technologies include the creation – using `diff` – and composition of patches, and the merging of files using `diff3`. We investigate the applicability of cryptographic techniques in the problems of automatic difference analysis, and storage and composition of patch files. We say that a repository is secure if the repository does not learn anything about the data it is hosting while still performing the tasks we require from a regular VCS.

We specifically look at the creation of patch files using `diff` function inspired by the work of Autexier [Aut15] and the composition of created patch functions.

[4] `https://github.com/pricing`

[5] Prices: `https://about.gitlab.com/pricing/` and Self-Hosting: `https://about.gitlab.com/install/`

We define three research questions we investigate in this thesis. The first is:

> *RQ1. How can we secure a repository using difference analysis and patch composition over encrypted documents?*

We split this research question into two subquestion that specify the techniques we wish to investigate. We discuss these two subquestion individually below. The first subquestion is:

> *RQ1a. How can we compose patch files constructed over encrypted documents?*

This first subquestion concentrates on the ability to develop a function that can compose patch files in a secure repository. This means that the server hosting the repository does not learn the content or nature of the two patches we compose or of the resulting patch. We investigate this question in Chapter 3.

The second subquestion to *RQ1.* is:

> *RQ1b. How can we perform difference analysis, including Diff and Three-Way Diff, over encrypted files?*

This question focuses on the functionalities as posed by Autexier in [Aut15]. We investigate how to obtain these functionalities between encrypted documents. The objective here is to keep the information held by individual parties hidden from other participants. We investigate the underlying alignment of sequences used in these functions in Chapter 4 and discuss the functionalities themselves in Chapter 5.

We pose two research questions to evaluate the performances of the procedures proposed under the first research question. These two questions test the feasibility of the methods developed under *RQ1.*

> *RQ2. What are the computational and communicational complexities of the algorithms developed under RQ1?*

This question helps provide insight in the theoretical bounds of the suggested functions. This will help us determine whether it is feasible to use the propose method in practice.

RQ3. What are the runtimes of the under RQ1 developed algorithms in realistic scenarios?

We test the applicability of the proposed methods. We then translate this to a realistic setting and evaluate whether the proposed method can offer the desired functionality.

The chapters that describe the answers to *RQ1a* and *RQ1b* also answer *RQ2* and *RQ3* for their respective function.

1.3 Contributions

We introduce the concept of Secure Compose – sCompose – for patches and suggest approaches that reveal minimal to no information. The former results in effective patch files. The latter does not. An effective patch file is a patch file that merits its use. The use of a patch file is warranted if it is more practical to send a patch file than to send the target file of the patch.

We introduce the concept of Secure Longest Common Subsequence – sLCS. sLCS is a function that takes two hidden sequences and determines the LCS between the two sequences. We suggest an interactive approach where the party executing the function and the receiver of the result communicate. Furthermore, we suggest non-interactive approaches that let the executor of the function determine the result on its own. These approaches are purely theoretical as the complexity does not allow them to be practical in real-world scenarios.

Finally, we introduce the notion of Secure Difference Analysis for the functions `diff` and `diff3`. These functions rely on sequence alignment. This sequence alignment is the LCS. Since we do not propose a feasible sLCS function, we do not propose methods for sDiff and sDiff3. We only introduce both concepts and their desired functionality.

1.4 Structure

The remainder of this thesis is structured as follows. We start by providing background and preliminary information in Chapter 2. We continue by describing and introducing Secure Patch Composition in Chapter 3. Chapter 4 introduces and anal-

yses the Secure Longest Common Subsequence function. We discuss the Secure `diff` and `diff3` and their feasibility with regards to a secure repository in Chapter 5. Finally, we conclude and discuss future work in Chapter 6.

Chapter 2

Background and Preliminaries

This chapter describes background knowledge and provides definitions and notations we use throughout this thesis. We introduce definitions of files and patch files. These are important when we discuss creating and composing commitments in a secure repository setting. Furthermore, we introduce the concept of sequences and the Longest Common Subsequence (LCS). The LCS aligns two sequences. The alignment is essential in the `diff` and `diff3` functions we introduce in this chapter. To create secure repositories, we introduce cryptographic notation and highlight Fully Homomorphic Encryption (FHE) as a tool. Additionally, we describe our security setting in a general sense. Finally, we conclude by providing an overview of all notation for convenience.

2.1 Sequence

A sequence is an indexed set of elements from a specific domain. We use sequences to define files. This is important since we will require alignment between files for the `diff` and `diff3` functions. Formally, we use the definition for a sequence as provided by Autexier [Aut15].

Definition 2.1.1 (Sequence [Aut15, p. 4])**.** Let I be a set of positive natural numbers and s_i, $i \in I$ be elements of a given domain D. Then $s := (s_i)_{i \in I}$ denotes the *sequence* $s_{i_1} \dots s_{i_n}$ such that $|I| = n$ and for all $1 \leq k < l \leq n$ holds $i_k < i_l$. The length of the sequence is the cardinality of I. The set of sequences is $\mathbb{S}$.

A sequence has a length. We use cardinality notation to indicate the length of a

sequence. For sequence s, $|s|$ denotes the length of s.

We can perform operations on a sequence. Let D be the domain of sequence s, where the index set of s is I_s. For $e \in D$, $s \,||\, e$ denotes the concatenation of the element e to sequence s. The resulting sequence is $r := s_{i_1} \dots s_{i_n} e$. The new sequence r has the associated index set I_r. The index of the new element e is greater than any index present in I_s. This is the case since we concatenate an element to the end of s thus the index cannot be smaller than any other element by definition. For the index of e in I_r it holds that $i_e > \max I_s$. For sequences where we maintain a consecutive index set, $I_s = \{1, \dots, |s|\}$, the index of e, $i_e \in I_r$, is $|s| + 1$.

Given a sequence, we wish to determine the last element of the sequence. This last element in a sequence s is the element with the greatest index in the associated index set. Thus $\text{Last}(s) = s_{\max I_s}$ this holds for $I_s \neq \emptyset$. If $I_s = \emptyset$ the sequence is empty and there exists no last element. Besides the last element we also define the operation for the initial elements. The initial elements are all elements in a sequence besides the last element. Thus, $\text{Init}(s) = (s_i)_{i \in I_s \setminus \{\max I_s\}}$. Finally, as an operation we define the predecessor of an index. The largest predecessor of element $i \in I_s$ is the largest number $i' \in I_s$ such that $i' < i$. We note this as $\text{Pred}_{I_s}(i) = i'$ if such a value exists. Note that in the case of consecutive index sets, $\{1, \dots, n\}$ for some $n > 1$ the predecessor of i is $i - 1$ for all $1 < i \leq n$. The smallest element in the index set, $\min I_s$, does not have a predecessor.

A subsequence, r, of a sequence, s is a sequence such that we can derive r from s by deleting some or none of the elements from s. Note that the ordering of the elements is essential. That is, the elements of the subsequence have the same relative order as the original sequence. However, it does not matter if elements in either sequence are subsequent. This means that for $i, i' \in I_r$, note that by definition $i, i' \in I_s$, $\text{Pred}_{I_r}(i) = i'$ does not imply $\text{Pred}_{I_s}(i) = i'$ and $\text{Pred}_{I_s}(i) = i'$ does not imply $\text{Pred}_{I_r}(i) = i'$. We note that r is a subsequence of s using the subset notation, $r \subseteq s$.

Substrings are subsequences where the elements of the subsequence are subsequent. A substring for the sequence s from index $m \in I_s$ till $n \in I_s$ is the sequence r such that for $r := (s_i)_{i \in \{\, x \mid x \in I_s \text{ and } m \leq x \leq n \,\}}$. We denote this as $s^{m,n}$. A prefix is a substring starting at the minimal value of I_s. As a shorthand for $s^{\min(I_s),n}$ we use s^n.

2.2 Longest Common Subsequence

Autexier uses the Longest Common Subsequence (LCS) to align two sequences. The LCS function with two input sequences gives the most extended sequences present in both the input sequences. Thus we can have multiple LCS sequences. We determine candidate LCS sequences based on the LCS present in the initial elements of both input sequences. $\max_{\mathsf{len}}$ determines the longest sequences of the input sequences and outputs these sequences. If two candidate LCS sequences are of equal length, both remain candidates. More formally, we have Definition 2.2.1.

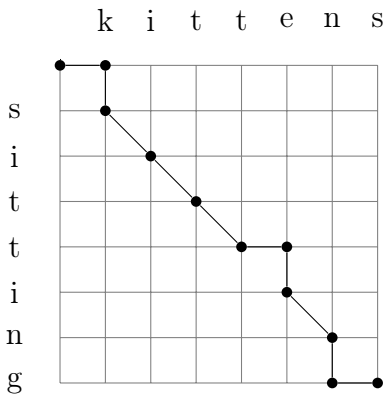
Definition 2.2.1 (Longest Common subsequence (LCS))**.** For two sequences, given as $s := (s_i)_{i \in I_s}$ and $r := (r_j)_{j \in I_r}$, there are one or more Longest Common Subsequences. These are given as:

$$\mathrm{LCS}\,(s,r) = \begin{cases} \emptyset & \text{if } I_s = \emptyset \text{ or } I_r = \emptyset \\ \mathrm{LCS}(\mathrm{Init}(s), \mathrm{Init}(r)) \mid\mid \mathrm{Last}(s) & \text{if } \mathrm{Last}(s) = \mathrm{Last}(r) \\ \max_{\mathsf{len}} \begin{cases} \mathrm{LCS}(s, \mathrm{Init}(r)) \\ \mathrm{LCS}(\mathrm{Init}(s), r) \end{cases} & \text{if } \mathrm{Last}(s) \neq \mathrm{Last}(r) \end{cases}$$

The output of the LCS is a sequence. Note that in the case of $|\mathrm{LCS}(s, \mathrm{Init}(r))| = |\mathrm{LCS}(\mathrm{Init}(s), r)|$ we retain both sequences as candidates. This means that we continue the procedure with two possible LCSs.

The LCS problem has an optimal substructure. A problem with an optimal substructure can find its optimal solution by solving its subproblems. This property helps determine if a dynamic programming approach is helpful to solve the problem. Since the LCS problem has such an optimal substructure, we see that an approach to solving the problem is dynamic programming. In this approach, we construct a dynamic programming table where one sequence aligns with the columns and the other sequence with the rows of the table. Using this layout, we can fill the table by following the definition of the LCS.

We can view the LCS as a path over the dynamic programming table. The route can take three possible steps from one table entry to the next. It can take a diagonal action, a horizontal step, or a vertical step. The last two steps occur if the two input sequence elements associated with the current table entry are not equal. The third line of Definition 2.2.1 depicts these steps. The diagonal step is the case where the



		s						
r		k	i	t	t	e	n	s
	0	0	0	0	0	0	0	0
s	0	0	0	0	0	0	0	1
i	0	0	1	1	1	1	1	1
t	0	0	1	2	2	2	2	2
t	0	0	1	2	3	3	3	3
i	0	0	1	2	3	4	3	3
n	0	0	1	2	3	3	4	4
g	0	0	1	2	3	3	4	4

Figure 2.1: Path depiction, and table showing the length, of LCS(*kitten*, *sittin*). We highlight the path we take in the table of the intermediary lengths.

elements are identical, the second line of Definition 2.2.1.

With this pathing we can derive the LCS based on a table of the lengths of the intermediary LCS values. An example of this is given in Figure 2.1. This figure depicts the path and a table containing the length of the LCS.

Under certain conditions, we can improve the complexity of the LCS procedure. Given a fixed alphabet Σ, we can precompute blocks of the LCS. This result in a dynamic programming table of size $n \cdot \log n$ instead of n^n. Masek and Patterson describe this approach where they use the Four Russian Technique.

2.2.1 LCS using "Four Russians"

The methodology proposed by Masek and Patterson [MP80] adopts the dynamic programming variant to include a pre-computed look-up table based on the Four Russian technique. We use the same notation and process as [FGM09] to address this approach. Masek and Patterson create t-sized blocks that contain all necessary information. Figure 2.2 depicts the normal dynamic approach and the four Russians approach. We observe that the blocks allow us to eliminate entries from the table. Commonly, for input of size n we choose $t = \log_2 n$.

The dynamic programming table based on sequence s indexed by $i \in I_s$ and sequence r indexed by $j \in I_r$ is given by L. We denote the boundaries of the window

		s					
r		k	i	t	t	e	n
	0	0	0	0	0	0	0
s	0	0	0	0	0	0	0
i	0	0	1	1	1	1	1
t	0	0	1	2	2	2	2
t	0	0	1	2	3	3	3
i	0	0	1	2	3	3	3
n	0	0	1	2	3	3	4

(a)

		s					
r		k	i	t	t	e	n
	0	0	0	0	0	0	0
s	0			0			0
i	0			1			1
t	0	0	1	2	2	2	2
t	0			2			3
i	0			2			3
n	0	0	1	2	3	3	4

(b)

Figure 2.2: Depictions of the necessary information in the full dynamic programming solution in (a) and the "Four Russians" approach in (b).

starting at i, j as:

$$
\begin{aligned}
N(i,j) &= (L[i,j], L[i,j+1], \cdots, L[i,j+t]) \\
W(i,j) &= (L[i,j], L[i+1,j], \cdots, L[i+t,j]) \\
S(i,j) &= (L[i+t,j], L[i+t,j+1], \cdots, L[i+t,j+t]) \\
E(i,j) &= (L[i,j+t], L[i+1,j+t], \cdots, L[i+t,j+t])
\end{aligned}
$$

Based on $N(i,j)$ and $W(i,j)$ we create two offset vectors. These result in $I_1(i,j), I_2(i,j) \in \{0,1\}^t$:

$$
I_1(i,j)[k] = \begin{cases} 0 & \text{for } k = 1 \\ N(i,j)[k] - N(i,j)[k-1] & \text{for } 1 < k \leq t \end{cases}
$$

$$
I_2(i,j)[k] = \begin{cases} 0 & \text{for } k = 1 \\ W(i,j)[k] - W(i,j)[k-1] & \text{for } 1 < k \leq t \end{cases}
$$

The offsets are binary vectors, as the most significant difference between any two neighbouring elements will be one. This follows from the fact that we can add at most one element from one table entry to the next.

The main observation is that the vectors $E(i,j)$ and $S(i,j)$ are completely dependent on $A^{i,j}$, $B^{i,j}$, $I_1(i,j)$, $I_2(i,j)$, and the initial values $L[i,j]$.

To this extent we can describe $E(i,j)$ and $S(i,j)$ as a function of these elements. This function is referred to as the Basic Block Function (BBF) and is given as follows [FGM09]:

$$\text{BBF}(A^{i,j}, B^{i,j}, I_1(i,j), I_2(i,j), L[i,j]) = (E(i,j), S(i,j)) \tag{2.1}$$

We now pre-compute all possible $t \times t$ blocks. We do this by considering all possible t sized sequences and all possible offset vectors. This means, for domain D, we have $|D|^t$ possible sequences of length t and we have 2^t possibilities for the offset vector. This means in total we have $|D|^{2t}2^{2t}$ entries in the lookup table we create. As a value for $L[i,j]$ we use 0 resulting in $\text{bff}(\cdot,\cdot,\cdot,\cdot,0)$. Note that we can adapt this value to any desired constant by adding it as $\text{bff}(\cdot,\cdot,\cdot,\cdot,0) + C = \text{bff}(\cdot,\cdot,\cdot,\cdot,C)$. This constant will add the actual value of the first entry to the precomputed block. The combined precomputed block and the actual value result in the correct output of the block.

Once we have the look-up table, we can construct L based on the $t \times t$ blocks. We combine the first row and column with the input sequences and determine the corresponding pre-computed block. We can correct any value in this block by adding the value $L[i,j]$. Recall that we initialised the blocks with $L[i,j] = 0$. The pre-computed block leaves us with two new vectors, namely $E(i,j)$ and $S(i,j)$ using these new vectors we can determine $I_1(i+t,j)$ and $I_2(i,j+t)$. These new offset factors, combined with the already known data, allows us to determine new blocks. We can repeat this until we arrive at $L[m,n]$.

2.3 File

A file is a sequence of lines. The lines of a file consist of characters from an alphabet Σ. For example, Σ can be all `utf-8` characters in the case of a `utf-8` encoded file. The domain of a file f, given as D_f, is the set of possible strings we can construct based on Σ. In other words, $D_f = \Sigma^*$. Since f is a sequence of lines, we have the index set I_f. A file uses line numbers to index the lines. These line numbers start at 1 for the first line, and the final line number is at the very end of the file. Thus, the final line number is equal to the length of the file, $|f|$. Combining the start index with the final index result in a strictly increasing consecutive index set of integers starting

at one and ending at the length of the file. In other words, $I_f = \{\, 1, \ldots, |f| \,\}$, where $|f|$ denotes the total number of lines in a. This result in $f := (f_i)_{i \in I_f}$ for $f_i \in \Sigma^*$.

2.4 Difference Analysis

We can analyse the differences of two files. Given files a and b, the difference function describes the deletions, insertions, and replacements we need to perform to obtain b from a. We can derive the different operations based on $\mathrm{LCS}(a, b)$. Namely, the deletions are all values present in a and not in $\mathrm{LCS}(a, b)$, while the additions are the elements present in b and not in $\mathrm{LCS}(a, b)$. Note that we derive the replacements by taking the intersection of those two operations. For patch files, we will not use deletions. We, instead, will use retentions and insertions. This means that we use the elements in $\mathrm{LCS}(a, b)$ as retentions, and the elements in b and not in $\mathrm{LCS}(a, b)$ as insertions.

2.4.1 Three-way Difference Analysis

Three-way difference analysis aims to determine the difference between the files a and b with their common origin o. Common origin means that a and b are independently constructed files both based on file o. Using the three-way difference, we can build an edit-script that includes current information from all files.

To obtain this edit-script, we first need to align the three files. Autexier does this based on the $\mathrm{LCS}(a, b)$ [Aut15]. Autexier aligns the two sequences that derive from o. After the sequences are aligned, they determine the conflicts between a and b. Conflicts between a and b are those elements that are adjusted by both a and b. Since there are two valid new values for these positions the algorithm can no longer determine which value is the correct one.

Besides the conflicts, the function determines the values that a and b change, delete, or insert that are not part of a conflict. The non-conflicting data can directly be part of the edit-script since these values are adaptation to the origin file o that should be part of the new file.

This function result in an edit-script that describes the merging of a and b into their common origin o. This edit-script may contain conflict, we refer to these conflicts as merge conflicts.

2.5 Patch Files

We use the difference between two files to determine a patch file from one file to another. That is, the difference serves as a description to transform file a to file b.

We describe a patch file as two functions. These functions are the Ret and Ins function. We denote a patch file from a to b as $\mathsf{Patch}_{a\to b} = (Ret_{a\to b}, Ins_{a\to b})$.

2.5.1 Retain

The retain function, Ret, is a partial function that works on the index set of b, I_b. Ret describes the lines that are present in both the origin file a and in the target file b. The function receives an input line number from file b and gives the corresponding line number in a if such a line number exists. It does so based on a link function. $L_{\mathrm{LCS}(a,b),t}(i) \in I_t$ describes this link function for $i \in I_{\mathrm{LCS}(a,b)}$ and t is either a or b.

The link function ensures that the elements in the LCS are associated with indices of the original sequences. For any given Longest Common Subsequence there might be different possible embeddings. We do not concern ourselves with the optimal mapping; the link function encompasses this property. The link function receives an index from the LCS and links this to an index in one of the two super sequences.

The link function ensures that linking the index allows all other indices – trailing and preceding – to associate with an index correctly. This means that for any given index in the index set of the LCS, i, and its associated index, j, in the index set of the supersequence a, it must hold that for all indices $i' \in I_{\mathrm{LCS}(a,b)}$ where $i' < i$ there is an associated index $j' \in I_a$ such that $j' < j$. The converse holds for $i'' > i$ and $j'' > j$.

More formally, for $i \in I_{\mathrm{LCS}(a,b)}$ with $\min_{j\in I_a}$ such that $\mathrm{LCS}(a,b)_i = a_j$ and $\mathrm{LCS}(a,b)^i \subseteq a^j$ and $\max_{j'\in I_a}$ such that $\mathrm{LCS}(a,b)_i = a_{j'}$ and $\mathrm{LCS}(a,b)^{i,|\mathrm{LCS}(a,b)|} \subseteq a^{j',|a|}$, we note that the linked index can be any value between these two extremes. Thus, for $L_{\mathrm{LCS}(a,b),a}(i) \in I_a$ it holds that $j \leq L_{\mathrm{LCS}(a,b),a}(i) \leq j'$. Note that a substring is a sequence. Recall we use $\mathrm{LCS}(a,b) \subseteq a$ to denote that $\mathrm{LCS}(a,b)$ is a subsequence of a.

Furthermore, the link function ensures that the order of the elements is proper. Proper ordering means that we have maintain the relative order of the sequences. To ensure this is the case we introduce one more property of the link function. The

link function ensures $L(i) > L(i')$ if and only if $i > i'$, where $i, i' \in I_{\mathrm{LCS}(a,b)}$ and $L(i), L(i') \in I_a$.

Using the link function to link the indices in the LCS to its two super sequences leads to the following Domain and Image for the *Ret* function;

$$\mathrm{Dom}(Ret) = \left\{ L_{\mathrm{LCS}(a,b),b}(i) \mid i \in I_{\mathrm{LCS}(a,b)} \right\} \tag{2.2}$$

$$\mathrm{Im}(Ret) = \left\{ L_{\mathrm{LCS}(a,b),a}(i) \mid i \in I_{\mathrm{LCS}(a,b)} \right\} \tag{2.3}$$

Note that $|\mathrm{Dom}(Ret)| = |\mathrm{Im}(Ret)|$ and that the relation, as described, is bijective.

This function does not only describe the lines we retain; it inherently describes the deletions as well. Namely, those line numbers in a that do not map to a corresponding line number in b are the line numbers of deleted lines from a. The latter assertion means we could describe a patch file as a combination of deletions and insertions. However, it will become clear that we require insertions and retentions for the composition of patch files.

2.5.2 Insert

The insert function describes the remaining lines, if any, that are present in the target file b but not in the origin file a. To this extent, upon receiving a line number in b, it outputs the new content of that line if it exists. This result in the following Domain and Image;

$$\mathrm{Dom}(Ins) = I_b \setminus \mathrm{Dom}(Ret) \tag{2.4}$$

$$\mathrm{Im}(Ins) = b \setminus \mathrm{LCS}(a, b) \tag{2.5}$$

Since a file is a sequence with a strictly increasing consecutive index set, we can determine the domain based on the file length in the given way.

2.5.3 Composition of Patch Files

We can compose two patch files that are created chronologically. This means that for files a, b, c, and d with the patch files $\mathsf{Patch}_{a\to b}$, $\mathsf{Patch}_{b\to c}$, and $\mathsf{Patch}_{c\to d}$ we can compose $\mathsf{Patch}_{a\to b}$ and $\mathsf{Patch}_{b\to c}$ resulting in $\mathsf{Patch}_{a\to c}$. We cannot compose $\mathsf{Patch}_{a\to b}$ and $\mathsf{Patch}_{c\to d}$ as the target file for the former and the origin file for the latter differ.

To create $\mathsf{Patch}_{a\to c}$ based on $\mathsf{Patch}_{a\to b}$ and $\mathsf{Patch}_{b\to c}$ we need to define the functions $Ret_{a\to c}$ and $Ins_{a\to c}$. We start with $Ret_{a\to c}$.

Composed Retain

The values that we retain from file a to c are those values that we retain from a to b and from b to c. Note that if we do not retain the value from a to b and insert it again in the same relative line from b to c, we do not observe this as a retention in the composition of patch files.

This functionality is encapsulated in the composition of the two known *Ret* functions and is given as

$$Ret_{a\to c} = (Ret_{a\to b} \circ Ret_{b\to c}) \tag{2.6}$$

Note that we first apply $Ret_{b\to c}$. This is in line with the function definition as $Ret_{b\to c}$ is a partial function over I_c to I_b and $Ret_{a\to b}$ is a partial function over I_b to I_a. Thus, the new Domain of the function is $\{i \mid i \in I_c, Ret_{b\to c}(i) \in \mathrm{Im}(Ret_{b\to c}) \cap \mathrm{Dom}(Ret_{a\to b})\}$ and the Image is $\{Ret_{a\to b}(i) \mid i \in \mathrm{Im}(Ret_{b\to c})\}$.

Composed Insert

In the case of $Ins_{a\to c}$ we need to define two cases. The first case is the insertions from b to c. These insertions are part of the new insert function as they end up in the final file and are not part of the $Ret_{a\to c}$ function.

The second case are those values that are inserted from a to b *and* are retained from b to c. This results in the following function;

$$Ins_{a\to c} = \begin{cases} Ins_{a\to b} \circ Ret_{b\to c} \\ Ins_{b\to c} \end{cases} \tag{2.7}$$

The domain for $Insert_{a\to c}$ is $\{i \mid i \in I_c, Ret_{b\to c}(i) \in \mathrm{Im}(Ret_{b\to c}) \cap \mathrm{Dom}(Ins_{a\to b})\} \cup \mathrm{Dom}(Ins_{b\to c})$.

2.5.4 Effective Patch and Composition

An effective patch file warrants its use. If a patch file is more extensive than its target file, it is rendered ineffective, since at that point, we might as well send the target file itself. In the case of patch composition, we consider one more aspect. While a composed patch file may have a greater length than the target, the composition can also combine multiple patch files. Combining multiple patch files can lead to a composed patch file that is longer than the target file. However, it is shorter than the concatenation of the patch files. Since it is shorter than the concatenation of the original patches, we say that the composition is effective.

We say a patch file is *effective* if the size of $\mathsf{Patch}_{a\to b}$ is smaller than the size of b. We say a patch composition of file set $P = \{\mathsf{Patch}_{f_1\to f_2}, \ldots, \mathsf{Patch}_{f_{n-1}\to f_n}\}$ is effective if $|\mathsf{Patch}_{f_1\to f_n}| < \sum_{p\in P} |p|$, where $\mathsf{Patch}_{f_1\to f_n}$ is the composition of all patches in P.

2.6 Delannoy Number

The Delannoy number derives its name from Henry Delannoy, an amateur mathematician [BS05]. Delannoy numbers describe all possible paths from the origin point to the opposite corner of a rectangular grid [Sul03]. The size of the grid is (m, n) where m is the size of the x-axis and n the size of the y-axis. The possible steps to move from one coordinate to another coordinate are diagonal, horizontal, and vertical.

Definition 2.6.1 (Delannoy Number)**.** The Delannoy Number represent all possible path from the origin $(0, 0)$ to the coordinate (m, n). The function is given as;

$$D(m,n) = \sum_{k=0}^{\min(m,n)} \begin{pmatrix} m+n-k \\ m \end{pmatrix} \begin{pmatrix} m \\ k \end{pmatrix} \tag{2.8}$$

The possible steps the Delannoy number considers are the same as those taken in the LCS. We will use this number in Chapter 4 to show the number of possible

routes we can take to arrive at a position in the LCS dynamic programming table.

2.7 Encryption

We use different kinds of encryption. We use public- and symmetric key schemes in this thesis. This section introduces the symmetric system and its notation.

We use AES encryption in Galois/Counter Mode [DBN+01, Dwo07]. This encrypts a message m using a key k and Initialisation Vector IV. We denote the AES encryption of m under k using IV as $\text{AES}_k^{IV}(m)$.

2.8 Homomorphic Encryption

We use homomorphic encryption to perform operations over encrypted data. The goal of homomorphic encryption schemes is to allow arbitrary computations on encrypted data [ABC+15]. Schemes vary in the number of operations they support. Partial Homomorphic Encryption (PHE) supports one operation, addition or multiplication, Fully Homomorphic Encryption (FHE) supports both operation. These schemes have a public and secret key. Where the owner of the secret key can decrypt the message encrypted under the associated public key.

Let $\mathcal{E}$ be a public-key homomorphic encryption scheme. It has associated public and secret key given as (pk, sk). Let $\mathbb{M}$ denote the message space of $\mathcal{E}$ the encrypt function for $m \in \mathbb{M}$ and random value r is given as $E_{pk}(m; r)$ and the decrypt function is defined such that $D_{sk}(E_{pk}(m; r)) = m$. For party A using $\mathcal{E}$ with public-secret key-pair (sk_A, pk_A) and message $m \in \mathbb{M}$ we use $E_A(m)$ as a shorthand for $E_{pk_A}(m; r)$ and $D_A(m)$ for $D_{sk_A}(m)$.

Furthermore, for the scheme $\mathcal{E}$ and $m_1, m_2 \in \mathbb{M}$, we have the homomorphic operation $\oplus$ such that $D_{sk}(E_{pk}(m_1) \oplus E_{pk}(m_2)) = m_1 + m_2$ and the operation $\otimes$ such that $D_{sk}(E_{pk}(m_1) \otimes E_{pk}(m_2)) = m_1 \times m_2$.

2.9 Security Model

We define three sets of participants:

Providers $\mathcal{P}$ The set of users that provide data. Data can either be a file or a patch file. In the case of Secure LCS, `diff`, and `diff3` the input is a file. In the case of Secure Compose the input is a patch file.

Servers $\mathcal{S}$ The set of honest-but-curious non-colluding entities that compute and store the data.

Receivers $\mathcal{R}$ The set of users that can view the result.

It is possible that $\mathcal{P} = \mathcal{R}$. The cardinality of either set can be 1; this means that a single entity can provide multiple data points if required and be the only receiver.

The goal is to develop a protocol that allows the entities in $\mathcal{P}$ to provide data to $\mathcal{S}$ so that it can determine the result for the entities in $\mathcal{R}$ to retrieve. However, $\mathcal{S}$ should not learn the outcome.

All entities in $\mathcal{S}$ are honest-but-curious. Honest-but-curious adversaries adhere to the protocol but try to distil valuable conclusions from the information they legitimately possess [Gol04, p. 603].

Note that no single entity in $\mathcal{S}$ should learn the result. We assume the entities in $\mathcal{S}$ do not collude. Non-colluding honest-but-curious parties are parties that avoid sharing useful information outside of the dictated protocol [KMR11, p. 10]. A consequence of non-colluding servers is that the combined information over all entities in $\mathcal{S}$ could reveal the result. However, since the parties do not collude no single entity can determine the output of the function.

Notation	Description
	General
$\mathbb{Z}_n$	The set of integers modulo n.
$a \in_R A$	a randomly sampled from A.
$D(m, n)$	The Delannoy number for the coordinate m and n.
$\mathfrak{S}_n$	The symmetric group on n elements indicating all possible permutations for a set of size n.
$f \circ g$	Composition of two functions f and g such that $f(g(\cdot))$.
$a \mid b$	a divides b such that $\frac{a}{b} \in \mathbb{Z}$.
	Sequence
$\|s\|$	Length of sequence s.
I_s	The set of indices for sequence s.
s_i	The element with index i in sequence s for $i \in I_s$.
$s \,\|\|\, e$	Concatenation of elements e to sequence s .
$s^{m,n}$	Substring of sequence s from index m to n.
s^n	Shorthand for $s^{\min(I_s),n}$.
$\text{Pred}_{I_s}(i)$	The largest preceding index of i in index set I_s of sequence s.
$\text{Last}(s)$	The last element of sequence s.
$\text{Init}(s)$	The subsequence of sequence s excluding $\text{Last}(s)$.
$\text{LCS}(a, b)$	The Longest Common Subsequence of sequence a and b.
	Patch
$Ret_{a\to b}$	Function that maps the indices of retained values in file b to the corresponding indices in a.
$Ins_{a\to b}$	Function that maps the indices of b not in $Ret_{a\to b}$ to the corresponding new values.
$\mathsf{Patch}_{a\to b}$	Patch consting of $Ret_{a\to b}$ and $Ins_{a\to b}$.
	Encryption
$\text{AES}^k_{IV}(m)$	AES-GCM encryption of m with Initialisation Vector IV under the key k.
$\mathcal{E}$	Public-key homomorphic encryption scheme.
$E_{pk_A}(m; r)$	The encryption of m under the public key pk_A of A using randomness r.
$D_{sk_A}(c)$	Decryption of c under the secret sk_A of A.
$E_A(m)$	Shorthand for $E_{pk_A}(m; r)$, for message m, randomness r, and public key pk_A of A.
$D_A(c)$	Shorthand for $D_{sk_A}(c)$, for message m, randomness r, and secret key sk_A of A.
$E_A(m_1) \oplus E(m_2)$	The homomorphic addition operation.
$E_A(m_1) \otimes E(m_2)$	The homomorphic multiplication operation.

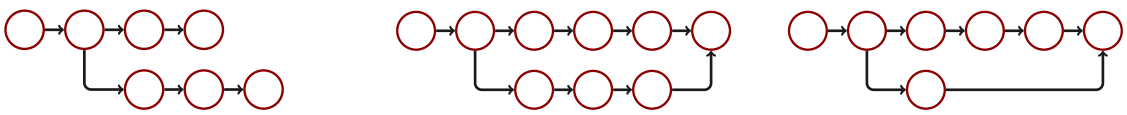
Table 2.1: Introduced notation

Chapter 3

Secure Patch Composition

This section discusses possible approaches to the secure composition of patch files. We introduce the notion of Secure Compose – sCompose – and suggest possible approaches to the problem. In the Version Control System (VCS) scenario we consider, patch composition plays an important role. A VCS allows multiple people to work on the same piece of software while at the same time keeping the system maintainable. A repository stores the files of a project. Everyone that has access to the repository has access to the files that reside within it. We assume that all authors of a project are allowed to read and write to the remote server and, by extension, are allowed to process the data contained within the repository.

Commonly, repositories allow for branching and merging these branches in the version history. Authors can use branches to distinguish between development and production versions and even allow for granular extensions only concerning a particular product feature. The complete history of a repository is a tree that consists of commitments. Commitments combine meta-data such as the author, textual description of the change, and the date, with the patch description.



(a) (b) (c)

Figure 3.1: Depiction of different scenarios in a Version Control System with regards to branching. (a) shows the start of a branch and the parallel existence of the main branch and feature branch. (b) depicts the merging of a feature branch with the main branch. (c) shows the same tree and feature as (b) with a squashed commitment.

The fine-grained branches focussed on features, named *feature branches*, can be merged with the main branch. For maintainability and stability, it might be desirable to squash commitments before performing such a merge operation [1]. Squash commits combine multiple commits in one single commit. Combining commitments ensures that we cannot roll back to a partially working version of a feature. It results in a tidy version history with working code if we wish to roll back to a certain point. Figure 3.1 depicts the different possible trees as a result of merging. We want to maintain this functionality in a secure repository.

This section discusses multiple approaches to the problem of securely composing patch files. These approaches are iterative. However, they do not all have the same Security Model. We first discuss the ideal functionality of our Secure Compose function. With secure, we mean that the composition function does not leak information. We elaborate on the exact functionality and the different Security Models in the next section.

3.1 Security Setting and Ideal Functionality

All authors have access to the repository and should be able to edit the project at will. Thus we see that the set of data providers $\mathcal{P}$ is equal to the group of receivers, $\mathcal{R}$. Note that we can perform difference analysis over the cleartext. Since $\mathcal{P} = \mathcal{R}$, all parties can share the same secret key. Thus there is no reason for any of the parties to perform the difference analysis in ciphertext space. This plaintext difference analysis allows for the cheapest difference analysis currently available.

We have two participants $p, p' \in \mathcal{P}$, note that $p = p'$ is possible. Both parties have access to file f_i. On individual basis, p has file f_o and p' has f_t. p creates $\mathsf{Patch}_{f_o \to f_i}$ and p' creates $\mathsf{Patch}_{f_i \to f_t}$. The goal is for $\mathcal{S}$ to determine $\mathsf{Patch}_{f_o \to f_t}$ without learning the number of operations, the location of operations, or what the content of the lines is. $\mathcal{S}$ stores the result until any $r \in \mathcal{R}$ requests the output. The ideal functionality of the Secure Compose function is given in Figure 3.2

We use two different sets for $\mathcal{S}$. We have a a single server setting, $\mathcal{S} = \{S\}$ and a two server setting, $\mathcal{S} = \{S_\mathsf{P}, S_\mathsf{D}\}$. In the latter case, we consider the two servers to be *non-colluding*. Non-colluding servers are servers that do not communicate other

[1] `https://blog.carbonfive.com/always-squash-and-rebase-your-git-commits/`

Functionality $\mathcal{F}_{\mathsf{sCompose}}$

Participants: $p, p' \in \mathcal{P}$ for the set of providers $\mathcal{P}$, $R \subseteq \mathcal{R}$ for the receiver set $\mathcal{R}$.
Parameters: Composition Function $C : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$, for patch files from input-space $\mathbb{F}$.
Input:
- $p : f_p \in \mathbb{F}$,
- $p' : f_{p'} \in \mathbb{F}$,

Output:
- p obtains $C(f_p, f_{p'})$ if $p \in R$ else $\perp$
- p' obtains $C(f_p, f_{p'})$ if $p' \in R$ else $\perp$
- R obtains $C(f_p, f_{p'})$

Figure 3.2: Ideal Functionality for the Secure Compose function.

than specified in the protocol.

3.2 Single Server

In the single server scenario, we propose two methods to hide information. The first method hides the content and retains the compose property as described in Section 2.5.3. The second method hides all information but loses the ability to compose patch files effectively.

3.2.1 Hiding Line Content

The image of a patch file's Ins and Ret function reveal the content of the target file when combined with the origin file. The Ins function even shows the content of certain lines without the origin file. We first hide these values. Since the composition of patch files is independent of the content of insertions, we are free to choose whatever method of hiding the content we want. Since all authors have access to the project, they can share the same secret key to hide the content of the patch files.

After the difference analysis is complete and we have the sets of values to retain and insert, we start hiding the content of the insertions.

All authors of the project possess a predetermined key k and Initialisation Vector

IV. We use AES, as described earlier, to encrypt the image of the Ins. Besides the content, c, we include the version number, v, and the line number, l, in the encryption. For every line, the author encrypts the following $\text{AES}^k_{IV}(l \,||\, v \,||\, c)$ where $l \,||\, v \,||\, c$ denotes the concatenation of the line number, version, and the content of the line. The content of the line is the value of the insert function for that line, $c = Ins(l)$. Including the version and the line number means every encrypted line of a patch file is unique from version to version. The content of a line can either be the content of an existing line or a new line. The only content in a patch file is the new line in the Ins function. The other form of content is not part of the patch file but the origin file. For a line l with operation o and content c in version v the functions take the following form:

$$Ins(l) = \text{AES}^k_{IV}(l \,||\, v \,||\, Ins(l)) \tag{3.1}$$

Note that *Ret* stays the same. *Ret* stays the same since the origin file is not present on the server and thus hides the content of the lines associated with *Ret* from $\mathcal{S}$.

The combination of encrypted and hidden lines means that $\mathcal{S}$ cannot learn anything about the content of a patch file. Furthermore, different lines have distinct encryptions as we include the version number and line number as unique identifier per line. This particular encryption means that $\mathcal{S}$ cannot determine how many lines are the same in the patch file.

Since the retain function stays the same in this case and the composition of patch files is independent of the content of Ins, we observe that the composition functions as provided in Equation 2.6 and 2.7 still hold.

3.2.2 Hiding All Content

In the previous approach, we still openly show the line number and the associated operation. This information can lead an attacker to learn something about the document. Instead of encrypting the file line by line, we can also encrypt the whole file. Doing this prevents an adversary from deriving the location or nature of an operation while a key holder can construct the original patch file before application. Since the ordering is no longer relevant, we can encrypt the whole file. Randomly permuting the order of lines before encryption guarantees lines cannot be associated with their

original position. In the randomly ordered new file, the lines should contain their original line number as a prefix. The inclusion of the original line numbers allows the receiver, $\mathcal{R}$, to construct the original patch file. For patch file $\mathsf{Patch}_{a\to b}$ we now have $\mathrm{AES}^k_{IV}(v \,||\, \mathsf{Patch}_{a\to b})$. This encryption hides the operations, the associated line numbers, and the content.

While we hide all content this approach does remove the effective composition of patch files. We can still compose patch files. However, composition is reduced to concatenation. For files a, b, and c and the associated patch files $\mathsf{Patch}_{a\to b}$ and $\mathsf{Patch}_{b\to c}$ it holds that $|\mathsf{Patch}_{a\to c}| = |\mathsf{Patch}_{a\to b}| + |\mathsf{Patch}_{b\to c}|$, where $\mathsf{Patch}_{a\to c}$ is the composed patch file. Where previously the size of $\mathsf{Patch}_{a\to c}$ was no greater than the size of c.

Besides the loss of effective composition, we also leak the length of the files. We can solve this problem by introducing padding. Padding files to a fixed size, and consequently, padding patch files solves the situation where we leak the length of the files. Limiting this size to a fixed one does lower the flexibility of the application. We can no longer use files of any length but have a strict upper limit.

3.2.3 Analysis

We propose two different approaches. The first approach upholds the composing functionality as described in Section 2.5.3. This approach hides the content of a line by encrypting it, thereby hiding the new content or the retained line number. However, this comes at the cost of leaking the number, target location, and type of the operations in the patch file. This information could lead an adversary to deduce the nature of the document or which author is productive and which one is not. This means that this function does not simulate $\mathcal{F}_{\mathsf{sCompose}}$.

Our second approach does not leak as much information as the function suggested above. It does not leak the operation location. However, it still leaks the number of lines in a file. Furthermore, hiding the information comes at the cost of losing the effective composition we introduced. This is clear as the length of the patch file composed from the consecutive patch files $F = \{\mathsf{Patch}_{f_1\to f_2}, \ldots, \mathsf{Patch}_{f_{n-1}\to f_n}\}$ is $\sum_{p\in P} |p|$. This size of the new patch file means it is not an effective composition of patch files. The only time this composition results in an effective patch file is when all patches in F only consists of insertions. All other cases result in a longer patch

	n				
	10	100	1000	10000	100000
3.2.1	$8.8 \cdot 10^{-5}$s	0.0008s	0.0082s	0.0814s	0.8133s
3.2.2	$2.0 \cdot 10^{-6}$s	$5.0 \cdot 10^{-6}$s	$3.6 \cdot 10^{-5}$s	0.0003s	0.0033s

Table 3.1: Timing results in seconds for the single server setting. Note that we time the composition so we disregard encryption or creation time.

file than the target file itself. Thus this patch file is not effective.

Introducing padding in the last function does solve its leakage problem. We argue that a padded version of the last function simulates $\mathcal{F}_{\mathsf{sCompose}}$. It hides the length of the files since all files have the same length. Furthermore, it hides the operations and their location. It does so by encrypting the entire file. This encryption means that we can no longer associate line numbers with operations or derive what the operations themselves are. Thus we hide, the length of a file, number of operations, location of operations, and insertions content.

Timing

We test the timing in a real-world scenario. We compose two files multiple times and report the result. We only time the time it takes for two patch files to compose. We do not consider any encryption or creating of patch files in the timing. Table 3.1 gives the time for different number of compositions. The patch files consist of 30 lines of which 17 are retentions and 13 are new insertions. The insertions are lines of 100 elements. A padded file is twice the size of a normal file. The test machine is equipped with a AMD EPYC 7V12 CPU and 28GB of RAM.

We see that the quickest composition is that of concatenating files. This makes sense as this is the simplest operation. The effective composition method scales linearly with the number of compositions we perform.

3.3 Non-Colluding Two-Servers

This section suggests an approach that allows for the effective composition of patch files and hides the location of operations. We extend this idea using a fixed file size to hide the number of operations partially. The proposed protocols operate under the *non-colluding* server assumption and use two of these servers.

3.3.1 Hiding Line Numbers

Hiding the line numbers is the first step in hiding the content of a patch file. For $\mathsf{Patch}_{a\to b} = (Ret_{a\to b}, Ins_{a\to b})$, we have $\mathrm{Dom}(Ret_{a\to b})$, $\mathrm{Dom}(Ins_{a\to b})$, and $\mathrm{Im}(Ret_{a\to b})$ that contain line numbers. These line numbers are subsets of two different index sets. The relation between the domain and image of the functions and the index sets is $\mathrm{Dom}(Ret_{a\to b}) \cup \mathrm{Dom}(Ins_{a\to b}) = I_b$ and $\mathrm{Im}(Ret_{a\to b}) \subseteq I_a$. The goal is to hide these line numbers from $\mathcal{S}$. Note that besides hiding the line numbers we also hide the content of an insertion using encryption. Encrypting the content of an insertion is the same procedure as we proposed in the first approach of Section 3.2.1. This means that the output of Ins function is encrypted under shared key, k, and Initialisation Vector, IV.

We hide the line numbers by randomly permuting the respective domains and image. For $\mathsf{Patch}_{a\to b}$, we randomly select two permutations $\pi_b \in_R \mathfrak{S}_{|b|}$ and $\sigma_a \in_R \mathfrak{S}_{|a|}$. $\mathfrak{S}_n$ is the set of all possible permutations on a set of size n. We use the former permutation, $\pi_b \colon I_b \to I_b$, to permute the domain of both functions. We use the latter permutations, $\sigma_a \colon I_a \to I_a$, to permute the image of $Ret_{a\to b}$.

We use the permutation and the original patch description to create a new patch description that does not reveal the target and origin location of lines. We refer to these new functions as the permuted patch description. We note the functions of the permuted patch description by $Ret^{\pi_b,\sigma_a}_{a\to b}$ and $Ins^{\pi_b}_{a\to b}$. Previously, to obtain the value of line $l_b \in I_b$ we would get the correct result by evaluating either function with l_b as input. If $l_b \in \mathrm{Dom}(Ret_{a\to b})$ it is a retention. Conversely, if $l_b \in \mathrm{Dom}(Ins_{a\to b})$ it is an insertion. For the permuted functions the input new input, $l_b^{\pi_b}$, is the permuted line number. This hides the actual location of the new operation. The input for the permuted functions is $l_b^{\pi_b} = \pi_b(l_b)$.

In the case of $Ret^{\pi_b,\sigma_a}_{a\to b}$ we do not only permute the input, we also permute the output. Without permuting the image of Ret we would reveal the lines we retain from the previous file. Instead of returning the normal value we would get from $Ret_{a\to b}(l_b)$ we now set the result to $\sigma_a(Ret_{a\to b}(l_b))$. Combining all the permutations results in the following descriptions:

$$Ret_{a\to b}^{\pi_b,\sigma_a} = \sigma_a \circ Ret_{a\to b} \circ \pi_b^{-1} \tag{3.2}$$
$$Ins_{a\to b}^{\pi_b} = \mathrm{AES}_k^{IV} \circ Ins_{a\to b} \circ \pi_b^{-1}. \tag{3.3}$$

Here π_b^{-1} denotes the inverse of π_b. Applying the inverse operation of σ_a to $Ret_{a\to b}^{\pi_b,\sigma_a}$ results in the actual line number $l_a \in I_a$ that we preserve. We prove correctness of the new functions with permuted output and input. For $i \in \mathrm{Dom}(Ret_{a\to b})$ we have:

$$\sigma^{-1}(Ret_{a\to b}^{\pi_b,\sigma_a}(\pi(i))) = \sigma^{-1}((\sigma \circ Ret_{a\to b} \circ \pi^{-1})(\pi(i))) \tag{3.4}$$
$$= (Ret_{a\to b} \circ \pi^{-1})(\pi(i)) \tag{3.5}$$
$$= Ret_{a\to b}(i) \tag{3.6}$$

For $j \in \mathrm{Dom}(Ins_{a\to b})$ we have:

$$Ins_{a\to b}^{\pi_b}(\pi(j)) = (\mathrm{AES}_k^{IV} \circ Ins_{a\to b} \circ \pi^{-1})(\pi(j))) \tag{3.7}$$
$$= \mathrm{AES}_k^{IV} \circ Ins_{a\to b}(j) \tag{3.8}$$

The permuted patch file consists of four elements. Namely, the two new functions – $Ret_{a\to b}^{\pi_b,\sigma_a}$ and $Ins_{a\to b}^{\pi_b}$ – and the respective permutations – π_b, σ_a. $\mathsf{Patch}_{a\to b}^{\pi_b,\sigma_a} = (Ret_{a\to b}^{\pi_b,\sigma_a}, Ins_{a\to b}^{\pi_b}, \pi_b, \sigma_a^{-1})$ describes the permuted patch.

Example: Creating a Secure Patch File

Suppose Alice has generated the patch file $\mathsf{Patch}_{a\to b}$ for files a and b. Where $|a| = 5$ and $|b| = 7$. The changes from file a to b are the insertion of two new line on line 1 and 2. This result in the following $Ins_{a\to b}$ and $Ret_{a\to b}$ functions,

$$Ret_{a\to b}(i) = i - 2 \qquad \text{if } i \in \{3, \ldots, 7\}$$
$$Ins_{a\to b}(i) = b_i \qquad \text{if } i \in \{1, 2\}$$

Alice generates the following two permutation:

$$\pi_b = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 2 & 4 & 5 & 1 & 6 & 7 & 3 \end{pmatrix} \qquad \sigma_a = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 1 & 5 & 4 \end{pmatrix}$$

She applies π_b to both functions. This results in hiding the target location of the newly inserted value or the retained value. Additionally, she applies σ_a to the Image of $Ret_{a\to b}$ to hide the location of the origin of the retained line. This result in the following functions:

$$Ret^{\pi_b,\sigma_a}_{a\to b}(\pi_b(i)) = \sigma_a(i-2) \qquad \text{if } i \in \{3,\ldots,7\}$$
$$Ins^{\pi_b}_{a\to b}(\pi_b(i)) = b_i \qquad \text{if } i \in \{1,2\}$$

To push a permuted patch file to the remote location, $\mathcal{S}$, we split the patch in the descriptions ($Ret^{\pi_b,\sigma_a}_{a\to b}$, $Ins^{\pi_b}_{a\to b}$) and the permutations (π_b, σ_a). We send the permuted patch description to S_D and the permutations to S_P. Since the entities in $\mathcal{S}$ do not collude we observe that S_D cannot satisfy the operation in Formulas 3.4 and 3.7. Thus, S_D cannot derive the actual values of either of the original functions $Ret_{a\to b}$ or $Ins_{a\to b}$. S_P has no information on the descriptions at all. Consequently, this server cannot derive the original functions either.

The functions still reveal the number of each operation we perform and the size of the files. The size of both functions – $|Ret^{\pi_b,\sigma_a}_{a\to b}|$, $|Ins^{\pi_b}_{a\to b}|$ – reveal the number of entries per operation. Furthermore, the size of the permutations reveal the size of the target file, b, and the origin file, a – $|\sigma_a|$, $|\pi_b|$.

3.3.2 Hiding File Size

To prevent disclosing the length of the origin and target file, we use a fixed file size. A fixed file size means that the length of the permutations and the functions do not reveal information about the length of either file other than that it is smaller or equal to the maximum size. Furthermore, this form of padding hides the number of insertions. We use padding to create an equal file size. The padding means we fill the Ins and Ret functions with entries till they both reach their maximum size. The size of both functions should be equal to prevent revealing information about the nature of the file. We note a padded patch file using $\mathsf{Pad}(\mathsf{Patch}_{a\to b})$. This means we pad both functions, given by $\mathsf{Pad}(Ret^{\pi_b,\sigma_a}_{a\to b})$ and $\mathsf{Pad}(Ins^{\pi_b}_{a\to b})$, where we ensure that $|\mathsf{Pad}(Ret^{\pi_b,\sigma_a}_{a\to b})| = |\mathsf{Pad}(Ins^{\pi_b}_{a\to b})|$. For Ret we pad by inserting values in the origin file and carrying these to the target file. For Ins we pad by adding new values on unused indices. For ease of use, we assume we build the padding after the actual content.

The desired length of actual content in a file dictates the length of a padded file. We say that the maximum number of lines in an unpadded file is n. Consequently, we note that $|Ret_{a\to b}| + |Ins_{a\to b}| \leq n$, since the patch file describes the target file. To hide the number of operations we pad both the functions to the size n this means the number of lines in a padded file is $2 \cdot n$ and $|\mathsf{Pad}(Ret^{\pi_b,\sigma_a}_{a\to b})| = |\mathsf{Pad}(Ins^{\pi_b}_{a\to b})| = n$. This ensures that we do not reveal anything about the nature of the document by indicating which operation the patch leans to.

Example: Padding a Secure Patch File

Alices has files a and b where $|a| = 2$ and $|b| = 3$. The length of padded files is 6. Note that $|b|$ is the maximum size content can have. File b adds one line to the two existing lines of a. This results in $\mathsf{Patch}_{a\to b} = (Ret_{a\to b}, Ins_{a\to b})$ where

$$Ret_{a\to b}(x) = \begin{cases} 1 & \text{if } x = 1 \\ 2 & \text{if } x = 3 \end{cases} \tag{3.9}$$

$$Ins_{a\to b}(x) = \quad 1 \quad \text{if } x = 2 \tag{3.10}$$

Alice pads these functions. Since the padded file size is 6 we have $|\mathsf{Pad}(Ret_{a\to b})| = |\mathsf{Pad}(Ins_{a\to b})| = 3$. Alices needs one extra dummy line in file a and two newly inserted lines to pad the functions. Thus we have the following one extra line in file a that contains no useful information, this is an empty line.

$$\mathsf{Pad}(Ret_{a\to b})(x) = \begin{cases} 1 & \text{if } x = 1 \\ 2 & \text{if } x = 3 \\ 3 & \text{if } x = 4 \end{cases} \tag{3.11}$$

Note that we use a line number for the input that is not yet part of any of the functions. Anagously, we insert empty lines in the remaining spots of Ins;

$$\mathsf{Pad}(Ins_{a\to b})(x) = \begin{cases} 1 & \text{if } x = 2 \\ \emptyset & \text{if } x \in \{5, 6\} \end{cases} \tag{3.12}$$

The new padded functions works the same as normal functions would. However, there is now data that does not contain information for the end-user. $\mathcal{R}$ can filter this data after receiving the result.

	$\mathsf{Patch}_{f_i \to f_t}$	$\mathsf{Patch}_{f_o \to f_i}$
S_P	$\pi_{f_t} \colon I_{f_t} \to I_{f_t}$ $\sigma_{f_i}^{-1} \colon I_{f_i} \to I_{f_i}$	$\pi_{f_i} \colon I_{f_i} \to I_{f_i}$ $\sigma_{f_o}^{-1} \colon I_{f_o} \to I_{f_o}$
S_D	$Ret_{f_i \to f_t}^{\pi_{f_t}, \sigma_{f_i}}$ $Ins_{f_i \to f_t}^{\pi_{f_t}}$	$Ret_{f_o \to f_i}^{\pi_{f_i}, \sigma_{f_o}}$ $Ins_{f_o \to f_i}^{\pi_{f_i}}$

Figure 3.3: Information of two consecutive patch files held by $\mathcal{S} = \{S_\mathsf{D}, S_\mathsf{P}\}$.

3.3.3 Composition

We now consider the composition of two consecutive permuted patch files. Recall we have two providers $p, p' \in \mathcal{P}$ with shared file f_i. p has f_o and creates $\mathsf{Patch}_{f_o \to f_i}^{\pi_{f_i}, \sigma_{f_o}} = (Ret_{a \to b}^{\pi_b, \sigma_a}, Ins_{a \to b}^{\pi_b}, \pi_{f_i}, \sigma_{f_o}^{-1})$, p sends the description $(Ret_{f_o \to f_i}^{\pi_{f_i}, \sigma_{f_o}}, Ins_{f_o \to f_i}^{\pi_{f_i}})$ to S_D and the permutations $(\pi_{f_i}, \sigma_{f_o})$ to S_P. Analgolously, p' has f_t, creates $\mathsf{Patch}_{f_i \to f_t}^{\pi_{f_t}, \sigma_{f_i}} = (Ret_{a \to b}^{\pi_b, \sigma_a}, Ins_{a \to b}^{\pi_b}, \pi_{f_t}, \sigma_{f_i}^{-1})$ and sends the description $(Ret_{f_i \to f_t}^{\pi_{f_t}, \sigma_{f_i}}, Ins_{f_i \to f_t}^{\pi_{f_t}})$ to S_D and permutations $(\pi_{f_t}, \sigma_{f_i})$ to S_P. Figure 3.3 shows the information held by each server.

To compose two patch files we need to relate $\mathrm{Im}(Ret_{f_i \to f_t})$ to $\mathrm{Dom}(Ret_{f_o \to f_i})$ and $\mathrm{Dom}(Ins_{f_o \to f_i})$. These values are part of the target file f_t. In the previous sections we permute the inputs and the output of the Ret and Ins functions that contain line numbers. In this section we create a new permutation that allows S_D to link the two patch files without revealing the location of the operation.

S_P creates this new permutation by composing two permutations. Referring to Figure 3.3 we see that two permutations have the same domain. These two permutations are $\sigma_{f_i}^{-1}$ and π_{f_i}. Using these two permutations we can link $Ret_{f_i \to f_t}$ to the functions of $\mathsf{Patch}_{f_o \to f_i}$. We do this by creating the new permutation $\alpha = \pi_{f_i} \circ \sigma_{f_i}^{-1}$. Recall that $\pi_{f_i} \in_R \mathfrak{S}_{|b|}$ and $\sigma_{f_i} \in_R \mathfrak{S}_{|b|}$, thus the composition of these two permutations is also a random permutation in $\mathfrak{S}_{|b|}$.

S_D can use the new permutation α to create the composition as described in Equations 2.6 and 2.7. We proof correctness S_D can use α to link the functions. Figure 3.4 shows the composition.

	$\mathsf{Patch}_{f_o \to f_t}$
S_P	$\pi_{f_t} \colon I_{f_t} \to I_{f_t}$ $\sigma_{f_o}^{-1} \colon I_{f_o} \to I_{f_o}$
S_D	$Ret_{f_o \to f_t}^{\pi_{f_t}, \sigma_{f_o}} = Ret_{f_o \to f_i}^{\pi_{f_i}, \sigma_{f_o}} \circ \alpha \circ Ret_{f_i \to f_t}^{\pi_{f_t}, \sigma_{f_i}}$ $Ins_{f_o \to f_t}^{\pi_{f_t}} = \begin{cases} Ins_{f_o \to f_i}^{\pi_{f_i}} \circ \alpha \circ Ret_{f_i \to f_t}^{\pi_{f_t}, \sigma_{f_i}} \\ Ins_{f_i \to f_t}^{\pi_{f_t}} \end{cases}$

Figure 3.4: $\mathsf{Patch}_{f_o \to f_t}$ based on $\mathsf{Patch}_{f_o \to f_i}$ and $\mathsf{Patch}_{f_i \to f_t}$ as composed by $\mathcal{S}$.

$$
\begin{aligned}
Ret_{a \to c}^{\pi_c, \sigma_a} &= Ret_{a \to b}^{\pi_b, \sigma_a} \circ \alpha \circ Ret_{b \to c}^{\pi_c, \sigma_b} && (3.13)\\
&= Ret_{a \to b}^{\pi_b, \sigma_a} \circ \alpha \circ (\sigma_b \circ Ret_{b \to c} \circ \pi_c^{-1}) && (3.14)\\
&= Ret_{a \to b}^{\pi_b, \sigma_a} \circ \pi_b \circ \sigma_b^{-1} \circ \sigma_b \circ Ret_{b \to c} \circ \pi_c^{-1}) && (3.15)\\
&= (\sigma_a \circ Ret_{a \to b} \circ \pi_b^{-1}) \circ \pi_b \circ Ret_{b \to c} \circ \pi_c^{-1}) && (3.16)\\
&= \sigma_a \circ (Ret_{a \to b} \circ Ret_{b \to c}) \circ \pi_c^{-1} && (3.17)\\
&= \sigma_a \circ Ret_{a \to c} \circ \pi_c^{-1} && (3.18)
\end{aligned}
$$

We see that by inserting α between the two *Ret* functions we obtain the composed *Ret* function. To achieve this we use Equation 3.2. Both in substituting the *Ret* functions and observing that we arrive at the correct value. Furthermore, we use Equation 2.6 to determine that we actually create the composition of the underlying *Ret* functions.

In the same fashion we can determine the insertions from the $\mathsf{Patch}_{a \to b}$ are still present in $\mathsf{Patch}_{b \to c}$. Recall that a composed *Ins* function consists of two parts. Equation 2.7 shows the two parts. We prove the correctness of the first case. The second case directly translates to a permuted setting. We will show that the first line also correctly forms with the use of α. Since we only discuss the first line of the composed *Ins* we use ${}^1Ins_{a \to c}^{\pi_c}$ to indicate we refer only to the first line of Equation 2.7. The retained insert is:

$$
\begin{aligned}
{}^{1}Ins^{\pi_c}_{a\to c} &= Ins^{\pi_b}_{a\to b} \circ \alpha \circ Ret^{\pi_c,\sigma_b}_{b\to c} && (3.19)\\
&= Ins^{\pi_b}_{a\to b} \circ \alpha \circ (\sigma_b \circ Ret_{b\to c} \circ \pi_c^{-1}) && (3.20)\\
&= Ins^{\pi_b}_{a\to b} \circ (\pi_b \circ \sigma_b^{-1}) \circ \sigma_b \circ Ret_{b\to c} \circ \pi_c^{-1} && (3.21)\\
&= (\mathrm{AES}^{IV}_k \circ Ins_{a\to b} \circ \pi_b^{-1}) \circ \pi_b \circ Ret_{b\to c} \circ \pi_c^{-1} && (3.22)\\
&= \mathrm{AES}^{IV}_k \circ (Ins_{a\to b} \circ Ret_{b\to c}) \circ \pi_c^{-1} && (3.23)\\
&= \mathrm{AES}^{IV}_k \circ ({}^{1}Ins_{a\to c}) \circ \pi_c^{-1} && (3.24)
\end{aligned}
$$

We see that the final line is part of a permuted Ins function as given in Equation 3.3. The other part of the equation is the already existing $Ins^{\pi_c}_{b\to c}$ function. We see that the whole function has the form we give in Equation 3.3 in the following:

$$
Ins^{\pi_c}_{a\to c} = \begin{cases} \mathrm{AES}^{IV}_k \circ {}^{1}Ins_{a\to c} \circ \pi_c^{-1} \\ \mathrm{AES}^{IV}_k \circ Ins_{b\to c} \circ \pi_c^{-1} \end{cases} \tag{3.25}
$$

This results in the new $Ins^{\pi_c}_{a\to c}$ as;

$$
Ins^{\pi_c}_{a\to c} = \begin{cases} Ins^{\pi_b}_{a\to b} \circ \alpha \circ Ret^{\pi_c,\sigma_b}_{b\to c} \\ Ins^{\pi_c}_{b\to c} \end{cases} \tag{3.26}
$$

Beside the newly created description, S_{P} needs to store the correct permutations. As we can observe from the new Ret function we require σ_a as our origin permutation. Furthermore, since we are still working on c we require π_c for the target permutation. Figure 3.4 provides an overview of the composed patch files stored on S_{D} and S_{P}.

Combining all the information results in the Secure Compose function. Figure 3.5 gives the protocol to perform the function.

3.3.4 Analysis

We propose a new way of formatting a patch file. We do this by randomly creating two permutations that work on the index sets of the origin and the target file (I_a, I_b). We store the permuted description separately from the permutations themselves. Since the servers in $\mathcal{S}$ are non-colluding, this means the server that holds the permuted patch description, S_{D}, does not learn the permutation. Conversely,

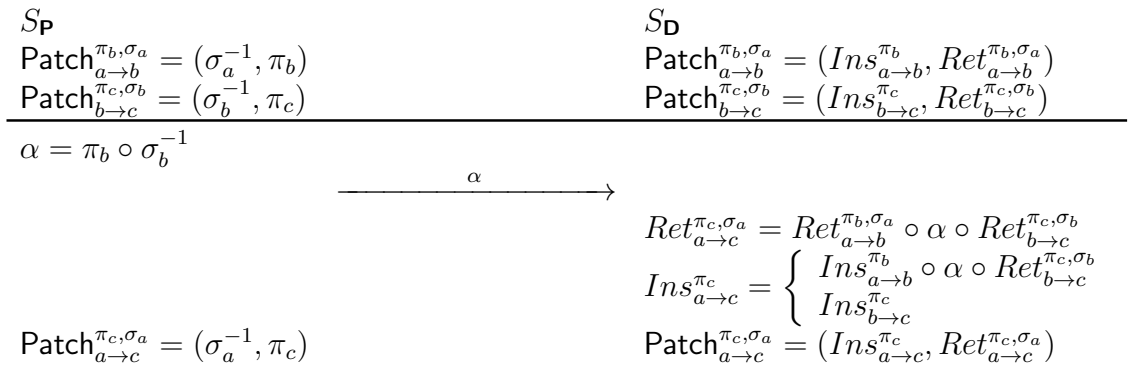


Figure 3.5: Depiction of the Secure Composition Protocol.

the server that stores the permutations, S_{P}, does not learn the permutated description. Since neither server has access to both the pieces of information, neither of the servers can infer the original patch file.

While $\mathcal{S}$ cannot determine the target or origin line number of an operation, it can deduce the size of both files. Additionally, it can determine how many retentions and insertions a patch file describes. These values derive from the size of the permutations and the size of the descriptions, respectively. To hide this, we introduce a form of padding. The padding makes all files have the same size.

Furthermore, since we set the number of retentions and insertions to a fixed number, we hide the number of operations and the file size. The specified number of operation causes a padded file to be twice the size of the content. These files are less flexible as we can no longer grow files to any desired length and need to work in the set bounds or reveal we need more space by once increasing the size of the files.

If the servers in $\mathcal{S}$ do collude, they can learn the original lines in the patch file. Learning the original line numbers means that the Ret and Ins functions no longer hide the origin and target line numbers. $\mathrm{Im}(Ins)$ is still encrypted. This situation is the same situation we describe in Section 3.2.1 with one distinction, the patch file is still padded. Since a padded patch file contains fake operations, we do not know which operations are real and fake. However, we are certain that the line numbers contained in a function are part of that function. Thus if the entities in $\mathcal{S}$ collude, they learn the actual line numbers. The patch still hides other information. The length of files, number of operations, and content are still hidden. The former two remain hidden due to the padding, the latter hold under the encryption we use on

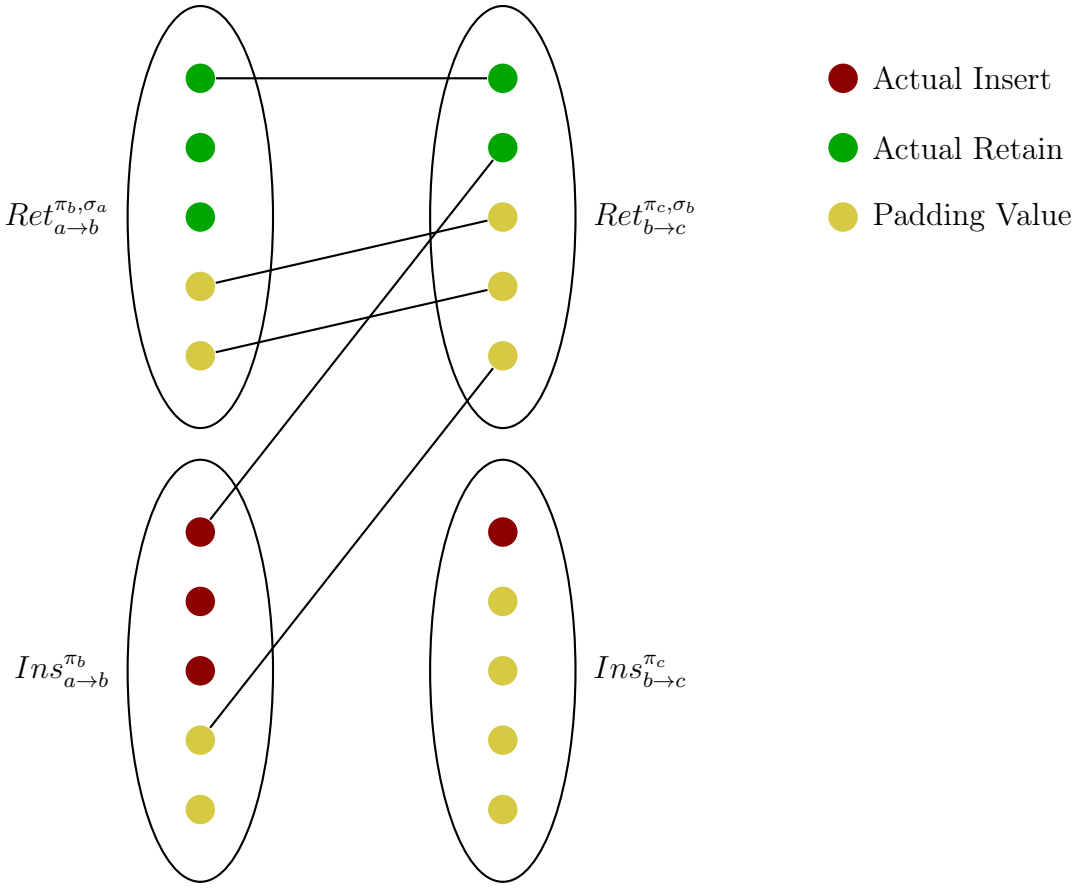


Figure 3.6: Visualisation of composition of two padded permuted patch files – $\mathsf{Pad}(\mathsf{Patch}^{\pi_b,\sigma_a}_{a\to b}), \mathsf{Pad}(\mathsf{Patch}^{\pi_c,\sigma_b}_{b\to c})$. It is the case that $|Ret_{a\to b}| > |Ret_{b\to c}|$ for $|\mathsf{Pad}(Ret^{\pi_b,\sigma_a}_{a\to b})| = |\mathsf{Pad}(Ins^{\pi_b}_{a\to b})| = |\mathsf{Pad}(Ins^{\pi_c}_{b\to c})| = |\mathsf{Pad}(Ret^{\pi_c,\sigma_b}_{b\to c})| = 5$. We see that since the inequality holds the composed Ret function shrinks, while the composed Ins function grows by the same amount.

the insertion content.

Composing permuted patch files is possible. S_P creates a new permutation and sends this permutation to S_D. This new permutation is $\alpha = \pi_{f_i} \circ \sigma^{-1}_{f_i}$. $\mathfrak{S}_{|f_i|}$ is a group under composition of permutations. Since $\pi_{f_i}, \sigma_{f_i} \in \mathfrak{S}_{|f_i|}$ we have $\alpha \in \mathfrak{S}_{|f_i|}$.

Analogously to composing permuted patch files, we can also compose padded permuted patch files. However, in composing two padded patch files, we may reveal information. The information revealed is inherent to the nature of the composition of patches as described in Equations 2.6 and 2.7. Figure 3.6 visualises the problem we run into in the situation where the second patch file has fewer actual retain operations than the first patch file. In this case, we cannot obtain enough padded

	n				
	10	100	1000	10000	100000
3.3.1	0.0002s	0.0014s	0.0140s	0.1400s	1.4001s
3.3.2	0.0002s	0.0020s	0.0198s	0.1985s	1.9800s

Table 3.2: Timing results in seconds for the non-colluding two-server setting. Note that we time the composition so we disregard encryption time.

retained lines. This means Ins grows by $|Ret_{a\to b}^{\pi_b,\sigma_a}| - |Ret_{b\to c}|$ and Ret shrinks by the same amount. Thus we do not simulate $\mathcal{F}_{\mathsf{sCompose}}$ entirely.

Timing

We test the timing in a real-world scenario. We simulate the two different servers by separating objects on the same machine. Only limiting the separation to objects means that the communication time is lower than we would expect in actuality. We compose two files multiple times and report the result. We only time the time it takes for two patch files to compose. We do not consider any encryption or creating of patch files in the timing. Table 3.2 gives the time for different number of compositions. The patch files consist of 30 lines of which 17 are retentions and 13 are new insertions. The insertions are lines of 100 elements. A padded file is twice the size of a normal file. The test machine is equipped with a AMD EPYC 7V12 CPU and 28GB of RAM.

Once again the time scales with the number of operations. Furthermore, we observe that the padded composition takes about twice as long as the unpadded variant. This timing difference is logical since the padded version is twice as long as the normal version. Thus the composition has twice as many lines to compose.

3.4 Conclusion

We propose multiple approaches to perform Secure Patch Composition – each with its limitations and capabilities. If the only goal is to hide the content of the lines, we can use a symmetric-key encryption scheme to hide said content. While this is effective at suppressing the content, it still leaves meta-data in the open. This data includes the type of operation for a given line in the target file, the number of retentions and insertion, and the length of the files concerned.

To hide the meta-data, we propose a naive approach that hides everything in a patch file. In doing so, we lose the capability to compose an effective patch file in any non-trivial case. To circumvent this, we introduce a new way of storing a patch file that consists of a permuted patch description stored on S_{D} and the associated permutations on S_{P}. This allows for the composition of effective patches.

We hide the number of operations by setting an overall size limit on the files. This file limit is twice the length of the number of retentions and insertions, where the number for both operations is equal. This number is also the maximum size of any content. All files have the same length. Thus we create effective patch files since all our patch files are, by definition, no longer than the target file. However, the size of the actual content is at most half of the patch file. A more extended file than the content means, with regards to the content, that we cannot create an effective patch file. This latter observation is the case for any system using a fixed file size with a content size that can be less than the size of the file.

In padding the files, we also hide the number of operations. We can no longer determine how many insertions or retentions are part of a patch file. However, in composing two patch files, we learn something about the number of retentions from one patch to another. We learn how many fewer retentions there are in the second patch file. We still do not learn anything about the insertions other than that we need the insertions to fill the set file size. Figure 3.6 shows this phenomenon.

We summarise all proposed functions in Table 3.3. We note that the approaches in Sections 3.2.1 and 3.3.1 are unpadded and that the approaches in Sections 3.2.2 and 3.3.2 are padded.

Sec.	Simulates $\mathcal{F}_{\mathsf{sCompose}}$	Effective Compose	Trust Assumption	Communication Complexity
		Single Server		
3.2.1	Leaks Meta-Data	Yes	Honest-but-Curious	0 rounds
3.2.2	Yes	No	Honest-but-Curious	0 rounds
		Non-Colluding Two-Servers		
3.3.1	Leaks Number of Lines and Operations	Yes	Honest-but-Curious, Non-Colluding	1 round
3.3.2	Leaks Number of Retentions	Yes	Honest-but-Curious, Non-Colluding	1 round

Table 3.3: Summary of sCompose functions and their properties. We show the protocols suggested in Sections 3.2 and 3.3.

Chapter 4

Privately Aligning Sequences

In both `diff` and `diff3` we require alignment of sequences. Autexier achieves the alignment of two sequences using the Longest Common Subsequence (LCS) [Aut15]. We can use these functions to strengthen a secure repository further. Additionally, we can also use them as a basis for a merge algorithm that can securely merge data from different sources. Securely merging could be relevant to data aggregation in secure settings.

This chapter investigates how we can solve the LCS problem in our security setting. Our security setting, in this case, requires the server executing the function, $\mathcal{S}$, to receive two inputs and produce the LCS while itself not learning anything about the LCS. We call the procedure Secure LCS, also noted as sLCs. This chapter describes the ideal functionality, security setting and suggests two approaches. The first is an interactive protocol. The second is an adaptation of this protocol.

We first discuss the security setting and the ideal functionality. We discuss the applicability of the state-of-the-art to our scenario. We then present and analyse our approaches.

4.1 Security Setting and Ideal Functionality

In this scenario, we operate under a more general assumption. In this setting, we say that the set of receivers of the output, $\mathcal{R}$, and the set of data providers, $\mathcal{P}$, can share elements but are not necessarily equal. The more general assumption is vital for more fine-grained repository structures, such as sub-modules, and for merging data for a specific target that owns the data but is not the processor.

Functionality $\mathcal{F}_{\mathsf{sLCS}}$

Participants: $p, p' \in \mathcal{P}$ for the set of providers $\mathcal{P}$, $R \subseteq \mathcal{R}$ for the receiver set $\mathcal{R}$.
Parameters: LCS Function, LCS$(\cdot, \cdot)$ for files from $\mathbb{F}$.
Input:
- $p : f_p \in \mathbb{F}$,
- $p' : f_{p'} \in \mathbb{F}$,

Output:
- p obtains LCS$(f_p, f_{p'})$ if $p \in R$ else $\perp$
- p' obtains LCS$(f_p, f_{p'})$ if $p' \in R$ else $\perp$
- R obtains LCS$(f_p, f_{p'})$

Figure 4.1: Ideal Functionality for the Secure Longest Common Subsequence function.

With this function we wish to determine the LCS in such a way that we can use it to perform sDiff and sDiff3. These two functions are the topic of the next chapter. To perform the sLCS means that a third party, $\mathcal{S}$, can determine the LCS without learning about the underlying data.

We have two participants $p, p' \in \mathcal{P}$, where $p = p'$ is possible. However, in the case of $p = p'$, it would be more effective for the participant to perform their LCS function locally on the plaintext files. We consider a single server scenario, so $\mathcal{S} = \{S\}$, and the receivers own a shared public-secret key pair. We give the ideal functionality for the Secure LCS, noted as $\mathcal{F}_{\mathsf{sLCS}}$, in Figure 4.1.

4.2 Secure LCS Calculation

Franklin et al. propose a method to privately determine the LCS [FGM09]. This method determines the length of the LCS and provides backtracking methods to derive an encrypted embedding of or the whole LCS. This private approach serves as a basis for our secure approach. Note that in a private setting, two parties jointly compute the LCS between their inputs. In our setting of Secure LCS, we let a third party determine the LCS while not learning it.

Franklin et al. base themselves on Masek and Patterson [MP80] who propose a method to determine the LCS based on a technique presented by Arlazarov, Dinic,

Kronod, and Faradzev [ADKF70]. This approach is commonly referred to as the "Four Russian" technique after the nationality and workplaces – Moscow at the time – of the four authors. The four Russians method can speed up algorithms that involve binary matrices.

We note that in the case of files, this approach will not pose any real-world benefits. It assumes a fixed alphabet size. This limitation is feasible in the case we are comparing words or sentences. The alphabet could be the Latin alphabet and some punctuation and whitespaces. However, we work with files. The elements are lines. Lines are a sequence in itself with characters from the alphabet Σ. We cannot precompute all possible blocks since there are infinite possibilities.

Since we cannot precompute blocks, we approach the Secure LCS problem by securely constructing the matrix as per the dynamic programming approach. To enable this, we use 0 as a special symbol to indicate an empty sequence. The empty symbol helps us in later stages to determine the actual LCS while hiding the length and embedding of the LCS from $\mathcal{S}$.

4.3 Sequence Encryption

For a sequence s, with indices in I_s, and public-key $pk_{\mathcal{R}}$ of the receiver we define the encryption of s under $pk_{\mathcal{R}}$ as $E_{\mathcal{R}}(s)$ where the encryption is given as $E_{\mathcal{R}}(s) = (E_{\mathcal{R}}(s_i))_{i \in I}$. The index set of the encrypted sequence is the same as the initial sequence. Since we encrypt a sequence element wise we retain the ability to note substrings for s. They are directly analogous. Thus the encrypted version of the substring s^i for $i \in I_s$ is $E_{\mathcal{R}}(s^i)$. Encrypting a sequence in this way does leak its length. The participants $p, p' \in \mathcal{P}$ use this form of encryption to hide the inputs from $\mathcal{S}$.

4.4 Interactive Protocol

This section expands on an interactive protocol between $\mathcal{S}$ and $\mathcal{R}$. It shows how $p, p' \in \mathcal{P}$ share their files, f_p, $f_{p'}$ respectively, with $\mathcal{S}$ and how $\mathcal{S}$ and $\mathcal{R}$ determine the LCS. With slight adjustments, $\mathcal{S}$ can perform the protocol by itself. However, this requires post-processing after decryption or in-place homomorphic multiplication.

We discuss these adjustments in later sections.

4.4.1 Interactive Equality

Equality over encrypted elements is not as straightforward as its plaintext counterpart. For our interactive approach, we use an interactive protocol suggested by Nateghizad et al. [NVEL18]. The authors use two cryptosystems in their equality protocol. These are Paillier [Pai99] and DGK [DGK07]. The latter finds its merit for this application in its efficient zero-check and small message space, enabling low communication cost.

Paillier. Paillier Encryption has a message space of $\mathbb{Z}_n$ where n is the product of two large prime numbers p and q for which $p \neq q$. The Paillier scheme is a partially homomorphic scheme that supports homomorphic addition. The encryption using Paillier is $E_{pk_{\mathcal{R}}}(m;r) = g^m \cdot r^n \mod n^2$. Recall that $E_{\mathcal{R}}(m)$ is a short-hand $E_{pk_{\mathcal{R}}}(m;r)$. In the encryption g is a generator of $\mathbb{Z}_n^*$ and $r \in_R \mathbb{Z}_n^*$. The public-key is (g,n) and the private key is (p,q). In line with [NVEL18] $[\cdot]$ indicates a Paillier encrypted message.

DGK. In DGK, for a DGK security parameter t, we generate keys by first choosing two different t-bit prime numbers. We denote these prime numbers as v_p and v_q. Using these values we determine two prime numbers p and q such that $v_p \mid p$ and $v_q \mid q$ and $n = p \cdot q$. We choose u to be the smallest possible prime number greater than $\ell+2$, where ℓ is the input length. Choose a random integer, r, that is moderately bigger than $2 \cdot t$, $2.5 \cdot t$ is sufficient. Finally, we have two values g and h where the multiplicative order for g is $u \cdot v_p \cdot v_q$ and for h is $v_p \cdot v_q$. The public-secret keypair now is $pk = (n, g, h, u)$ and $sk = (p, q, v_p, v_q)$. In line with [NVEL18] $[\![\cdot]\!]$ indicates a DGK encrypted message.

EQT-1

This protocol is a two-party protocol that returns a Paillier encrypted bit indicating the equality of two elements. It determines this by either computing the Hamming Distance or performing a secure comparison based on the work by Dåmgrad et al. [DGK07]. The method requires two different approaches because only using one

would reveal the result to the second party. To prevent this, the first party tosses a coin to decide between the Hamming Distance and the Comparison approach. We sketch EQT-1 in Protocol 1. ℓ denotes the length of the input in bits.

Protocol 1 EQT-1

Require: Two input values a and b.
Ensure: The Paillier encrypted bit representation under the key of the second party.

1: A generates random value r, determines $[x] \leftarrow [a - b + r]$ and sends $[x]$ to B.
2: B decrypts $[x]$ and encrypts the first ℓ bits using DGK to obtain $[\![x_i]\!]$ for $0 \leq i < \ell$.
3: A determines $[\![r_i \oplus x_i]\!]$ for every bit. ▷ In this case $\oplus$ denotes the bit xor.
4: A determines $\delta_A \in_R \{0, 1\}$.
5: **if** $\delta_A = 0$ **then**
6: Determine the Hamming Distance between r and x. This results in a single element. Additionally, generate a list of $\ell - 1$ non-zero random elements smaller than ℓ.
7: **else if** $\delta_A = 1$ **then**
8: Use the DGK Comparison approach on r and x. This results in a list of ℓ encrypted elements.
9: **end if** ▷ Both approaches work since if $a = b$, $x = r$.
10: A randomly permutes the order of the encrypted elements and send this to B.
11: B decrypts and sets $\delta_B = 1$ if at least one of the bits is 0, else $\delta_B = 0$. B sends $[\delta_B]$ to A.
12: A determines $[\vartheta]$. If $\delta_A = 0$, $[\vartheta] = [\delta_B]$. If $\delta_A = 1$, $[\vartheta] = [1] \cdot [\delta_B]^{-1}$.

We can replace the Paillier Cryptosystem with any that supports homomorphic addition. Furthermore, the same holds for the DGK. However, values encrypted using DGK do not interact with any part outside of the confines of the EQT-1 protocol. The Paillier encrypted values do as these are the input and output of the protocol. Suppose we wish to use the algorithm's result in any other system we only need to change the Paillier system.

4.4.2 Protocol

Using EQT-1, we construct an interactive protocol between $\mathcal{S}$ and $\mathcal{R}$ that securely determines the LCS. To this extent, we use a different encryption scheme in the EQT-1 Protocol. Instead of Paillier, we use the Fully Homomorphic Encryption Scheme proposed by Cheon, Han, and Kim [CHK20]. Note that this scheme could be any scheme that supports homomorphic operations that are useful in the further

application of the LCS. It is application dependent whether these operations are limited to homomorphic addition or require homomorphic multiplication as well. Since the result of the LCS will possibly be used in a later stage for `diff` and `diff3` we choose a scheme that allows for both operations.

CHK Cheon, Han, and Kim built on the bootstrapping technique proposed by Nuida and Kurosawa [NK15]. They apply their new bootstrapping technique to the CLT homomorphic encryption scheme [CLT14]. The message space is $\mathbb{Z}_p$ for prime p the definition of the cryptosystem include the homomorphic Addition, Convert, and Multiplication function. We note encryption under this scheme as $(\!|\cdot|\!)$.

We now have a scheme that supports homomorphic multiplication ($\otimes$) and homomorphic addition ($\oplus$). Protocol 2 gives the scheme we propose. This procedure does not leak the length of the LCS. As in every operation for every possible LCS we determine the new LCS.

Protocol 2 Two-Party sLCS

Require: $p, p' \in \mathcal{P}$ provide files f with index set I_p, and f' with index set $I_{p'}$, respectively.
Ensure: $\mathcal{R}$ learn the Longest Common Subsequence
1: p determines $(\!|f|\!)_{\mathcal{R}}$, p' determines $(\!|f'|\!)_{\mathcal{R}}$. They send $(\!|f|\!)_{\mathcal{R}}$ and $(\!|f'|\!)_{\mathcal{R}}$ to $\mathcal{S}$.
2: $\mathcal{S}$ initialises matrix M of size $|I_p| + 1 \times |I_{p'}| + 1$ with all values as $\{[0]_{\mathcal{R}}\}$.
3: **for** $i \in I_p$ **do** ▷ $I_p = \{1, \ldots, |f|\}$
4: **for** $j \in I_{p'}$ **do** ▷ $I_{p'} = \{1, \ldots, |f'|\}$
5: $\mathcal{S}$ performs the EQT-1 protocol with $\mathcal{R}$ for the i^{th} element of f and the j^{th} element of f' such that $\vartheta \leftarrow$ EQT-1(f_i, f'_j).
6: $\mathcal{S}$ determines the next element in the LCS by $c \leftarrow f_i \otimes \vartheta$.
7: $\mathcal{S}$ determines $TL = \{m \,||\, c \mid m \in M_{i-1,j-1}\}$
8: $\mathcal{S}$ determines $T = \{[0]_{\mathcal{R}} \mid m \in M_{i-1,j}\}$
9: $\mathcal{S}$ determines $L = \{[0]_{\mathcal{R}} \mid m \in M_{i,j-1}\}$.
10: $M_{i,j} = TL \cup T \cup L$
11: **end for**
12: **end for**

4.5 Non-Interactive Protocol

In the previous section, we sketch a protocol where $\mathcal{S}$ and $\mathcal{R}$ work together to obtain the LCS. This section depicts two protocols that allow $\mathcal{S}$ to do this themselves.

4.5.1 Non-Interactive Equality

Equality testing of encrypted elements is a problem that knows many approaches. There exist multi-party protocols that let no party learn any of the values compared, nor the result [NVEL18, SK16], we use such a method in the interactive variant. Furthermore, public-key encryption schemes exist that allow for equality testing for an entity with a trapdoor function [LLS+20, ZCL+19]. These schemes are called Public Key Encryption with Equality Test (PKEET). Contrary to its interactive counterparts, PKEET schemes let one party perform the equality test.

In our scenario, we wish that an external party, $\mathcal{S}$, can determine the equality of two elements independently without learning the outcome or the input of the test. If they did learn something about the inputs or the output, it would enable them to learn something about the Longest Common Subsequence.

Since $\mathcal{S}$ cannot learn anything, PKEET schemes are not helpful as they reveal the outcome of the equality test. Furthermore, we want $\mathcal{S}$ to perform the function; thus, an interactive protocol can only take place between members of $\mathcal{S}$.

We also look at a different solution. This solution requires post-processing to obtain the final result but does not require communication or a trapdoor function. We cite the following theorem:

Theorem 4.5.1 (Fermat's Little Theorem [Lis05])**.** If p is a prime number and a is any number not divisible by p, then $a^{p-1} \equiv 1 \mod p$.

This theorem allows us to create an equality test. For any two number in $a, b \in \mathbb{Z}_p, a \neq b$ for prime p we have $(a-b)^{p-1} = 1$. While, if $a = b$ we have $(a-b)^{p-1} = 0$.

We can even extend this beyond the necessity where p is a prime. Fermat's Little Theorem is a particular case of the Euler's Totient function [Kal05]. For a given integer n, the Totient function determines the number of co-primes with n smaller than n. We say two numbers are coprime if the Greatest Common Divisor (GCD) of the two numbers is 1. GCD(a, b) notes the GCD for the numbers a and b.

Definition 4.5.1 (Euler's Totient Function [Kal05])**.** $\phi(n)$ gives the order of the multiplicative group $\mathbb{Z}_n$. If n is prime $\phi(n) = n - 1$. If $n = \prod_{i=1}^{d} p_i^{k_i}$, where $d > 2$, $p_1, \ldots, p_d$ are distinct primes, and $k_i \geq 1$, the totient function is given as:

$$\phi(n) = \prod_{i=1}^{d} \phi(p_i^{k_i}) \tag{4.1}$$

The Totient function gives the order of a group. In a group, raising an element to the power of the group's order results in one. Here it is clear that Fermat's Little Theorem is a special case of the Totient function since for all $n \in \mathbb{Z}_p^*$ we have $\text{GCD}(n, p) = 1$, for prime p. Thus $\phi(p) = p - 1$ for prime p.

Instead of $\phi(n)$ we could use $\lambda(n)$. The latter is defined to give the smallest number such that $x^{\lambda(n)} \equiv 1 \mod n$.

The more general case of Fermat's Little Theorem is helpful in cases where the message space is not of prime order. This property means we can still use schemes such as Paillier.

4.5.2 Protocol

We sketch two non-interactive protocols. Both protocols share that we only adept line 5 through 9 with regards to Protocol 2.

In Place

Using Fermat's Little Theorem and the Fully Homomorphic Encryption sheme described in the previous section we determine the equality of two element locally by $\mathcal{S}$. We achieve this by the following formula:

$$\bigotimes^{p-1}(E_{\mathcal{R}}(m) \ominus E_{\mathcal{R}}(m')) \tag{4.2}$$

Where the encryption operation denotes the scheme proposed in [CHK20], $\bigotimes^{p-1}$ are $p - 1$ homomorphic multiplications, and $\ominus$ is the homomorphic substraction operation. Note that this operation is expensive as we perform $p - 1$ homomorphic multiplications. Protocol 3 reflects the use of this approach.

Post Processing

This protocol eliminates the need for in-place multiplication to achieve an equality result. Eliminating the in-place multiplications comes at the cost of revealing the content of one of the file to $\mathcal{R}$. This approach could be useful when $p \in \mathcal{P} \cap \mathcal{R}$.

We eliminate the need to perform the multiplication in place by making use of a post-processing step. We determine $E_{\mathcal{R}}(m) \ominus E_{\mathcal{R}}(m')$ and store this along side the element m. Once $\mathcal{R}$ decrypts the LCS they can disregard all values for which

Protocol 3 In-place Non-Interactive sLCS

Require: $p, p' \in \mathcal{P}$ provide files f with index set I_p, and f' with index set $I_{p'}$, respectively.
Ensure: $\mathcal{R}$ learns the Longest Common Subsequence

p determines $(\!|f|\!)_{\mathcal{R}}$, p' determines $(\!|f'|\!)_{\mathcal{R}}$. They send $(\!|f|\!)_{\mathcal{R}}$ and $(\!|f'|\!)_{\mathcal{R}}$ to $\mathcal{S}$.
$\mathcal{S}$ initialises matrix M of size $|I_p| + 1 \times |I_{p'}| + 1$ with all values as $\{[0]_{\mathcal{R}}\}$.
for $i \in I_p$ **do** ▷ $I_p = \{1, \ldots, |f|\}$
  for $j \in I_{p'}$ **do** ▷ $I_{p'} = \{1, \ldots, |f'|\}$
    $\mathcal{S}$ determines $\vartheta \leftarrow (\!|f_i - f'_j|\!)_{\mathcal{R}}$.
    $\mathcal{S}$ determines the next element in the LCS, $c \leftarrow f_i \otimes (\bigotimes^{p-1} \vartheta)$.
    $\mathcal{S}$ determines $TL = \{m \,||\, c \mid m \in M_{i-1,j-1}\}$
    $\mathcal{S}$ determines $T = \{[0]_{\mathcal{R}} \mid m \in M_{i-1,j}\}$
    $\mathcal{S}$ determines $L = \{[0]_{\mathcal{R}} \mid m \in M_{i,j-1}\}$.
    $M_{i,j} = TL \cup T \cup L$
  end for
end for

$D_{\mathcal{R}}(E_{\mathcal{R}}(m) \ominus E_{\mathcal{R}}(m')) \neq 0$. Since we no longer require multiplications we can even use any scheme that support homomorphic subtraction, such as Paillier. We depict the protocol, using Paillier, in Protocol 4.

Protocol 4 Post Processing Non-Interactive sLCS

Require: $p, p' \in \mathcal{P}$ provide files f with index set I_p, and f' with index set $I_{p'}$, respectively.
Ensure: $\mathcal{R}$ learns the Longest Common Subsequence

p determines $[f]_{\mathcal{R}}$, p' determines $[f']_{\mathcal{R}}$. They send $[f]_{\mathcal{R}}$ and $[f']_{\mathcal{R}}$ to $\mathcal{S}$.
$\mathcal{S}$ initialises matrix M of size $|I_p| + 1 \times |I_{p'}| + 1$ with all values as $\{([0]_{\mathcal{R}}, [0]_{\mathcal{R}})\}$.
for $i \in I_p$ **do** ▷ $I_p = \{1, \ldots, |f|\}$
  for $j \in I_{p'}$ **do** ▷ $I_{p'} = \{1, \ldots, |f'|\}$
    $\mathcal{S}$ determines $\vartheta \leftarrow [f_i - f'_j]_{\mathcal{R}}$.
    $\mathcal{S}$ determines $TL = \{(m, \vartheta) \mid m \in M_{i-1,j-1}\}$
    $\mathcal{S}$ determines $T = \{([0]_{\mathcal{R}}, [0]_{\mathcal{R}}) \mid m \in M_{i-1,j}\}$
    $\mathcal{S}$ determines $L = \{([0]_{\mathcal{R}}, [0]_{\mathcal{R}}) \mid m \in M_{i,j-1}\}$.
    $M_{i,j} = TL \cup T \cup L$
  end for
end for

We can obtain the binary result of the subtraction in both cases. For the scheme proposed in [CHK20], we note that the message space is $\mathbb{Z}_p$ for a prime p. Thus, we can use Fermat's Little Theorem to obtain 1 in all cases that are not 0. The Paillier cryptosystem has the message space $\mathbb{Z}_n$ where $n = p \cdot q$ for two large prime

p and q. Since the post processing happens at $\mathcal{R}$, the owner of the private key of the encryption scheme, we can use $\phi(n)$ without the risk of the cryptosytem losing its effectiveness. We can determine $\phi(n) = \phi(p) \cdot \phi(q) = (p-1) \cdot (q-1)$. Once again, we can now verify that all values besides 0 equal to 1.

4.6 Analysis

The methods proposed above are both build on the dynammic programming approach to the Longest Common Subsequence (LCS) problem.

The procedures suggested above maintain every possible path. Tracking all possible paths ensures we do not leak any information as to which path we are choosing. From the procedure we observe that we have three possible path per operation. These are, in accordance with the steps on lines 6 through 8 of Protocol 2; (a) Diagonal Step - $M_{i-1,j-1}$; (b) Vertical Step - $M_{i-1,j}$; (c) Horizontal Step - $M_{i,j-1}$.

Since the pathing is limited to these three steps, we can express the number of sequences stored at (m, n) as $D(m, n)$. The total number of sequences for a whole table is the summation of possible paths to all coordinates and is given as:

$$\sum_{i=0}^{m} \sum_{j=0}^{n} D(i, j). \tag{4.3}$$

We can reduce the size of the total table. We can reduce the size by disregarding values we no longer require. This limits the maximum size of the table to $D(m, n)$. However, we cannot reduce the number of sequences at (m, n). Reducing the number of sequence at (m, n) would entail reducing the viable paths to said coordinate. With every such reduction we learn more about the LCS as we learn which path it is not. Furthermore, learning which path to take reveals the result of the equality check we use in all protocols.

Kiselman [Kis12] describes the lower- and upper bound for any Delannoy number as:

$$3^{\min(x,y)} \leq D(x, y) \leq (\sqrt{2}+1)^{x+y}, (x, y) \in \mathbb{N}^2. \tag{4.4}$$

For the square Delannoy number, where $D(x, x)$, the worst case, Kiselman gives the

bounds as:

$$3^x \leq D(x,x) \leq (3+\sqrt{8})^x, x \in \mathbb{N}. \tag{4.5}$$

The lower bound is exponential in the smallest input size. This complexity makes it infeasible to work with in real world scenarios. Especially since in further steps we would need to consider each possibility.

The method could be useful in scenarios where we compare a short sequence with a long sequence. However, in most cases, the two sequences will approximately be of the same length. Figure 4.2 visualises the growth rate of the lower and upper bounds.

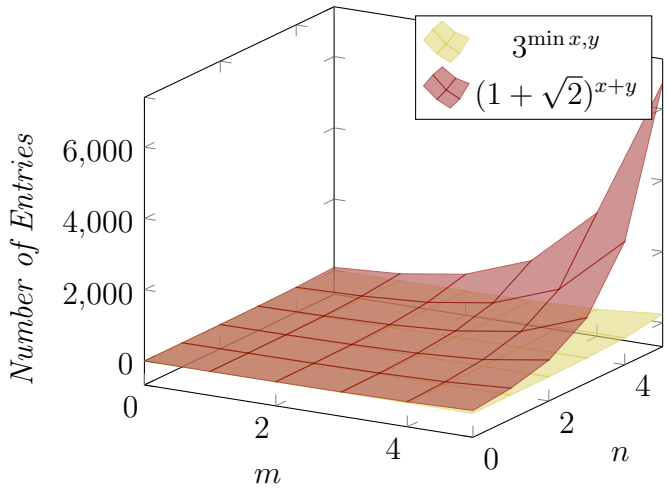


Figure 4.2: Depiction the lower and upper bound of the $D(m,n)$.

We argue that the protocols in Protocol 2 and 3 reveal all equal elements, thus they do not simulate the ideal functionality. In either case the equality function does not reveal anything to either $\mathcal{S}$ or $\mathcal{R}$. In Protocol 2 this is by virtue of the equality proposed by Nateghizad et al. In Protocol 3 this is by virtue of the Fully Homomorphic Encryption cryptosystem. Thus in the communication, or calculation step $\mathcal{S}$ does not learn anything about the LCS. Furthermore, both the protocols send over the last entry of the matrix, $M_{m,n}$, to $\mathcal{R}$. In both cases the entries in $M_{m,n}$ result in $\mathcal{S}$ not learning anything about the LCS. However, the final information that $\mathcal{S}$ sends to $\mathcal{R}$ includes the encryption of all paths. This means that we also include all the paths that contain equal elements. For the execution of LCS("$badf$","$adfb$") we get every path containing equal elements. This will include the LCS, "adf", and

	m				
n	5	10	15	20	25
5	0.0003s	0.0082s	0.0795s	0.4314s	1.5400
10	-	1.9162s	ME	ME	ME
15	-	-	ME	ME	ME
20	-	-	-	ME	ME
25	-	-	-	-	ME

Table 4.1: Timing results in seconds for standard Longest Common Subsequence while storing all possible path. The inputs are two completely distinct strings. In the table, m gives the length of the first input and n gives the length of the second input. ME indicates a memory error.

all its subsequences. However, it will also contain the subsequence "b". This is not part of the LCS. Thus this will reveal more information than only the LCS.

Protocol 4 does not simulate the ideal functionality since in the post processing step decrypting all results in $M_{m,n}$ cause $\mathcal{R}$ to learn the content of one entire sequence.

4.6.1 Timing

To get the lower bound of the timing for the proposed algorithm we use the normal LCS function. However, we store all possible paths. This means that all our suggested approaches take at least as long as the timing we present here. Table 4.1 gives the timing of different input length using a basic LCS implementation. Normal files contain more than 25 lines. This means that the given values are not realistic. They illustrate that the suggested methods are not feasible in a real world scenario. The test machine is equipped with a AMD EPYC 7V12 CPU and 28GB of RAM.

4.6.2 Learning from Equality

To solve the Longest Common Subsequence problem, at a certain point every element of one sequence will be compared to every element in the other sequence. Methods like the Four Russian's allow these steps to be part of a pre-computed process, others do this in place. The equality of two elements dictates the path we take. This means we cannot circumvent the equality check as given in the Definition 2.2.1 ($s_i = r_j$).

As we show earlier, secure equality checking has multiple approaches. These can

be interactive [NVEL18] or based on a trapdoor functions [LLS+20]. We have seen that this last scheme does not offer the functionality we desire. We wish that $\mathcal{S}$ does not learn the outcome of the equality functions whether they are interactive or not.

In the case of an interactive protocol we see that we can use this property to construct a protocol that determines the LCS, this is Protocol 2. Extending this, we let $\mathcal{S}$ determine equality on its own and use this result in decision making, sketched in Protocols 3 and 4.

The problem lies in reducing the size of the final entry in the matrix. We can minimize this value by excluding certain pathings. If the protocol can disregard irrelevant paths we end up with a smaller set of possible Longest Common Subsequences. While we have a third party that can determine the equality of two elements we cannot solve the problem that we need to make a decision based on the equality result. This is the case since one party cannot obfuscate the decision path for itself. Thus in a scenario where one party needs to make a decision based on the result of an equality test, that party will learn the outcome of the equality test.

In our case we have a binary path. When the result of the equality test is positive we take a diagonal step, in the case of inequality we go either to the top or to the left. Consequently, we can learn something about the result of our equality test if we know something about the pathing we took.

If we can determine the path of the algorithm with a probability greater than $\frac{1}{2}$, we can do the same with the result of the equality and vice versa. Reveaowering any of this information will help an adversary in determining the Longest Common Subsequence. Thus, pruning the pathing of the LCS will reveal something about the LCS.

4.7 Conclusion

We propose multiple functions for Secure Longest Common Subsequence. All of the function have the same space complexity. The space complexity of these functions scales with the Delannoy number. The lower bound of the required space is exponential in the length of the shortest input. This space complexity makes it unsuitable for real-world applications.

We propose an interactive function that requires $\mathcal{S}$ and $\mathcal{R}$ to work together for $\mathcal{S}$ to obtain the LCS matrix. The two other approaches do not require the cooperation

	Simulates $\mathcal{F}_{\mathsf{sLCS}}$	Space Complexity	Communication Complexity	Number of $\otimes$
2	Leaks All Equal Elements	$D(m,n)$	$m \cdot n \cdot$ EQT-1	$m \cdot n$
3	Leaks All Equal Elements	$D(m,n)$	0	$m \cdot n \cdot (p-1)$
4	Leaks One Sequence	$D(m,n)$	0	0

Table 4.2: Summary of sLCS functions and their properties. We compare Protocols 2, 3, and 4.

of $\mathcal{R}$ for $\mathcal{S}$ to obtain the matrix. One approach requires $p-1$ homomorphic multiplications, where p is the prime order of the message space of encryption scheme $\mathcal{E}$ that supports the homomorphic multiplication. The other non-interactive approach assumes a post processing step that eliminates the homomorphic multiplication step. This comes at the cost of revealing one of the input sequences to $\mathcal{R}$.

We argue that non of the functions suggested simulate the ideal functionality $\mathcal{F}_{\mathsf{sLCS}}$. The last suggested approach leaks one sequence, the two other suggested approaches leak all identical elements between the two input sequences. We summarise the functions and their properties in Table 4.2.

Chapter 5

Secure Difference Analysis

Autexier bases the proposed function for `diff` and `diff3` on the alignment of two sequences using Longest Common Subsequence (LCS). In the previous section we propose multiple methods to determine the LCS. We also observe that these functions do not support any real-world scenario where we compare files of a larger size. This section describes what the Secure `diff` – sDiff – and Secure `diff3` – sDiff3 – should do in an ideal situation. The following section introduce the terminology and concept further. It is these functionalities that trickle down in the ideal functionality of the Longest Common Subsequence. Since we did not develop any feasible alignment methods this chapter is limited to describing the desired functionality.

5.1 Ideal Functionality

The secure variants of `diff` and `diff3` are variants of the functions introduced by Autexier [Aut15]. They aim to determine the result of the functions while not revealing other information about the input. The inputs for the function that $\mathcal{S}$ executes come from participants $p, p' \in \mathcal{P}$. In the case of `diff` these are any two files. In the case of `diff3` the two input files share a common origin. This common origin, o, is a file that directly predates the two inputs provided by p and p'. In both cases it is possible that $p = p'$.

The goal of sDiff is for a third-party to determine the difference between two hidden files. Only $\mathcal{R}$ should be able to view the result. Furthermore, $\mathcal{R}$ should only learn the result and nothing else. We use the `diff` function to create a patch file. Ideally, sDiff can thus create a secure form of a patch file. Figure 5.1(a) depicts the

Functionality $\mathcal{F}_{\mathsf{sDiff}}$

Participants: $p, p' \in \mathcal{P}$ for the set of providers $\mathcal{P}$, $R \subseteq \mathcal{R}$ for the receiver set $\mathcal{R}$.
Parameters: Difference Function $D : \mathbb{F} \times \mathbb{F} \rightarrow \mathbb{P}$, for files from input-space $\mathbb{F}$ and patch files from output-space $\mathbb{P}$.
Input:
- $p : f_p \in \mathbb{F}$
- $p' : f_{p'} \in \mathbb{F}$

Output:
- p obtains $D(f_p, f_{p'})$ if $p \in R$ else $\perp$
- p' obtains $D(f_p, f_{p'})$ if $p' \in R$ else $\perp$
- R obtains $D(f_p, f_{p'})$

(a)

Functionality $\mathcal{F}_{\mathsf{sDiff3}}$

Participants: $p, p' \in \mathcal{P}$ for the set of providers $\mathcal{P}$, $R \subseteq \mathcal{R}$ for the receiver set $\mathcal{R}$.
Parameters: Three-Way Difference Function $D : \mathbb{F} \times \mathbb{F} \times \mathbb{F} \rightarrow \mathbb{P}$, for files from input-space $\mathbb{F}$ and patch files from output-space $\mathbb{P}$.
Input:
- origin file o.
- $p : f_p \in \mathbb{F}$ s.t. $\mathsf{Patch}_{o \rightarrow f_p}$
- $p' : f_{p'} \in \mathbb{F}$ s.t. $\mathsf{Patch}_{o \rightarrow f_{p'}}$

Output:
- p obtains $D(f_p, f_{p'}, o)$ if $p \in R$ else $\perp$
- p' obtains $D(f_p, f_{p'}, o)$ if $p' \in R$ else $\perp$
- R obtains $D(f_p, f_{p'}, o)$

(b)

Figure 5.1: The Ideal Functionalities for the Secure Difference Analysis. (a) gives the Ideal Functionality for Secure `diff`. (b) gives the Ideal Functionality for Secure `diff3`.

ideal functionality.

Much like sDiff, sDiff3 should allow a third-party to determine the result of `diff3` without learning anything about the content of the input or the output. The result should only be known to $\mathcal{R}$. Note that function requires a third input. This input is not necessarily provided by p or p', this file could already be present on $\mathcal{S}$. Figure 5.1(b) gives the ideal functionality.

We do not sketch a layout for $\mathcal{S}$ as there is no suggested procedure. However, if we wish to keep using the effective patch composition as provided in Section 3.3.3, it is likely that we end up with a non-colluding multi-server setting.

Chapter 6

Conclusion

This thesis explores multiple techniques required to create a secure repository. Specifically, we introduce and consider the concepts of Secure Patch Composition, Secure Longest Common Subsequence (LCS), Secure `diff`, and Secure `diff3`. Secure variants of these functions allow a server hosting a Version Control System (VCS) to perform all necessary function without learning the content it is hosting.

We start in Chapter 3 with introducing the Secure Compose – $\mathsf{sCompose}$ – functionality. We continue this chapter by introducing possible approaches to the problem. We consider two security settings in the suggested approaches. The first security setting consists of a single server setting, $\mathcal{S} = \{S\}$. This setting leaves us with two approaches. The first variant allows for effective composition but leaks all data in the patch file besides the content of the new lines. The new lines are encrypted using AES in GCM Mode. The encryption means only the line content is hidden. $\mathcal{S}$ still learns which lines are new lines, which lines are retained lines, and the number of operations and lines. In a real-world scenario, this approach performs as good as an unencrypted variant as it does not require any extra steps.

We argue that the second approach in the single server setting simulates the ideal functionality $\mathcal{F}_{\mathsf{sCompose}}$. This approach randomly orders the lines in a patch file and pads the files to a fixed size. We encrypt this padded and randomly ordered file using AES in GCM Mode. The encrypted randomly ordered file means that $\mathcal{S}$ does not learn anything about the size of the content or number of operations by observing the files it receives. Furthermore, the random ordering of the lines means $\mathcal{S}$ cannot derive any information about the location of lines. These properties come at the cost of effective composition. The composition is the same as concatenation

in this approach. In a realistic scenario, this approach takes little time to perform since concatenation is a cheap operation. However, each composition creates a patch file that is larger by the amount of a padded file. Thus there is a limit to how many patches we can compose. The limit depends on the size of a padded file.

The second security setting we sketch for sCompose is a two-server scenario. In this case $\mathcal{S} = \{S_{\mathsf{P}}, S_{\mathsf{D}}\}$, where S_{P} and S_{D} are honest-but-curious non-colluding servers. Once again, we suggest two scenarios: an unpadded and a padded variant of the same procedure. The procedure requires the creator of a patch to determine two random permutations. One permutation operates over the index set of the target file. The other permutation works over the index set of the origin file. The creator of the patch uses these permutations to permute the functions that define a patch file. Consequently, the line numbers in the patch file are now no longer meaningful. They are random. The only way to obtain the correct patch file is by using the inverses of the permutation.

The creator of the patch sends the permuted patch description to S_{D}. They send the permutation that operates on the target file and the inverse of the permutation that works over the origin file to S_{P}. These two servers are non-colluding; thus, they do not learn the original patch file. To compose two patch files that share an origin and target file S_{P} creates a new permutation. For these two patch files, two permutations operate over the index set of the shared file. By first applying the permutation where the shared file is the origin file and then the permutation where it is the target file, we link the two patch files. This composition of the patches is a new random permutation.

S_{P} sends this new permutation to S_{D}. S_{D} uses this permutation to create the new function to create the composed patch file. S_{P} needs to store two permutations associated with the new composed patch. These permutations are the permutation of the first patch file that operates over the origin index set and the permutation of the second patch file that works on the target index set.

The unpadded approach leaks the number of operations and the length of the files. However, it does not leak the actual location of the operations. The padded approach does not leak the length of the files. However, the padded version can reveal the number of lines we retain from the first file to the next when we compose two files. This means both do not simulate $\mathcal{F}_{\mathsf{sCompose}}$. They do reveal minimal information and allow for an effective composition.

In Chapter 4 we introduce the concept of Secure Longest Common Subsequence – sLCS – and propose three methods to achieve the functionality. We propose one interactive approach between the server, $\mathcal{S}$, and the receiver, $\mathcal{R}$. The other two approaches are single-sided on $\mathcal{S}$. All approaches rely on the creation of the dynamic programming table that solves the LCS problem for two given inputs. The first uses an interactive equality checking protocol. The latter two use homomorphic encryption and group properties to check for equality.

To hide the length and embedding of the LCS from $\mathcal{S}$, we track every possible path in the dynamic programming table. If we introduce the ability to prune paths, we argue that we also gain the ability to say something about the equality of two elements. Disclosing anything about equality will reveal at least the embedding of the LCS. Storing all possible paths has a space complexity that scales exponentially in the length of the smallest input. This space complexity makes these approaches not workable in a real-world scenario. We argue that our approaches do not simulate the ideal functionality $\mathcal{F}_{\text{sLCS}}$. They either leak the content of one of the input sequences, or they leak all the identical elements between the two sequences.

Finally, in Chapter 5 we introduce the concept of Secure `diff` and Secure `diff3`. These functions rely on a method to align the sequence. Since we did not propose a workable solution for alignment we only introduce the concept of sDiff and sDiff3.

6.1 Future Work

The work in this thesis is mainly theoretical. It considers necessary functions for a secure repository, yet it does not provide workable versions of these functions. To achieve a full secure repository we still require a sCompose function that is effective and fully simulates the ideal functionality $\mathcal{F}_{\text{sCompose}}$. Furthermore, we require sDiff and sDiff3 functions that facilitate the creation of patch files and allows for merging of files. Both these functions require a sLCS function that simulates $\mathcal{F}_{\text{sLCS}}$ and is usable in a real-world setting.

All these functions are subject to further research. It could be interesting to evaluate different approaches or changes in the setting. Different approaches to the LCS method might prove beneficial in limiting its space complexity. We argue that such another method should contain a different approach to the equality of the last elements of two sequence in the LCS procedure. Limiting the alphabet size might

provide insights we could extrapolate in the case where our alphabet is infinite.

Currently, the length of a patch file is equal to the length of the target file. This equal length property follows from our definition of a patch file that uses retentions and insertions. In other words, we describe the target file entirely. However, a patch file is smaller than the target file since the image of the retain function does not consist of an element from Σ^*. The image consists of indices from the origin file. A patch file over plaintext files consists of deletion and insertion. Using deletions and insertions instead of retentions and insertions generally results in shorter patch files than the target file. It is an interesting question to see if we can translate this approach to hidden patch files while maintaining the compose functionality as we describe.

Finally, we did not propose a method to tackle the problem of sDiff and sDiff3. It should be the subject of future work to see if, with different approaches, we can obtain the functionality of these procedures.

Bibliography

[ABC+15] Frederik Armknecht, Colin Boyd, Christopher Carr, Kristian Gjøsteen, Angela Jäschke, Christian A. Reuter, and Martin Strand. A guide to fully homomorphic encryption. Cryptology ePrint Archive, Report 2015/1192, 2015. `https://eprint.iacr.org/2015/1192`.

[ADKF70] Vladimir L'vovich Arlazarov, Yefim A Dinitz, MA Kronrod, and IgorAleksandrovich Faradzhev. On economical construction of the transitive closure of an oriented graph. In *Doklady Akademii Nauk*, volume 194, pages 487–488. Russian Academy of Sciences, 1970.

[Aut15] S. Autexier. Similarity-based diff, three-way diff and merge. *Int. J. Softw. Informatics*, 9:259–277, 2015.

[Bon15] Oscar Bonilla. *The Advantages and Disadvantages of Monolithic, Multiple, and Hybrid Repositories*, 2015.

[BS05] Cyril Banderier and Sylviane Schwer. Why delannoy numbers? *Journal of Statistical Planning and Inference*, 135(1):40–54, 2005. Special issue on lattice path combinatorics and discrete distributions.

[CHK20] Jung Hee Cheon, Kyoohyung Han, and Duhyeong Kim. Faster bootstrapping of fhe over the integers. In Jae Hong Seo, editor, *Information Security and Cryptology – ICISC 2019*, pages 242–259, Cham, 2020. Springer International Publishing.

[Cim20] Catalin Cimpanu. Intel investigating breach after 20GB of internal documents leak online. *ZDNet*, August 2020.

[Cim21] Catalin Cimpanu. Nissan source code leaked online after git repo misconfiguration. *ZDNet*, January 2021.

[CLT14] Jean-Sébastien Coron, Tancrède Lepoint, and Mehdi Tibouchi. Scale-invariant fully homomorphic encryption over the integers. In Hugo Krawczyk, editor, *Public-Key Cryptography – PKC 2014*, pages 311–328, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[CS14] Scott Chacon and Ben Straub. *Pro Git*. Apress, USA, 2nd edition, 2014.

[DBN+01] Morris Dworkin, Elaine Barker, James Nechvatal, James Foti, Lawrence Bassham, E. Roback, and James Dray. Advanced encryption standard (aes), 2001-11-26 2001.

[DGK07] Ivan Damgård, Martin Geisler, and Mikkel Krøigaard. Efficient and secure comparison for on-line auctions. In Josef Pieprzyk, Hossein Ghodosi, and Ed Dawson, editors, *Information Security and Privacy*, pages 416–430, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[Dwo07] Morris Dworkin. Recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac, 2007-11-28 2007.

[FGM09] Matthew Franklin, Mark Gondree, and Payman Mohassel. Communication-efficient private protocols for longest common subsequence. Cryptology ePrint Archive, Report 2009/019, 2009.

[FSF08] Inc Free Software Foundation. *Version Management with CVS*, 2008.

[Gol04] Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, USA, 2004.

[HM75] J. W. Hunt and M. D. McIlroy. An algorithm for differential file comparison. computer science, 1975.

[Kal05] Burt Kaliski. *Euler's Totient Function*, pages 206–206. Springer US, Boston, MA, 2005.

[Kis12] C. Kiselman. Asymptotic properties of the delannoy numbers and similar arrays, 2012.

[KMR11] Seny Kamara, Payman Mohassel, and Mariana Raykova. Outsourcing multi-party computation. Cryptology ePrint Archive, Report 2011/272, 2011. `https://eprint.iacr.org/2011/272`.

[Lev66] Vladimir Iosifovich Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, February 1966. Doklady Akademii Nauk SSSR, V163 No4 845-848 1965.

[Lis05] Moses Liskov. *Fermat's Little Theorem*, pages 221–221. Springer US, Boston, MA, 2005.

[LLS+20] Hyung Tae Lee, San Ling, Jae Hong Seo, Huaxiong Wang, and Taek-Young Youn. Public key encryption with equality test in the standard model. *Information Sciences*, 516:89–108, 2020.

[MP80] William J. Masek and Michael S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18–31, 1980.

[NK15] Koji Nuida and Kaoru Kurosawa. (batch) fully homomorphic encryption over integers for non-binary message spaces. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, pages 537–555, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[NVEL18] Majid Nateghizad, Thijs Veugen, Zekeriya Erkin, and Reginald L. Lagendijk. Secure equality testing protocols in the two-party setting. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, ARES 2018, New York, NY, USA, 2018. Association for Computing Machinery.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *Advances in Cryptology — EUROCRYPT '99*, pages 223–238, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[SK16] T. K. Saha and T. Koshiba. Private equality test using ring-lwe somewhat homomorphic encryption. In *2016 3rd Asia-Pacific World Congress on Computer Science and Engineering (APWC on CSE)*, pages 1–9, 2016.

[Sta21] Nick Statt. Cyberpunk and witcher hackers auction off stolen source code for millions of dollars. *The Verge*, February 2021.

[Sul03] Robert Sulanke. Objects counted by the central delannoy numbers. *Journal of Integer Sequences*, 6, 05 2003.

[War17] Tom Warren. Microsoft confirms some windows 10 source code has leaked. *The Verge*, June 2017.

[ZCL+19] Kai Zhang, Jie Chen, Hyung Tae Lee, Haifeng Qian, and Huaxiong Wang. Efficient public key encryption with equality test in the standard model. *Theoretical Computer Science*, 755:65–80, 2019.