Frameworks for secure collaborative and concurrent editing

Shashank Arora

University at Albany, State University of New York, sarora3@albany.edu

The University at Albany community has made this article openly available. Please share how this access benefits you.

Follow this and additional works at: https://scholarsarchive.library.albany.edu/legacy-etd

Part of the Computer Sciences Commons

Recommended Citation

https://scholarsarchive.library.albany.edu/legacy-etd/2855

This Dissertation is brought to you for free and open access by the The Graduate School at Scholars Archive. It has been accepted for inclusion in Legacy Theses & Dissertations (2009 - 2024) by an authorized administrator of Scholars Archive. Please see Terms of Use. For more information, please contact scholarsarchive@albany.edu.
FRAMEWORKS FOR SECURE COLLABORATIVE AND CONCURRENT EDITING

by

Shashank Arora

A Dissertation
Submitted to the University at Albany, State University of New York
in Partial Fulfillment of
the Requirements for the Degree of
Doctor of Philosophy

College of Engineering and Applied Sciences
Department of Computer Science
August 2022
Dissertation Committee

- Dr. Pradeep K. Atrey (Thesis Advisor/Committee Chair)
  Department of Computer Science
  University at Albany, State University of New York
  Albany, NY, USA

- Dr. Michael Zink (External Committee Member)
  Department of Electrical and Computer Engineering
  University of Massachusetts, Amherst
  Amherst, MA, USA

- Dr. Paliath Narendran (Internal Committee Member)
  Department of Computer Science
  University at Albany, State University of New York
  Albany, NY, USA

- Dr. Amir Masoumzadeh (Internal Committee Member)
  Department of Computer Science
  University at Albany, State University of New York
  Albany, NY, USA
Dedicated to
my parents, Dr. Surender Kumar Arora and Dr. Veena Arora
and
my research advisor, mentor and friend, Dr. Pradeep K. Atrey
for believing in me and sticking with me
through thick and thin
ABSTRACT

Cloud-based online document editing services, such as Google Docs and Office 365, provide an inexpensive and efficient means of managing documents. However, storing data on the cloud also raises certain security and privacy concerns, especially when the data is of confidential and sensitive nature. Storing data on third-party servers can potentially be compromising as it gives an opportunity to the third-party cloud service providers to turn semi-honest and become curious about user data. User data stored on third-party servers is also prone to attacks like virtual machine-based side-channel along with natural language processing and machine learning-based content search and retrieval techniques. In a collaborative online editing environment, user data should never be exposed to the cloud in plaintext form. Thus, there is a need for secure collaborative editing frameworks and services. Prior works around secure collaborative editing have several limitations, and some are not practical.

To that end, this thesis proposes a suite of frameworks, SecureCEdit and two variations of SecureC2Edit to secure user data and enable co-authors of collaborative documents to keep their data confidential while providing collaborative editing services. SecureCEdit is a client-server-based secure collaborative editing framework that provides collaborative access to users. It encrypts user documents on the fly before committing them to cloud service providers. SecureCEdit uses Advanced Encryption Standard (AES) to encrypt document contents and is platform-independent, lightweight with minimal data and resource requirements. SecureC2Edit is a structured peer-to-peer secure collaborative and concurrent editing framework. It makes use of a novel Hybrid Differential Synchronization algorithm to facilitate both collaborative access and concurrent editing. The first variation of SecureC2Edit makes use of the symmetric encryption algorithm, AES to encrypt collaborative document contents. The primary limitation of symmetric encryption, key distribution is overcome by a novel asynchronous key distribution algorithm. The second variation of SecureC2Edit is keyless and uses Shamir’s secret sharing (SSS) as the security mechanism. The use of SSS enables the framework to be synchronized on every edit, minimizing content loss, and scaling better on collaborative documents with a large amount of data. The proposed frameworks and their variations are evaluated on performance and security.
ACKNOWLEDGMENT

This thesis is the result of several years of work during which I have been accompanied and supported by many people. It is now my great pleasure to take this opportunity to thank them. After working as a full-stack developer for three years, I was very keen to pursue full-time doctoral research. I thank the College of Engineering and Applied Sciences, University at Albany for providing me with this opportunity with financial support.

My most earnest acknowledgment goes to my advisor Prof. Pradeep K Atrey who has been instrumental in ensuring my academic, professional, financial, and moral well-being ever since. I could not have imagined having a better advisor for my Ph.D. During the tenure of my Ph.D., I have seen him as an excellent advisor who can bring the best out of his students, an outstanding researcher who can constructively criticize research, and an amazing human being, who is honest, fair, and extremely helpful to others.

I sincerely thank Prof. Michael Zink, Prof. Paliath Narendran, and Prof. Amir Masoumzadeh for serving on my doctoral committee. Their constructive feedback and comments at various stages have played an invaluable role in shaping the thesis up to completion.

My sincere thanks go out to Prof. Manoj Misra, Prof. Priyanka Singh, and Prof. Gaurav Varshney with whom I have collaborated during my Ph.D. research. Their conceptual and technical insight into my work has been invaluable.

There are several people in my everyday circle of colleagues who have enriched my professional and personal life in various ways. I would like to thank Omkar Kulkarni, Shivam Parikh, Vikram Patil, and Gaurav Kodwani whom I also have collaborated with at various stages during my Ph.D. tenure. I am extremely proud of the work we have produced together.

My thanks also go out to my family and friends I have made along the way who have supported me without whom my life would have been a little less colorful. Their unwavering encouragement and camaraderie have been paramount in guiding me all the way.
CONTENTS

ABSTRACT ................................................................. iv

ACKNOWLEDGMENT ..................................................... v

LIST OF FIGURES ....................................................... xi

LIST OF TABLES ......................................................... xiii

LIST OF SYMBOLS ....................................................... xiv

LIST OF ABBREVIATIONS ............................................... xvii

1. Introduction ......................................................... 1
   1.1 Thesis Objectives and Contributions ............................. 5
   1.2 Organization of the Thesis ...................................... 6

2. Background and Related Work .................................... 8
   2.1 Architecture Types ............................................... 8
      2.1.1 Distributed Application Architectures ..................... 9
      2.1.2 Client-Server ............................................... 9
      2.1.3 Peer-to-Peer ............................................... 10
      2.1.4 Hybrid Model ............................................. 10
   2.2 Collaborative Editing Frameworks ............................... 11
      2.2.1 OT ......................................................... 11
      2.2.2 CRDTs ..................................................... 12
      2.2.3 DS ......................................................... 13
   2.3 Cryptographic Algorithms ...................................... 13
      2.3.1 Symmetric Encryption ..................................... 13
      2.3.2 Asymmetric Encryption ................................... 14
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Design Criteria</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.1.1 Real-time Collaboration</td>
<td></td>
<td>35</td>
</tr>
<tr>
<td>4.1.2 Security</td>
<td></td>
<td>35</td>
</tr>
<tr>
<td>4.1.3 Scalability</td>
<td></td>
<td>36</td>
</tr>
<tr>
<td>4.1.4 Consistency</td>
<td></td>
<td>36</td>
</tr>
<tr>
<td>4.2 Assumptions and Threat Model</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.2.1 Honest-but-curious Server</td>
<td></td>
<td>37</td>
</tr>
<tr>
<td>4.2.2 Malicious External Adversary</td>
<td></td>
<td>37</td>
</tr>
<tr>
<td>4.2.3 Honest Peers/Controllers</td>
<td></td>
<td>37</td>
</tr>
<tr>
<td>4.3 HDS-based SecureC2Edit Architecture</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.4 HDS Control Operations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.4.1 Document Pull (docPull)</td>
<td></td>
<td>39</td>
</tr>
<tr>
<td>4.4.2 Document Decrypt (docDecrypt)</td>
<td></td>
<td>39</td>
</tr>
<tr>
<td>4.4.3 Document Encrypt (docEncrypt)</td>
<td></td>
<td>40</td>
</tr>
<tr>
<td>4.4.4 Document Control Commit (docCtrlCmt)</td>
<td></td>
<td>40</td>
</tr>
<tr>
<td>4.4.5 Document Content Commit (docCntCmt)</td>
<td></td>
<td>41</td>
</tr>
<tr>
<td>4.4.6 Document Commit (docCmt)</td>
<td></td>
<td>41</td>
</tr>
<tr>
<td>4.5 HDS Peer and Controller Operations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.5.1 Controller Initialization (ctrlInit)</td>
<td></td>
<td>42</td>
</tr>
<tr>
<td>4.5.2 Peer Initialization (peerInit)</td>
<td></td>
<td>42</td>
</tr>
<tr>
<td>4.5.3 Peer Add (peerAdd)</td>
<td></td>
<td>43</td>
</tr>
<tr>
<td>4.5.4 Peer Exit (peerExit)</td>
<td></td>
<td>43</td>
</tr>
<tr>
<td>4.5.5 Controller Exit (ctrlExit)</td>
<td></td>
<td>44</td>
</tr>
<tr>
<td>4.5.6 Controller Hand Off (ctrlHandOff)</td>
<td></td>
<td>44</td>
</tr>
<tr>
<td>4.6 Asynchronous Key Distribution</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.6.1 Content Embedded Key</td>
<td></td>
<td>45</td>
</tr>
<tr>
<td>4.6.2 hdsEncrypt</td>
<td></td>
<td>46</td>
</tr>
</tbody>
</table>
4.6.3 hdsDecrypt .................................................. 46

4.7 SecureC2Edit Protocol ............................................. 46
  4.7.1 Session Setup ................................................. 48
  4.7.2 Editing Session ............................................... 49
  4.7.3 Session Commit ............................................... 49

4.8 Evaluation ..................................................... 51
  4.8.1 Operability Analysis ........................................... 51
    4.8.1.1 Doc subnet reliability .................................... 51
    4.8.1.2 Content loss ............................................... 52
  4.8.2 Cost Analysis ................................................ 55
    4.8.2.1 Storage cost .............................................. 55
    4.8.2.2 Communication cost ...................................... 56
    4.8.2.3 Computation cost ......................................... 56
  4.8.3 Performance Analysis ......................................... 57
    4.8.3.1 Implementation details and experimental setup ........ 57
    4.8.3.2 Results .................................................... 60
  4.8.4 Security Analysis ............................................. 62

4.9 Chapter Summary ................................................ 65

5. SecureC2Edit Using Secret Sharing ............................... 66
  5.1 Review of the Architecture of SecureC2Edit with AES .......... 67
  5.2 Using SSS with Peer-to-Peer Secure Collaborative Editing .... 68
    5.2.1 SSS with Single Set of Coefficients ....................... 68
    5.2.2 SSS with Multiple Sets of Coefficients .................... 69
  5.3 Entities and Workflow .......................................... 70
    5.3.1 Controller Text ............................................ 70
    5.3.2 Save Shadow ................................................ 70
    5.3.3 Controller Shares .......................................... 71
LIST OF FIGURES

1.1 An illustration of security and privacy issues in current systems. . . . . . . . . 3
1.2 An illustration of advantages of secure systems. . . . . . . . . . . . . . . . . . . 3
2.1 Client-server architecture. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.2 Peer-to-peer architectures. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.3 Operational transformation example. . . . . . . . . . . . . . . . . . . . . . . . 12
2.4 Differential synchronization example. . . . . . . . . . . . . . . . . . . . . . . . 13
3.1 SecureCEdit architecture. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
4.1 SecureC2Edit architecture. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
4.2 The docPull operation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
4.3 The docDecrypt operation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
4.4 The docEncrypt operation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
4.5 The docCtrlCmt operation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
4.6 The docCntCmt operation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
4.7 The docCmt operation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
4.8 The ctrlInit operation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
4.9 The peerInit operation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
4.10 The peerAdd operation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
4.11 The peerExit operation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
4.12 The ctrlExit operation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
4.13 The ctrlHandOff operation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
4.14 A simplified illustration of the SecureC2Edit workflow. .......................... 47

4.15 SecureC2Edit workflow - Create Document - denoted as Box 2a in Figure 4.14]. 48

4.16 SecureC2Edit workflow - Open Document with Controller Initialization - de-
noted as Box 2c in Figure 4.14]. .............................................. 48

4.17 SecureC2Edit workflow - Open Document - denoted as Box 2b in Figure 4.14]. 48

4.18 SecureC2Edit workflow - Controller Watch - denoted as Box 4 in Figure 4.14]. 50

4.19 SecureC2Edit workflow - Save Document during Editing Session - denoted as Box 5a in Figure 4.14]. .............................................. 50

4.20 SecureC2Edit workflow - Save Document after Editing Session - denoted as Box 5b in Figure 4.14]. .............................................. 50

4.21 A complete one-shot illustration of the SecureC2Edit workflow. .................... 51

4.22 Controller synchronization cycle. ....................................................... 53

4.23 Results of edit length experiment. ...................................................... 58

4.24 Results of peer scaling experiment. ..................................................... 60

5.1 Peer-to-peer secure document editing with SSS: different entities. ............... 67

5.2 The docDecrypt operation for secret sharing. ....................................... 74

5.3 The docEncrypt operation for secret sharing. ....................................... 74

5.4 The docCntCmt operation for secret sharing. ....................................... 75

5.5 The docCmt operation for secret sharing. ............................................ 75

5.6 Execution time comparison between AES and SSS with \( \phi = 1 \) and \( \phi = 5 \). 77
## LIST OF TABLES

1.1 Survey of terms of service of popular CSPs. .................................................. 2

2.1 A comparison of the past works related to secure collaborative editing with the work proposed in this thesis. ................................................................. 19

3.1 Literature review w.r.t. specified design criteria for SecureCEdit. ....................... 24

3.2 Running time. ........................................................................................................ 29

4.1 Average time (in ms) to edit a character in the two cases. Case 1: For varying edit length with two peers; and Case 2: For varying number of peers, when a block of 20 characters is added or deleted.) .................................................. 61

5.1 Results of AES experiment. .................................................................................. 76

5.2 Results of SSS experiment with $\phi = 1$. .......................................................... 76

5.3 Results of SSS experiment with $\phi = 5$. .......................................................... 77
# LIST OF SYMBOLS

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_i$</td>
<td>The $i^{th}$ coefficient used in the polynomial equation for Shamir’s secret sharing</td>
</tr>
<tr>
<td>auth_tok</td>
<td>An authentication token received by a peer upon successful authentication</td>
</tr>
<tr>
<td>$b$</td>
<td>The length of document that receives edits from other peers</td>
</tr>
<tr>
<td>$c$</td>
<td>Ciphertext</td>
</tr>
<tr>
<td>$c'$</td>
<td>The updated cipher text of document content after an editing session</td>
</tr>
<tr>
<td>$c'_i$</td>
<td>Encrypted text for $i^{th}$ share</td>
</tr>
<tr>
<td>$c_{ip}$</td>
<td>The IP address of the controller</td>
</tr>
<tr>
<td>$c_{port}$</td>
<td>The active port number of the controller</td>
</tr>
<tr>
<td>const</td>
<td>A constant term comprised of index values</td>
</tr>
<tr>
<td>cred</td>
<td>The user credentials</td>
</tr>
<tr>
<td>$d_{c_i}$</td>
<td>The number of seconds after a synchronization cycle begins when $i^{th}$ peer drops off</td>
</tr>
<tr>
<td>$D$</td>
<td>Size of the minimum edit script that covers one input to other</td>
</tr>
<tr>
<td>$doc_con$</td>
<td>Contents of a collaborative document</td>
</tr>
<tr>
<td>$doc_con_i$</td>
<td>Document contents for $i^{th}$ share</td>
</tr>
<tr>
<td>$doc_len$</td>
<td>Number of characters in the collaborative document</td>
</tr>
<tr>
<td>$doc_req$</td>
<td>Document fetch request</td>
</tr>
<tr>
<td>$D_{PKC}(\cdot)$</td>
<td>The public-key decryption function</td>
</tr>
<tr>
<td>$D_{AES}(\cdot)$</td>
<td>The AES decryption function</td>
</tr>
<tr>
<td>$e$</td>
<td>A single edit made to the collaborative document</td>
</tr>
<tr>
<td>$ECL_i$</td>
<td>Expected content loss for $i^{th}$ peer</td>
</tr>
<tr>
<td>$ECL^r$</td>
<td>Expected content loss for $r$ peers</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>$E_{AES}($</td>
<td>The AES encryption function</td>
</tr>
<tr>
<td>$E_{PKC}($</td>
<td>The public-key encryption function</td>
</tr>
<tr>
<td>$E_{id}$</td>
<td>The email identifier of an active peer</td>
</tr>
<tr>
<td>$E_u$</td>
<td>The email identifier of a co-author $u$</td>
</tr>
<tr>
<td>$f(.)$</td>
<td>The function used to create shares in Shamir’s secret sharing</td>
</tr>
<tr>
<td>$file_id$</td>
<td>File identifier</td>
</tr>
<tr>
<td>$file_id_i$</td>
<td>File identifier of the file with $i^{th}$ share</td>
</tr>
<tr>
<td>$fp$</td>
<td>The FilePicker Object</td>
</tr>
<tr>
<td>$g$</td>
<td>Average number of words in a document</td>
</tr>
<tr>
<td>$k$</td>
<td>The minimum number of shares required to reconstruct the secret in a $(k,n)$ secret sharing scheme</td>
</tr>
<tr>
<td>$K_{Pr}$</td>
<td>The private key</td>
</tr>
<tr>
<td>$K_{Pu}$</td>
<td>The public key of a co-author $u$</td>
</tr>
<tr>
<td>$K_{sym}$</td>
<td>The symmetric encryption key</td>
</tr>
<tr>
<td>$K_{sym_i}$</td>
<td>The symmetric encryption key for the $i^{th}$ character</td>
</tr>
<tr>
<td>$K'_{sym}$</td>
<td>The encrypted version of the symmetric encryption key</td>
</tr>
<tr>
<td>$l$</td>
<td>Edit length</td>
</tr>
<tr>
<td>$l_i$</td>
<td>Length of edit generated by $i^{th}$ peer</td>
</tr>
<tr>
<td>$m$</td>
<td>Plaintext</td>
</tr>
<tr>
<td>$mod$</td>
<td>The modulus operation</td>
</tr>
<tr>
<td>$n$</td>
<td>The number of shares created by a $(k,n)$ secret sharing scheme</td>
</tr>
<tr>
<td>$N$</td>
<td>Sum of lengths of old document and new document</td>
</tr>
<tr>
<td>$p$</td>
<td>Number of peers in a doc subnet</td>
</tr>
<tr>
<td>$P$</td>
<td>Probability value</td>
</tr>
<tr>
<td>$p_i$</td>
<td>$i^{th}$ peer</td>
</tr>
<tr>
<td>$peer_list$</td>
<td>A list of all active peers in a doc subnet</td>
</tr>
<tr>
<td>$pwd$</td>
<td>Password of a co-author</td>
</tr>
<tr>
<td>$q$</td>
<td>A prime number</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>$r$</td>
<td>Number of peers that drop off</td>
</tr>
<tr>
<td>$s_1$</td>
<td>Share 1 of SecureCEdit key generation algorithm</td>
</tr>
<tr>
<td>$s_2$</td>
<td>Share 2 of SecureCEdit key generation algorithm</td>
</tr>
<tr>
<td>$s_3$</td>
<td>Share 3 of SecureCEdit key generation algorithm</td>
</tr>
<tr>
<td>$S$</td>
<td>Set of encrypted keys of all co-authors</td>
</tr>
<tr>
<td>$Sc_i$</td>
<td>Share for $i^{th}$ CSP</td>
</tr>
<tr>
<td>$t_{tok}$</td>
<td>A successful termination token</td>
</tr>
<tr>
<td>$t_{p_1}$</td>
<td>The time when peer $p_1$ is initialized</td>
</tr>
<tr>
<td>$t_{p_2}$</td>
<td>The time when peer $p_2$ is initialized</td>
</tr>
<tr>
<td>$t_{p_3}$</td>
<td>The time when peer $p_3$ is initialized</td>
</tr>
<tr>
<td>$u_{id}$</td>
<td>User identifier of a co-author</td>
</tr>
<tr>
<td>$U$</td>
<td>A list of identifiers of all co-authors</td>
</tr>
<tr>
<td>$v$</td>
<td>Total number of random coefficients</td>
</tr>
<tr>
<td>$w$</td>
<td>An existing word</td>
</tr>
<tr>
<td>$x$</td>
<td>The parameter used in $f(.)$ for Shamir’s secret sharing</td>
</tr>
<tr>
<td>$y$</td>
<td>Average number of characters in a word</td>
</tr>
<tr>
<td>$Z_q$</td>
<td>A finite field with order $q$</td>
</tr>
<tr>
<td>$\bot$</td>
<td>null/no output</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>Negligible Probability value</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>Network reliability</td>
</tr>
<tr>
<td>$\delta$</td>
<td>The average typing speed in characters per second</td>
</tr>
<tr>
<td>$\Gamma_c$</td>
<td>Synchronization cycle of the controller $C$</td>
</tr>
<tr>
<td>$\Gamma_p$</td>
<td>The time required to synchronize an edit</td>
</tr>
<tr>
<td>$\Upsilon$</td>
<td>The user’s machine</td>
</tr>
<tr>
<td>$\phi$</td>
<td>Number of sets of $k - 1$ coefficients</td>
</tr>
<tr>
<td>$\Phi$</td>
<td>The third party web application</td>
</tr>
<tr>
<td>$\Omega$</td>
<td>Intercepting traffic between client machine and server</td>
</tr>
</tbody>
</table>
## List of Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>Advanced Encryption Standard</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ASCII</td>
<td>American Standard Code for Information Interchange</td>
</tr>
<tr>
<td>CRDT</td>
<td>Conflict-free Replicated Data Types</td>
</tr>
<tr>
<td>CSP</td>
<td>Cloud Service Provider</td>
</tr>
<tr>
<td>DEK</td>
<td>Document Encryption Key</td>
</tr>
<tr>
<td>Doc Subnet</td>
<td>A sub-network formed by a controller and peer(s) for a collaborative document.</td>
</tr>
<tr>
<td>DoS</td>
<td>Denial of Service</td>
</tr>
<tr>
<td>DDoS</td>
<td>Distributed Denial of Service</td>
</tr>
<tr>
<td>DS</td>
<td>Differential Synchronization</td>
</tr>
<tr>
<td>ECL</td>
<td>Expected Content Loss</td>
</tr>
<tr>
<td>HDS</td>
<td>Hybrid Differential Synchronization</td>
</tr>
<tr>
<td>HTTPS</td>
<td>Hypertext transfer protocol secure</td>
</tr>
<tr>
<td>ML</td>
<td>Machine Learning</td>
</tr>
<tr>
<td>NLP</td>
<td>Natural Language Processing</td>
</tr>
<tr>
<td>OT</td>
<td>Operational Transformation</td>
</tr>
<tr>
<td>P2P</td>
<td>Peer-to-peer</td>
</tr>
<tr>
<td>RB</td>
<td>Red Black</td>
</tr>
<tr>
<td>REST</td>
<td>Representational State Transfer</td>
</tr>
<tr>
<td>SSS</td>
<td>Shamir’s Secret Sharing</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
CHAPTER 1

Introduction

Cloud paradigm has spread across all the sectors replacing the traditional systems with more efficient, fast, enhanced storage capabilities and minimized cost-based services. People are regularly availing numerous cloud-based services like Google Drive [44], Microsoft One Drive [71], etc for storage and Google Docs [3], Microsoft Office 365 [5], etc for allowing online document editing. With the increase in availability and ease of use of these cloud services, the user base of cloud service providers (CSPs) and online office suites has greatly increased [69]. The collaborative features provided by online office suites like the G Suite by Google, Office 365 by Microsoft, etc provide users with a productive working environment. Though cloud-based systems provide numerous aforementioned facilities but outsourcing one’s data to the remotely distributed cloud servers managed by third-party service providers, often poses serious concerns about the security and privacy of user data.

The majority of the services provided by these cloud-based systems require all user data to be stored on cloud storage which results in user data being in possession of a third party [40], [89], [54]. This gives an opportunity to the honest third party to turn into a semi-honest (also called honest-but-curious) entity, thereby potentially revealing a user’s or an organization’s confidential information [27], [83]. Securing user information is important, especially when the contents are sensitive and private [97], [1], [4]. Confidential documents such as research articles, patents, lawsuits, internal documents of businesses, and new product proposals often have multiple co-authors that work on them collaboratively. However, the contents of these documents need to be kept private until they are ready to be published. It is imperative that these documents are kept protected against not only malicious entities but also semi-honest ones [25].

Users are often unaware of the risks involved or what is actually happening to their data while in transition for availing these services. The storage need for multiple users is facilitated by creating numerous virtual machines (VMs) on the same server as per their request. Ristenpart et al. [89] identified that it is possible to extract information from one
VM from another co-existing on the same physical computer. The contents residing across multiple VMs but residing on the same server are prone to this attack called cross-VM side-channel attack [89]. This attack allows the content stored on one VM to be breached and accessed by another VM residing on the same server. The cross-VM side-channel attack can potentially pose a threat to users’ data present on CSPs.

Moreover, popular CSPs have a clause in their terms of services that allows them to use and modify the content stored on their cloud [10], which is also a topic of concern [103]. Table 1.1 lists popular CSPs and provides excerpts from their terms of service documents highlighting this concern. As summarized in the table, all major CSPs have at least some combination of read, update or delete access to user data. Users, while availing these services ignorantly grant authorization to these CSPs to access and modify the content as per their need [103]. Advanced technologies like various semantic tools, Natural Language Processing
(NLP) [22] [21], Machine Learning (ML) techniques [49] [114] etc further equip the attackers with tools that can prove as a privacy threat to the users content residing at these cloud centers [25].

Figure 1.1 shows the security and privacy issues in current collaborative editing systems. Since the collaborative documents exist in plaintext form on the CSPs, features like keyword-based search and collaborative editing can be provided to the users. However, due to the semi-honest nature of CSPs, user contents become susceptible to the use of the aforementioned NLP and ML techniques. Figure 1.2 shows an example of a secure editing framework, in which documents stored on the cloud are encrypted. The upside of such a framework is that user data becomes secure against unauthorized reads from CSPs but services provided by them no longer become usable. Current collaborative editing applications
can only work in plaintext form. Hence, there is a need for developing such features to work in the encrypted domain.

There are numerous challenges that need to be addressed in secure collaborative editing of documents. The key requirements of a secure collaborative editing framework are:

- **Security**: ensuring the privacy and confidentiality of users’ documents,
- **Collaborative access**: allowing access of the documents to multiple users, and
- **Concurrent editing**: enabling real-time editing for multiple users.

In such a framework, the CSP is assumed to be semi-honest [25]. Semi-honest entities while providing services to users, possess an intent to access user data. Thus it has to be ensured that user data is never exposed to the CSP in plaintext form. Secondly, all communication between the collaborating users in an editing session needs to be secure, to make sure no information is leaked to a third party. Lastly, a good solution needs to ensure that the collaboration allows concurrent access to the document while ensuring its security.

The problem of collaborative editing has been an active area of research ever since cloud productivity suites and CSPs came into mainstream usage [18], [61]. Optimistic replication is a consistency model widely studied in the area of collaborative editing. Most notable are Operational Transformation (OT), Conflict-free Replicated Data Types (CRDTs) and Differential Synchronization (DS) [100] [101] [99]. Amongst these, OT requires a server to propagate all edit messages to all collaborators. Whereas no CRDT-based text editors have been developed as of the writing of this thesis even though numerous research has been done in the area. There is a disconnect between research and industry when it comes to CRDT-based collaborative data editing solutions.

A lot of research has been done in the area of secure collaborative editing [32, 9, 113, 31, 70, 57]; however, existing solutions fail to guarantee the security of user data from either a server that is used to synchronize the data from multiple users or semi-honest CSPs that are used to store the data. This is mainly because most of the existing solutions for secure document editing embrace the client-server architecture, which means that the server acts as the edit synchronization hub. Thus, it gains access to all the edits that the users make and can potentially reconstruct the document by pooling them together. An ideal framework
needs to ensure that the document data is always encrypted on the storage but remains available in plaintext form at the users’ end. To achieve this, there is a need to decouple the editing and edit-synchronization module from the encryption and storage module.

Along with a decoupled concurrent editing framework, an ideal solution needs to ensure that the co-authors of the document have the means to fetch their documents from the CSP and decrypt its contents. Depending on the encryption framework used, it introduces the challenge of key distribution \[105\]. Most researches incorporate symmetric encryption for content security but keep key distribution out of scope \[9\] \[51\]. It might be the case that not all the co-authors are available for a synchronous key sharing protocol. Thus the key distribution protocol needs to be asynchronous enabling them to access the key whenever they become available.

A semi-honest CSP might also choose to act maliciously with access control \[110\]. The CSP may choose to revoke access of certain users to a document or may covertly try to give access to an unauthorized user of a document. An ideal solution should be able to prevent unauthorized users from accessing the contents of a document and should be capable of restoring access to users whose access to a document has been revoked.

Persuaded by the aforementioned discussion, we now establish the goals and objectives of this thesis and outline the contributions, in Section 1.1. Later, in Section 1.2, we describe the organization of the rest of the thesis.

### 1.1 Thesis Objectives and Contributions

While the overarching goal of this thesis is to design, develop and evaluate frameworks for secure collaborative and concurrent editing, the specific objectives are as follows:

- **Objective 1**: To examine the suitability of a client-server framework for secure collaborative and concurrent editing, and identify its limitations.

- **Objective 2**: To evaluate the advantages of a peer-to-peer (P2P) framework for secure collaborative and concurrent editing over client-server-based solutions.

- **Objective 3**: While in **Objective 2**, the security mechanism adopted in the framework can be convectional key-based, in this objective, we explore a key-less scheme for the
Keeping the above objectives in mind, the key contribution of this thesis is a suite of frameworks for secure collaborative and concurrent editing. More specifically, the thesis presents the following three frameworks, aligning with the three objectives listed above:

- **(For Objective 1) SecureCEdit**: A client-server model for secure cloud-based document editing that supports collaborative access but not concurrent editing.

- **(For Objective 2) SecureC2Edit (Key-based)**: A structured P2P model for secure collaborative and concurrent editing. As this framework requires symmetric encryption for encrypting the data, an asynchronous key distribution protocol is proposed for key management and distribution to facilitate the use of encryption methods that do not require a key.

- **(For Objective 3) SecureC2Edit (Key-less)**: This is also a structured P2P model for secure collaborative and concurrent editing which does not require a key, hence eliminating the overhead of key management and distribution.

### 1.2 Organization of the Thesis

The organization of the rest of this thesis is as follows:

- **Chapter 2** discusses the background material related to collaborative editing frameworks and the related works on secure collaborative editing. The aim of this chapter is to review the work that has been done in this area and highlight research gaps.

- **Chapter 3** presents the first of the three frameworks (corresponding to Objective 1), SecureCEdit, which is a client-server model for secure cloud-based document editing. The chapter also analyzes its security and viability.

- **Chapter 4** presents the second framework (corresponding to Objective 2), SecureC2Edit, which is a structured P2P model for secure collaborative and concurrent editing. Since this version of SecureC2Edit requires a key for encrypting the data, this chapter also
proposes an asynchronous key distribution protocol to facilitate security. In this chapter, the evaluation of the proposed SecureC2Edit framework is also presented in terms of operability, cost, performance, and security analyses.

- Chapter 5 describes the third framework (corresponding to Objective 3), i.e., SecureC2Edit with a key-less security mechanism. This chapter also evaluates the framework in terms of performance and security.

- In Chapter 6, a thesis summary is presented, which is followed by the conclusions drawn from the work presented in this thesis. Also, the potential future work is discussed.
CHAPTER 2
Background and Related Work

This chapter introduces the background material pertaining to the thesis, followed by related works in this field of research. The background material discussion starts with various application architecture types and their applicability and viability in building secure collaborative and concurrent editing frameworks (in Section 2.1). We then discuss the most popular collaborative editing frameworks and analyze their feasibility for building secure frameworks for collaborative and concurrent editing (in Section 2.2). Next, cryptographic techniques relevant to our work are discussed in Section 2.3, which is followed by an analysis of literature in the area of cloud security, specifically, security issues pertaining to user data confidentiality and privacy (in Section 2.4). Next, Section 2.5 presents a review of the past works related to secure collaborative editing. Section 2.6 outlines the novelty of the work presented in this thesis compared to the literature. Finally, the chapter summary is provided in Section 2.7.

2.1 Architecture Types

There are integrated architectures vs modularized architectures. In an integrated architecture, the editor, consistency management module, and synchronization module are tightly coupled versus in a modularized architecture, they are not. Naturally, the development of modularized applications is favored over one where all the modules are tightly integrated. In the area of collaborative editing, both architectures have been researched. However, modularized architectures are favored as they make it easier to add collaborative editing features to an existing editor. Additionally, a modularized architecture makes it easier to extend collaborative features to different productivity suite applications like spreadsheets, presentations, etc.
2.1.1 Distributed Application Architectures

Distributed applications are a class of software applications whose modules exist on different machines on a network. All modules are capable of working independently and co-ordinate with each other via message passing [104]. The most widely used distributed application architectures are: client-server, peer-to-peer (P2P), three-tier, n-tier, and hybrid. Out of these, client-server, P2P, and hybrid architectures are most relevant to this work and are explained below:

2.1.2 Client-Server

Figure 2.1 showcases the client-server model. In a client-server model, there exist, multiple clients, that request services from a central server. There can be more than one servers, however, all clients communicate with the server, and the server, in turn, provides services to all clients. The clients themselves do not communicate with each other directly [6]. The advantage of a client-server architecture is that the implementation of a service can be abstracted from the client. The only knowledge a client is required to have is the communication protocol with the server, to construct and send recognized requests. The server responds to the requests by performing any designated computations associated with the requests, constructs an appropriate output, and sends it to the respective client. In order to serve multiple client requests, the server prioritizes them by means of a schedule to serve them effectively. This architecture model is the most susceptible to Denial of Service (DoS)
attacks [79].

2.1.3 Peer-to-Peer

In a P2P architecture, all machines connected to a network are called peers. Peers communicate with each other providing and requesting services from other peers. There is no central server, instead, all peers serve as both client and server. Peer-to-peer networks can be structured or unstructured [68]. In an unstructured P2P network, all peers have the same data and capabilities and possess the ability to directly communicate with all the other peers on the network. Whereas in a structured P2P network, the peers possess different data and capabilities. Peers are connected by a tree-like structure and communicate back and forth with their respective parent nodes. Figure 2.2a showcases an unstructured P2P architecture and Figure 2.2b showcases a structured P2P architecture.

2.1.4 Hybrid Model

The hybrid network model makes use of both the client-server and the P2P architectures. There exists a central server that helps peers discover and connect with each other. After the process of discovery, the peers then behave either as being a part of an unstructured P2P network or a structured P2P network. Hybrid networks have been known to perform better than pure client-server and pure P2P architectures in various scenarios [95].
2.2 Collaborative Editing Frameworks

The problem of collaborative editing has been an active area of research ever since cloud productivity suites and cloud service providers (CSPs) came into mainstream usage [18], [61]. Optimistic replication is a consistency model widely studied in the area of collaborative editing. Most notable optimistic replication techniques are Operational Transformation (OT), Conflict-free Replicated Data Types (CRDTs), and Differential Synchronization (DS). Amongst these, OT requires a server to propagate all edit messages to all collaborators [98], [34], [28]. CRDT based solutions are generally too slow for real time concurrent editing because of the eventual consistency property of CRDT [64], [109], [82], [13]. In DS, each collaborator also has their own copy of the document, and can work independently. DS achieves consistency across the replicas with a continuous cycle of difference (diff) and patch operations [100] [101] [99].

2.2.1 OT

OT is an optimistic replication framework for consistency maintenance and concurrency control in collaborative editing applications. In OT, each collaborator has its own copy of the document and can work concurrently with other collaborators. All changes made by one collaborator are propagated to other collaborators through a central server. OT works in a client-server model and works by encoding each edit into sets of operations which are propagated to all collaborators and then incorporated into their copies. The encoded operations are sent to the central server the corresponding edits are incorporated. If multiple operations are sent to the central server by multiple clients, they are then transformed by the server to ensure that they don’t conflict with each other. The disadvantages of using OT are its reliance on a central server for the transformations and the inability to send multiple edits in one operation.

An example of OT is illustrated in Figure 2.3. The operation O1 = Ins[0,"x"] is generated at Client 1 and O2 = Del[2,"c"] is generated at Client 2. Both operations are sent to the central server. Since O1 is executed first, the content string becomes “xabc”. Now, execution of the operation of O2 as is would result in an inconsistency. Hence O2 is transformed to O2' to accommodate for O1 being processed first.
2.2.2 CRDTs

CRDTs are data structures that are replicated across multiple nodes in a network, and each replica can be updated concurrently and independently. CRDTs eventually become consistent with all the replicas due to their *Eventual Consistency* property. CRDTs typically work in a hybrid model, where a central server helps peers find each other. All further communication for consistency management takes place between the peers. There are two types of CRDTs, operation-based CRDTs, and state-based CRDTs. State-based CRDTs communicate the entire state of the CRDT to all other replicas to resolve inconsistencies. This makes state-based CRDTs easier to design and implement but introduces additional communication overhead. Operation-based CRDTs send just the edit operations to all other replicas resulting in lower communication overhead but requiring that the operations are not dropped, duplicated, and arrive in order when sent across the network. The main drawback of CRDTs is that they guarantee eventual consistency but do not guarantee real-time synchronization, which makes them less desirable for developing a real-time collaborative editing framework.
2.2.3 DS

DS is another optimistic replication technique for consistency maintenance that is highly modularized. In DS, each collaborator has their own copy of the document and can work independently. DS achieves consistency across the replicas with a continuous cycle of difference (diff) and patch operations. DS traditionally works on a client-server model where the server is responsible for propagating changes to all collaborators. A basic architecture of DS is illustrated in Figure 2.4 [41]. When changes are made to the Client Text, a diff operation is executed between the Client Text and Common Shadow to create Edits. After the Edits are compiled, the changes made to the Client Text are propagated to the Common Shadow. The Edits are then Patched to the Server Text.

2.3 Cryptographic Algorithms

2.3.1 Symmetric Encryption

Symmetric encryption is a class of encryption algorithms that make use of one cryptographic/symmetric key to both encrypt and decrypt information. Encrypted data is called ciphertext which can only be decrypted back to plaintext using the cryptographic key. Symmetric encryption algorithms typically consist of complex functions which take plain text data and convert it to ciphertext using the symmetric key. These algorithms are fast and
suited for encrypting large amounts of data. There are two subclasses of symmetric encryption algorithms, block ciphers, and stream ciphers. Block ciphers encrypt data in fixed-size chunks, called blocks whereas stream ciphers encrypt one bit at a time. In the area of secure collaborative editing, block symmetric encryption algorithms, such as the Advanced Encryption Standard (AES) [7], have been the most prevalent. AES operates on blocks of 128 bits and supports key lengths of 128, 192, and 256 bits. The key size dictates the amount of transformation of data that takes place during the execution of AES, with a longer key size resulting in greater transformation. The main drawback of symmetric encryption is that the sender and receiver involved in the communication of data are required to possess the symmetric key. This limitation is overcome using asymmetric encryption, explained below.

2.3.2 Asymmetric Encryption

Asymmetric encryption algorithms make use of a pair of keys, a public key, and a private key. Data encrypted with public keys can only be decrypted using a corresponding private key, whereas data encrypted using a private key can only be decrypted using the corresponding public key. Public keys are known to everyone, whereas private keys are not. Typically, a sender encrypts data using the public of the receiver, who upon receiving the ciphertext, decrypts it using the private key. Asymmetric algorithms are usually slower than symmetric algorithms and thus are used to communicate symmetric keys. In a typical use case, the sender encrypts bulk data using a symmetric encryption algorithm and the symmetric key using an asymmetric encryption algorithm. It then sends both, the encrypted data and key to the receiver. The receiver is then able to decrypt the symmetric key first using their private key, and subsequently the data using the decrypted symmetric key. Some examples of the most widely regarded asymmetric encryption algorithms are Diffie-Hellman key exchange [36],[37], ElGamal encryption [38], Elliptic Curve Cryptography, RSA [90] etc.

2.3.3 Shamir’s Secret Sharing

Shamir’s Secret Sharing (SSS) [8] technique is a secret sharing technique where a secret message is divided into multiple parts, called shares. The shares, by themselves, do not reveal any information about the secret. In order to reconstruct the secret from the shares, some of the secrets need to be combined. This minimum number of shares required to construct
the secret can be set as a threshold value. If the number of shares present is less than the threshold value, it is not possible to reconstruct the secret. Nor it is possible to infer anything about the secret. In this way, SSS provides information-theoretic security.

In the traditional \((k, n)\) SSS scheme, \(k - 1\) coefficients are chosen from a finite field defined over \(\mathbb{Z}_q\), where \(q\) is a prime. The document shares (i.e., encrypted documents) of a secret document are created using the following equation:

\[
f(x) = \sum_{i=0}^{k-1} a_i \times x^i \mod q \tag{2.1}
\]

In order to reconstruct the secret from shares, \(k\) data points (i.e., shares) can be combined using the following Lagrange’s interpolation equation:

\[
f(x) = \sum_{i=1}^{k} \left[ y_i \times \prod_{j=1,i\neq j}^{k} \frac{x - x_j}{x_i - x_j} \right] \mod q \tag{2.2}
\]

To reconstruct the secret, exactly \(k\) shares are required. It is not possible to reconstruct the secret with \(k - 1\) or fewer shares.

SSS is computationally more efficient than symmetric encryption methods but has the limitation of requiring more than one non-colluding CSPs [33], though there are some recent works to address this limitation of SSS scheme [106], [107].

The SSS method has been used for various applications, such as secure 3D medical data visualization [73], [75], secure image and video scaling [74], [59], encrypted domain image and speech enhancement [63], [62], [112], [111], and image tampering and localization detection [96]. It has also been used for secure merging of PDF files [94] and secure searching into PDF files [92].

### 2.4 Cloud Data Security

Migration of existing infrastructure to cloud has resulted in a massive scale-up of storage and services available to users. In [11] Ahmed and Saeed highlight various issues related to cloud data specifically that existing encryption methods fail to scale well and that
real-time monitoring algorithms do not work well with large volumes of data. Balachandran and Prasad discuss the pros and cons of big data mining and analytics on cloud data in [19]. These services, while facilitating the development of applications and features for the user, put confidential data at risk. In [88], the author analyzes security and privacy concerns arising from the redistribution of responsibilities across various cloud supply chains to develop services. They focus on access and usage of user data across multiple parties and the risks associated with it. They also discuss the increased risk of data breaches because of the distributed access to data. In Al-Shomrani et al. [12], the authors establish that as the number of users availing large-scale data storage services increases, the security and privacy of data becomes more challenging. They discuss the importance of the establishment of detailed policies for user data on cloud. Sun et al. [102] discuss challenges associated with maintaining the security and privacy of user data due to the distributed nature of CSPs. They argue the importance of the implementation of security and privacy measures being integrated into the architecture of cloud applications and services. In [86], the authors evaluate the performance overhead associated with cloud security models and assess the detrimental impact it has on users. Kauffman et al. [55] discuss the role of various parties involved in the cloud architecture in establishing the security of user data. They argue that it cannot be the sole responsibility of the CSP to have data security measures, nor that it is sufficient. In order to ensure the security of user data, all parties involved need to keep security in mind. Rao et al. [87] discuss various challenges associated with data security over the cloud. They discuss numerous issues like application security, data leakage prevention, physical and personal security, identity and access management, etc. They argue that all user data should be encrypted before it is uploaded to cloud. In [110], the author discusses various access control challenges and concerns in encrypted data over CSPs. They state that just having encrypted storage is not sufficient. Without adequate access control mechanisms, the cloud may be susceptible to other attacks such as Denial of Service (DoS) and propose a model to prevent such attacks. With all challenges highlighted in the area of cloud data security, it is evident that there is still a lot of work to be done to solve the problem. The main challenges lie in the distributed nature of cloud applications. There is a need for innovation and research for data security on cloud systems as well as applications and services that use data stored on the cloud. This thesis proposes methods and frameworks to secure user data in collaborative editing applications.
2.5 Existing Works on Secure Collaborative Editing

Secure real-time collaborative editing is a widely studied area and presents a plethora of challenges due to the complexity of the system. Multiple approaches have been studied to solve the problem. Initial works dealt with secure collaborative editing frameworks that were developed as browser extensions. D’Angelo et al. [32] presented a scheme that encrypted data when it was transmitted from Google Docs to Google Drive and decrypted it when fetched from Google Drive. However, changes in the communication protocol between Google Drive and Google Docs made the scheme obsolete. A similar solution was presented by Adkinson-Orellana et al. [9] that supported multiple encryption schemes. However, it suffered from the overhead of maintaining two hidden documents on Google Drive to store the information regarding the encryption algorithms and keys used for each document. Huang and Evans [51] presented a scheme that overcame the disadvantages of the scheme presented by D’Angelo et al. [32] and Adkinson-Orellana et al. [9]. They added incremental updates to their browser extension solution by utilizing Google Docs auto-save functionality. However, applying these schemes for the distribution of decryption keys to the users who share the same document proved to be challenging [113]. Later, in [31], D’Angelo et al. provided a theoretical analysis of the popularity of concurrent editing services using Etherpad document data. They established that most documents on Etherpad do not have more than one collaborator but there are users who make use of real-time collaborative document editing services.

Furthermore, Michalas and Bakopoulos [70] developed a solution called SecGOD, as a JavaScript Greasemonkey add-on that encrypted Google Doc documents before saving them to Google Drive. These solutions suffered from two challenges. Firstly, the systems did not work in real-time, and secondly, they were developed as browser extensions which meant that any change in communication protocols resulted in the solutions becoming obsolete. In [57], the document is stored as an authenticated hash chain of encrypted, signed diffs to prevent the CSP from gaining access to the contents of a shared document but greatly increases its size. They also do not discuss the performance of their proposed method in real-time collaboration scenarios. Felsch et al. proposed a secure real-time editing framework in [39]. However, their proposed method requires all changes to be encrypted as they take place, introducing a significant overhead in real-time collaboration. Moreover, their proposed
method has a heavy trade-off between storage and security. In [35] Debreceni et al. proposed a secure collaborative framework for version control systems but it is not real-time. Kollmann et al. [58] have proposed a pure P2P collaborative editing scheme that achieves privacy with not relying on a CSP. The scheme however relies on at least one collaborator being available in order to add a new collaborator. Moreover, the documents in the proposed method are never encrypted. They report an average add time of 11 ms and an average delete time of 64.9 ms. Our proposed method shows better performance, as discussed in Section 4.8.3 and can add new collaborators asynchronously.

In addition to the above works, Jiang et al. [53] proposed a lightweight collaborative editing framework that uses stream cipher to encrypt document edits. This introduces an additional overhead due to encrypting and decrypting each edit made by every collaborator. Moreover, their framework requires a passcode to be communicated to all collaborators through external means, which either has to be done through a trustworthy central server, or through secure channels which is an additional overhead. In [56], Kleppmann et al. proposed a secure and collaborative messaging scheme, which uses CRDTs. Since CRDTs guarantee eventual consistency edit synchronization is too slow for real-time concurrent editing. Moreover, their scheme relies on external infrastructure for key management.

There have been a few privacy concerns pertaining to P2P connections. P2P connections can lead to confidential information on the network to be exposed [76] [20]. There has been research done in the area of secure P2P communication using blockchain-based smart contracts [80]. A blockchain-based P2P network verifies the identity of a peer before sending or receiving a message. After verifying the identities the traffic is then encrypted using a shared secret. This ensures confidentiality, integrity, and reliability [66]. Recent studies have also worked on improving the communication delays over bitcoin-based networks [91].

2.6 Thesis Novelty Against Existing Work

Table 2.1 provides a comparison of the proposed work with the other related works in the area of secure collaborative editing. As can be seen in the table, the three frameworks (listed in the last three rows) presented in this thesis are different from the past works in various aspects. SecureCEdit uses a web-based application that stores the users’ documents over CSP (i.e., Google Drive) in encrypted form, yet provides a majority of online edit-
Table 2.1: A comparison of the past works related to secure collaborative editing with the work proposed in this thesis.

<table>
<thead>
<tr>
<th>Work</th>
<th>Collaborative Access</th>
<th>Concurrent Editing</th>
<th>Asynchronous Key Distribution</th>
<th>Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>D’Angelo et al. [32]</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Client-Server</td>
</tr>
<tr>
<td>Adkinson-Orellana et al. [9]</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Client-Server</td>
</tr>
<tr>
<td>Huang and Evans [51]</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Client-Server</td>
</tr>
<tr>
<td>Yeh et al. [113]</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Client-Server</td>
</tr>
<tr>
<td>Kleppmann et al. [56]</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Client-Server</td>
</tr>
<tr>
<td>Michalas and Bakopoulos [70]</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Client-Server</td>
</tr>
<tr>
<td>Felsch et al. [39]</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Client-Server</td>
</tr>
<tr>
<td>Kolmann et al. [58]</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Peer-to-Peer</td>
</tr>
<tr>
<td>Jiang et al. [53]</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Client-Server</td>
</tr>
<tr>
<td><strong>SecureCEdit</strong> (Presented in Chapter 3)</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Client-Server</td>
</tr>
<tr>
<td><strong>SecureC2Edit - key-based</strong> (Presented in Chapter 4)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Structured Peer-to-Peer</td>
</tr>
<tr>
<td><strong>SecureC2Edit - key-less</strong> (Presented in Chapter 5)</td>
<td>Yes</td>
<td>Yes</td>
<td>No key required</td>
<td>Structured Peer-to-Peer</td>
</tr>
</tbody>
</table>

ing functions. While SecureCEdit includes an asynchronous key distribution mechanism, it does not support concurrent editing, the limitation which is overcome in SecureC2Edit. SecureC2Edit is presented with two encryption options, key-based and key-less, and is different from other works (including SecureCEdit [16]) in the following aspects:

1. The SecureC2Edit framework adopts a structured P2P architecture, while most other solutions are based on client-server architecture.

2. In a structured P2P architectural setting, the SecureC2Edit framework provides both collaborative access and concurrent editing while providing security.

3. The SecureCEdit framework with a key-based security mechanism requires collaborators to propagate document encryption key (DEK) by themselves externally, whereas the SecureC2Edit framework integrates an asynchronous key distribution strategy that
enables sharing of the DEK by embedding it into the document.

4. SecureC2Edit with a key-less option eliminates the need for an encryption key.

2.7 Chapter Summary

This chapter provided a review of the background for the work presented in this thesis. These include different distributed architecture types such as client-server, P2P, and hybrid; collaborative editing frameworks like OT, CRDT, and DS; and cryptographic methods such as symmetric, asymmetric, and secret sharing. It also covers cloud security concepts and the state-of-the-art of secure collaborative editing frameworks. The chapter ends with positioning the work proposed in this thesis against the existing work and establishing its novelty.
CHAPTER 3
SecureCEdit: A Secure Collaborative Editing Framework Using
Client-Server Architecture

This chapter presents the first of the three secure editing frameworks, SecureCEdit\(^1\) (for *Objective 1* of the thesis). The other two are: SecureC2Edit with a key-based encryption mechanism (will be presented in Chapter 4), and with a key-less option (will be presented in Chapter 5). The SecureCEdit is also the first and the preliminary step toward building a more comprehensive secure editing framework SecureC2Edit. The SecureCEdit framework is based on the client-server architecture and encrypts user documents on the fly before saving them to the cloud service provider (CSP). Unlike the traditional browser extension-based solutions for secure document editing, being an independent application, SecureCEdit is platform-independent. Since SecureCEdit has its own communication protocol, it does not rely on the communication protocol between the CSP and their first-party editing applications. This limitation made browser extension-based solutions that relied on reverse-engineering the communication protocol between the CSP and their editing applications obsolete whenever the protocol changed. SecureCEdit ensures that all content being saved on the cloud is encrypted, thereby ensuring the security and privacy of user content. We use the Advanced Encryption Standard (AES) as the encryption algorithm to encrypt document content.

In the rest of this chapter, we begin by outlining the design criteria for SecureCEdit, in Section 3.1. This section also reviews the literature based on these design criteria and highlights the novel aspects of SecureCEdit. Next, in Section 3.2 the SecureCEdit framework is described in detail, with the architectural details in Section 3.2.1 and the algorithmic steps in Section 3.2.2. Further, in Section 3.3, experimental evaluation of the SecureCEdit framework, and in Section 3.4, the security analysis, are presented. Finally, Section 3.5 provides the chapter summary.

\(^1\)The work presented in this chapter was published at the 2nd IEEE Workshop on Security and Privacy in the Cloud in conjunction with IEEE Conference on Communications and Network Security (CNS 2016) [16]
### 3.1 Design Criteria

The following are the design criteria for SecureCEdit:

1. **Platform independence**: Browser extension-based solutions, while having the advantage of being easy to deploy and use, are only specific to one browser. The effectiveness of the solution depends on the availability of the browser itself. This somewhat defeats the purpose of using cloud services for document storage as the user is now limited to using computers having a specific browser. Also, a browser-based extension solution prevents it from being used on smartphones and tablets, which are equally popular as computers nowadays.

2. **Dependence on application specific protocols**: Browser-based solutions have an underlying limitation that their functionality is based on reverse engineering and understanding application-specific protocol (for example, the communication protocol between Google Docs and Google Drive) [30] [67]. If the application-specific protocol is modified or obfuscated, the solutions become obsolete. Hence, there is a need for a solution that is not dependent on application-specific protocols.

3. **Data storage on third party applications**: The past works [32] [51] [9] store information about reverse engineered application-specific protocols. Moreover, the solution presented by Adkinson-Orellana et al. [9] maintains encryption metadata in the solution. Storing data on third-party applications make it a potential vulnerability and a point of attack. Hence, an ideal solution should not store any user data on the third-party application.

4. **Collaborative editing**: It should support collaborative editing of documents. We define collaborative editing as the ability of multiple users to be able to edit a document. In the plaintext domain, this is trivial, however, under the encrypted domain, when the document stored on the cloud is encrypted, this introduces several challenges. The document should be in plaintext whenever a collaborator makes changes to it but encrypted when the changes are committed. Moreover, all collaborators should possess adequate information to decrypt the documents whenever required.

5. **Resources requirement**: It should introduce minimal resource and memory overhead for
it to function as intended. Some of the past works have relied on browser-extension-based solutions which require the use of the specific browser the extension is developed for, which might not always be feasible. Some past works also create additional files with encryption metadata and communication protocol information, which is not ideal. SecureCEdit should not rely on the use of any specific browser, nor should it create any new files on the cloud with additional information.

**Literature review w.r.t. specified design criteria:** Keeping in view of the above, here we discuss the past works that satisfy or deviate from these design criteria. Angelo et al. [32] presented a solution in which a browser extension written for Mozilla Firefox intercepts communication between Google Docs (online document editing tool) and Google Drive, decrypting documents as they are fetched from Google Drive and encrypting them when sent from Google Docs to Google Drive. A key for every document has to be provided by the user. However, their solution did not support collaborative editing of documents, a feature that was taken into consideration while designing SecureCEdit. Moreover, Adkinson-Orellana et al. [9] presented a similar solution as a browser extension. Their solution supported multiple encryption schemes. However, it maintained two hidden documents on Google Drive to store information about encryption algorithms and keys used for every document. Further, Huang [51] also presented a browser extension-based solution, building on solutions presented by [32] and [9] by adding incremental updates to their browser extension solution, by utilizing Google Docs’ auto-save functionality. In addition, [113] proposed an XML-based document format using Red Black (RB) trees to reduce the encryption and decryption time of successive updates to documents. However, their proposed solution induces an overhead to maintain an RB-tree for every document. Table 3.1 provides a summary of the past work w.r.t. design criteria set for SecureCEdit, highlighting its distinctiveness.

### 3.2 The SecureCEdit Framework

In this section, we describe the SecureCEdit framework in detail. The SecureCEdit architecture and algorithmic details are provided in Sections 3.2.1 and 3.2.2 respectively.
Table 3.1: Literature review w.r.t. specified design criteria for SecureCEdit.

<table>
<thead>
<tr>
<th>Work</th>
<th>Collaborative Editing</th>
<th>Resources Required</th>
<th>Platform Dependency</th>
<th>Data Stored</th>
<th>Obsolete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yeh et al. [113]</td>
<td>Yes</td>
<td>Client machine capable of maintaining RB-trees</td>
<td>No</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Adkinson-Orellana et al.</td>
<td>No</td>
<td>Browser add-on</td>
<td>Yes</td>
<td>two tables having encryption metadata</td>
<td>Yes</td>
</tr>
<tr>
<td>Huang and Evans [51]</td>
<td>No</td>
<td>Browser add-on</td>
<td>Yes</td>
<td>application-specific protocol information</td>
<td>Yes</td>
</tr>
<tr>
<td>D’Angelo [32]</td>
<td>No</td>
<td>Browser add-on</td>
<td>Yes</td>
<td>encryption metadata, application-specific protocol information</td>
<td>Yes</td>
</tr>
<tr>
<td>Proposed (SecureCEdit)</td>
<td>Yes</td>
<td>Browser with JavaScript</td>
<td>No</td>
<td>NA</td>
<td>No</td>
</tr>
</tbody>
</table>

3.2.1 SecureCEdit Architecture

Since browser extension-based solutions have limited reliability and are platform-dependent, the obvious choice is to build a web application-based solution. A proof of concept, SecureCEdit, was developed to demonstrate third party text editing tools can be made secure by adding a layer of encryption lying between client and server. Figure 3.1 shows the architecture of SecureCEdit.

The workflow of SecureCEdit can be described as follows:

1. User accesses his/ her CSP using SecureCEdit and opens a document.
2. User requests to generate a key to decrypt the document using the key generation algorithm described in Section 3.2.2.2.
3. SecureCEdit fetches the Developer Key.
4. SecureCEdit fetches the File ID of the document opened by the user.
5. User inputs a passkey.
6. SecureCEdit generates the key.
7. User decrypts the document and starts an editing session.
8. After the editing session, the user can either use the same key generated earlier or generate a new key to encrypt the document.
9. The document is encrypted and sent to the CSP after the user chooses to save the document.

The aim was to make the solution lightweight and easy to use. For security purposes, the solution was designed in such a way that there is no need to store any data associated with the user or the encryption scheme.

### 3.2.2 Algorithmic Details

SecureCEdit uses AES [72] for encryption and decryption. AES is a symmetric-key encryption algorithm, which means that it uses a single key for both encryption and decryption. The key is generated using the key generation algorithm listed in Section 3.2.2.2. There are currently no known attacks to break AES.
APIs: The following APIs were used for implementing SecureCEdit:

1. NicEdit, Version 0.9 r25
2. Google Drive API Version 2
3. Google OAuth 2.0
4. Google Picker API
5. CryptoJS Version 3.1.2

3.2.2.1 Algorithm

The steps for editing a file using SecureCEdit are outlined in Algorithm 1. Below we first describe the functions and keywords used in the algorithm, and then explain the algorithmic steps.

authenticate(user): Users are authenticated using Google OAuth 2.0 API. SecureCEdit validates whether a user is logged in to his Google account using an authentication token supplied by the OAuth 2.0 API. If an authentication token is present, the function returns true. If no authentication token is found, the user is prompted to log in using Google credentials.

open(FilePicker): Opens Google FilePicker to allow the user to select and open a file from Google Drive.

getFileContents(file): SecureCEdit uses Google Drive REST API to fetch the contents of the file selected by the user using FilePicker.

get(developerKey): SecureCEdit fetches the developer key as $s_1$ (described in Section 3.2.2.2).

get(file.fileID): SecureCEdit fetches the FileID attribute of the file opened by the user.

input(passkey): A string to be supplied by the user which is used to generate a key for encryption/decryption of documents, described in detail in Section 3.2.2.2.

generateKey($s_1, s_2, s_3$): The key for encryption/decryption is generated by SecureCEdit using the key generation algorithm described in detail in Section 3.2.2.2.

$D_{AES}(K_{sym}, c)$: Decrypts contents of the file using AES decryption algorithm and the key $K_{sym}$.
\textit{edit(m)}: Captures changes made by the user in the plaintext file \(m\).

\(E_{AES}(K_{sym}, m)\): Encrypts the contents of the plaintext file \(m\) using AES encryption algorithm and the key \(K_{sym}\).

updateFileContents(file, \(c\)): Update the contents of the file to the new cipher text \(c\).

\(save(file)\): Saves the file to Google Drive.

\underline{Algorithm 1}: File editing using SecureCEdit.

\begin{verbatim}
if \texttt{!authenticate(user)} then
    authenticate(user)
fp \leftarrow \texttt{open(FilePicker)} ;
file \leftarrow fp.fileName ;
c \leftarrow \texttt{getFileContents(file)} ;
s_1 \leftarrow \texttt{get(developerKey)} ;
s_2 \leftarrow \texttt{get(file.FileID)} ;
s_3 \leftarrow \texttt{input(passkey)} ;
K_{sym} \leftarrow \texttt{generateKey(s_1, s_2, s_3)} ;
m \leftarrow D_{AES}(K_{sym}, c) ;
m \leftarrow \texttt{edit(m)} ;
c \leftarrow E_{AES}(K_{sym}, m) ;
file \leftarrow \texttt{updateFileContents(file, c)} ;
\texttt{save(file)} ;
\end{verbatim}

The first step is to check whether the user is authenticated with the CSP. In case they are not, user authentication takes place. After an authentication token is retrieved from the CSP, they are allowed to choose the document they want to work on. After the document is chosen, the contents of the document are retrieved. Upon successful retrieval of the document contents, the key generation algorithm is executed followed by decryption. After the user gains access to the decrypted contents, the editing session begins and the user makes the edits. After the editing session, the updated contents of the document are encrypted and saved to the CSP.

3.2.2.2 Key generation

A novel key generation algorithm is proposed, in which three entities provide their shares to form a key for symmetric encryption, the three entities being:

1. The editor (web application used to edit the document)
2. The CSP

3. The user

The following sequence of steps is used to create the key: The algorithm uses three key shares, namely, $s_1$, $s_2$, $s_3$. The $i^{th}$ character of the key, $K_{sym_i}$, is computed by performing addition modulo 256 character by character, as follows:

$$K_{sym_i} = (s_{1i} + s_{2i} + s_{3i}) \mod 256 \quad (3.1)$$

SecureCEdit compares the length of the three shares to find the share with the maximum length. If the share with the maximum length does not have at least 32 characters, padding is added to make the length 32. Thus, the output is a key of length 32 characters. A key of length 32 characters, or 256 bits (1 character = 8 bits) would give $2^{256}$ possible key combinations in case of a brute force attack.

SecureCEdit uses the developer key used to authenticate Google APIs usage from the application, as the first share. For the second share, File ID is used, which is a unique string given to every file stored on Google Drive. The user provides the third share as a string/passkey for every file.

The possibility that multiple users might want to work on the same document is also considered. The proposed key generation algorithm allows users to do that. Assuming, that all potential co-authors choose to use the same web application, which would make share $s_1$ the same for all of them. All co-authors working on the same document will have the same file ID for that document, making share $s_2$ common for all co-authors. Finally, all co-authors require $s_3$ to be distributed among them to decrypt/encrypt the shared document. We acknowledge that this key distribution problem is challenging, but it is out of the scope of this work, but can be solved using a secure key exchange protocol. Thus, it is possible for multiple users to work on the same document without the need to distribute the complete key. Only one share, $s_3$ needs to be distributed among co-authors. This gives an advantage that even if the share gets compromised during distribution, the adversary would not get the complete key.
Table 3.2: Running time.

<table>
<thead>
<tr>
<th>Data Size (in characters)</th>
<th>Google Docs</th>
<th>SecureCEdit</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>3.02s</td>
<td>2.33s</td>
</tr>
<tr>
<td>2000</td>
<td>3.01s</td>
<td>2.33s</td>
</tr>
<tr>
<td>10000</td>
<td>3.16s</td>
<td>2.71s</td>
</tr>
</tbody>
</table>

3.3 Experiments and Analysis

3.3.1 Development Environment

SecureCEdit was developed and tested on a system with a 2.7GHz dual-core Intel Core i5 processor, 8GB LPDDR3 RAM and 256GB PCIe-based flash storage. The browser used for development and testing was Google Chrome Version 50.0.2661.102 (64-bit).

3.3.2 Efficiency

The running time of SecureCEdit was tested and compared with that of Google Docs. Datasets of documents with 500, 2000, and 10000 characters were used for testing. The time it took for Google Docs to save a document with each dataset and the time it took for SecureCEdit to encrypt and save the same document were compared. Table 3.2 shows the average running time of 20 runs.

Table 3.2 shows that SecureCEdit is approximately 19.87% faster than Google Docs when editing and saving documents. It can be inferred that while there is a trade-off between text editing features provided by Google Docs and SecureCEdit, SecureCEdit being a lightweight solution, provides a significant performance boost.

3.3.3 Reliability

Reliability of the solution is defined by the exhaustive testing of the code to make sure it meets all requirements and the stability of the APIs used for developing the solution, SecureCEdit has been thoroughly tested to ensure proper working of the solution and has been developed by using stable versions of the APIs listed in Section 3.2.2, so we can be sure that SecureCEdit is also stable.
3.3.4 Maintenance

Maintenance of a web application is a long-term task to ensure its proper working. It was identified that any future changes that might be required in SecureCEdit would likely be related to the underlying APIs used to develop the solution, SecureCEdit uses APIs listed in Section 3.2.2. If any changes are made to the APIs in a way that changes how the APIs are used would require minimal effort. Thus, maintaining SecureCEdit as a long-term solution would also require minimal effort.

3.4 Security Analysis

3.4.1 Threat Model

Three types of entities are defined for the purpose of security analysis:

*Honest* An entity from which there is no threat to the security and privacy of user data.

*Semi Honest* An entity which is usually not a threat, but should it choose to, can gain access to user data.

*Not Honest* An entity which is always treated as a threat to user data.

The goal is to provide security to documents stored on cloud. Three types of adversaries are considered:

1. External malicious adversary
   The external adversary may refer to an individual or a party intending to gain unauthorized access to a user’s documents. This entity is considered not honest. Various points that such an adversary may choose to attack are taken into account.

2. CSP
   The second type of adversaries are the CSPs themselves, although beneficial because of the services they provide, might choose to access a user’s documents, and thus, are considered semi-honest. The service provider might or might not mean harm, regardless, the goal is to make the contents of users’ documents secure from the service providers.

3. Third party text editing application
Lastly, the threat from the third party application itself, which is also taken as a semi-honest entity, as in this case, SecureCEdit, is also analyzed. Security threats that might occur from such third-party application intended to provide security, are analyzed and the security of the proposed framework is argued.

The client machine, user, and client browser are assumed to be honest entities.

3.4.2 Threat Analysis

In this section, potential threats from various adversaries listed in the threat model are discussed and the security of the presented solution is analyzed.

3.4.2.1 Malicious external adversary

An individual or a party intending to gain unauthorized access to a user’s content might choose to attack different “points”. Four possible locations of attack are considered:

1. The user’s machine (Υ)

Should an attacker choose to attack a user’s machine Υ, the fact that the user is storing his documents on a CSP rather than on his own machine undermines this threat. Since the document itself is not present on Υ, it is not possible for an attacker to gain access to the document from it. However, there is a possibility that the attacker might gain access to keys used for encryption which might have been stored on Υ. Herein lies the security of the proposed key generation algorithm. The keys retrieved from Υ are only one part of the actual key, they would not be useful for the attacker.

2. The third party web application (Φ)

The second location of attack is the third-party web application Φ. As described in Section 3.2, SecureCEdit does not require any data to be stored by Φ. Users are authenticated using Google’s authentication API, OAuth 2, and keys for encryption/decryption are generated on need basis. So Φ itself cannot reveal anything to the attacker. However, the security of the web application also depends on the security of the client machine Υ. The threat model assumes a benign client machine. Malicious applications such as key loggers or any other application that sniff users’ input data in any way compromise data security, but are out of the scope of this work.
3. Intercepting traffic between client machine and server (Ψ)

An attacker might choose to intercept traffic between browser and server Ψ. The proposed solution possesses twofold security from such an attack. Firstly, the security lies in the security of the HTTPS protocol used for communication over the internet. Since Ψ is encrypted under HTTPS protocol, this prevents an attacker from making sense of data being transmitted or tampering it in any way. Secondly, the content of the document itself is encrypted by Φ before it leaves the browser. The solution uses AES encryption scheme which is considered secure. Thus, due to the underlying security of the HTTPS protocol and AES scheme, such an attack would be rendered futile.

4. The CSP (Ω)

Lastly, the possibility that an adversary might choose to attack the CSP Ω itself is considered. The attacker would have to bypass the security provided by Ω, which is a challenging task in itself. Assuming that the attacker is able to do so, documents stored on Ω would be encrypted. The security of the contents of the leaked documents lies in the security of the encryption scheme used, which in this case is AES, and is considered secure. Additionally, the attacker would not have any information about the key used for encryption since, in the proposed algorithm, keys are generated on the fly using shares from multiple entities, so no individual entity has information about the complete key.

3.4.2.2 CSP

Popular CSPs like Google Drive, and Microsoft OneDrive are generally trusted due to their reputation, however, the increasing popularity of CSP services necessitates considering them as adversaries, should they choose to use users’ contents in any way.

SecureCEdit provides functionality to encrypt documents before saving them to CSP, thus making the contents unusable by any adversary not possessing the key, CSPs included. The security of users’ contents relies on the security of the encryption algorithm used, which in the solution is AES, and is considered secure.
3.4.2.3 Third Party Application

Third-party online text editing applications built with the intention of providing security, also need to be considered adversaries. In the case of SecureCEdit, no data is required to be maintained by the application itself as described in Section 3.2.2.

Since the application does not store any data, it cannot be reproduced from the application itself without the input of an authenticated user. Furthermore, SecureCEdit does not communicate with any server other than the CSP itself. Moreover, code auditing can also be used to bolster the trust and security of any such third-party online text editing applications like the one presented in this framework.

3.5 Chapter Summary

In this chapter, we accomplished Objective 1 of the thesis by presenting, SecureCEdit, a secure online editing framework that ensures the confidentiality of user data by encrypting document contents before they are committed to the CSP. The proposed solution is platform-independent and not dependent on application-specific protocols, unlike the traditional browser extension-based solutions. SecureCEdit provides collaborative access in the sense that it enables multiple users to edit the encrypted collaborative document, but not concurrent access. While SecureCEdit presented in this chapter was a preliminary step in the direction of building a secure online collaborative editing solution, in the next chapter we will present a comprehensive SecureC2Edit framework, which provides both, collaborative access and concurrent editing.
 CHAPTER 4


In this chapter, we propose a secure online editing framework, called SecureC2Edit\(^2\), which makes use of a novel Hybrid Differential Synchronization (HDS) algorithm for collaborative and concurrent editing\(^3\). The core idea behind the proposed SecureC2Edit framework is to use a structured peer-to-peer (P2P) architecture \([20]\), \([85]\), as opposed to client-server architecture (adopted in SecureCEdit framework in previous chapter), which allows decoupling of the edit-synchronization module from the encryption/storage module \([65]\). The main responsibility of the edit-synchronization module is to make sure that all edits made by the co-authors are relayed and incorporated, while the responsibility of the encryption/storage module is to periodically encrypt the contents and push them to the cloud service provider (CSP). Further, we propose an asynchronous key sharing algorithm that enables sharing of the document encryption key (DEK) among all the authorized collaborators/users. The proposed key sharing algorithm encrypts the DEK using the users’ public keys and embeds it into the document. In this work, we use Advanced Encryption Standard (AES) as the symmetric encryption mechanism to encrypt the contents of the documents to ensure that there is no information leakage.

To the best of our knowledge, this is the first secure collaborative editing framework that uses a structured P2P architecture with asynchronous key distribution and allows concurrent access and editing to multiple collaborators. The key contributions of this work are as follows:

1. a secure collaborative editing framework that makes use of a hybrid network model, HDS, which ensures that all communication between collaborators does not go through any third party, thus preventing any information leakage, and

\(^2\)Corresponding to Objective 2 of the thesis

\(^3\)The work presented in this chapter is under review at IEEE Transactions on Dependable and Secure Computing \([15]\)
2. an asynchronous key sharing algorithm that makes use of public-key cryptography and the collaborative documents to propagate the DEK to all collaborators.

In the rest of this chapter, we begin by outlining the design criteria for SecureC2Edit, in Section 4.1. Next, in Section 4.2 we discuss the assumptions made while designing SecureC2Edit and establish a threat model based on the assumptions. In Section 4.3 the SecureC2Edit architecture is described in detail, along with the entities involved. Further, in Section 4.4 the control operations based on the novel HDS scheme are discussed in detail, followed by the peer and controller operations in Section 4.5. Next, in Section 4.6 we discuss the need for an asynchronous key distribution protocol along with our proposed framework. We also discuss the \textit{hdsEncrypt} and \textit{hdsDecrypt} algorithms that complement our asynchronous key distribution method. Further, in Section 4.7 we present the SecureC2Edit protocol and the three phases of the SecureC2Edit workflow. We evaluate the SecureC2Edit protocol in Section 4.8 in terms of operability, reliability, content loss. We also present storage, communication, and computation cost analysis as well as a performance and security analysis. Finally, Section 4.9 provides the chapter summary.

4.1 Design Criteria

Following are the design criteria that dictated the design of SecureC2Edit:

4.1.1 Real-time Collaboration

SecureC2Edit needs to be able to support real-time collaboration. All collaborators working on a document should be able to see all changes made by other collaborators in real-time. The problem of real-time collaborative editing in the plaintext domain has been greatly studied and numerous solutions exist in the literature. However, the problem of secure real-time collaborative editing is a challenging one.

4.1.2 Security

SecureC2Edit, while providing a real-time collaborative editing environment, needs to ensure that no user data is exposed to the CSP in plaintext form, thus, preventing any information leakage. Moreover, since SecureC2Edit is a distributed application, it needs to
ensure that it is also secure against network-based attacks such as man-in-the-middle and replay attacks.

4.1.3 Scalability

Adding a layer of security naturally increases the computational complexity of any system, resulting in a decrease in performance. However, this is considered an acceptable trade-off in performance, for the price of security. But, in a real-time collaborative system, the performance of the system on a large scale dictates its success and usability. Thus, SecureC2Edit, while providing the required security, has to be scalable to a large number of users.

4.1.4 Consistency

Consistency is a problem widely studied in the area of collaborative editing. In distributed real-time collaborative editing applications, each collaborator has its own copy of the document. Each collaborator can independently and concurrently edit the document. During this process, the copies of the document diverge from each other across collaborators and eventually converge to be consistent when the collaborator machines communicate with each other. Consistency, while maintaining security, is a vital requirement of SecureC2Edit.

4.2 Assumptions and Threat Model

We make the following assumptions for the SecureC2Edit:

A1 The CSP possesses adequate security mechanisms to defend against external adversaries from reading, tampering, and access control.

A2 The peers possess security mechanisms to defend against unauthorized access from external adversaries.

With the assumptions listed above, the SecureC2Edit framework works under the following threat model:
4.2.1 Honest-but-curious Server

The CSP server is considered honest-but-curious or semi-honest. The server provides storage and access management services but is curious about the data present in the user documents. The server can read the contents of all documents stored on it and has some information about all users connecting to it, like the identity of the registered user, the source IP of each request made by the user, etc. The server also maintains an access control list of each document and has the capability to change it but only on user request. It can also create, update and delete files, also only upon user request. The server can behave maliciously in the sense that it can gain access to and read all user contents stored in a document.

4.2.2 Malicious External Adversary

An external adversary is considered a completely malicious entity capable of dodging SecureC2Edit protocol and authentication schemes to gain access to the CSP. The goal of a malicious external adversary could be to gain unauthorized read or write access to a collaborative document. Examples of an external adversary could be a person working on the CSP side or an independent entity with malicious intent.

4.2.3 Honest Peers/Controllers

Peer are client machines having SecureC2Edit application. Peers are assumed to be honest entities capable of connecting to the CSP server. A peer that is a registered user of the CSP should be able to access its documents. Any peer not registered or logged in needs to do so first in order to gain access to its content on the CSP through SecureC2Edit. The controller of a doc subnet, while having additional privileges, is also considered a peer.

4.3 HDS-based SecureC2Edit Architecture

Keeping in mind, the previously discussed design criteria, in the proposed SecureC2Edit framework, HDS has been designed to work in a hybrid model. We found that the hybrid model works best for secure collaborative editing. Figure 4.1 shows the architecture of SecureC2Edit. Various entities in the SecureC2Edit architecture are described as follows:
Figure 4.1: SecureC2Edit architecture.

- **CSP Server**: The CSP acts as the server. Its main responsibilities are the storage of encrypted documents and assisting peers to connect to the controller of the document they wish to work on.

- **Peer**: All collaborators of a document are called peers. Each peer has its own copy of the document, on which it can work concurrently and independently of other peers. All peers communicate with each other using their Controller.

- **Controller**: The controller is a peer responsible for relaying messages from a peer to all the other peers. The controller propagates all changes made by one peer to all the other peers. The controller is responsible for committing all changes during an editing session to the document on the CSP.

- **Doc Subnet**: In HDS, for each collaborative document, there exists a sub-network. Each sub-network consists of one controller and multiple peers. The controller connects to the CSP, for pulling and pushing the document before and after an editing session, while the peers communicate with each other using the controller. Peers can be a part of multiple Doc Subnets, but a Doc Subnet can have only one controller.

## 4.4 HDS Control Operations

This section describes all the control operations in HDS. These control operations are used by SecureC2Edit in various phases of the workflow.
4.4.1 Document Pull \((docPull)\)

During a \(docPull\) operation, a peer connects to the CSP using the user credential \(cred\), consisting of id \((u_id)\) and password \((pwd)\) of a co-author. The CSP computes the login token \(auth\_tok\) using a function \(auth\) and input \(cred\), and sends it back to the peer. Upon receiving a successful authentication token \((auth\_tok)\), the peer then builds a document fetch request \((doc\_req)\), which is comprised of the file id \((f_id)\) of the document and \(auth\_tok\) and relays it to the CSP. The CSP then executes the request by fetching the contents of the document \((doc\_con)\) using the function \(getDoc\) with input \(auth\_tok\), and sends it back to the peer. This is illustrated in Figure 4.2. \(doc\_con\) contains the cipher text \(c\), the set of encrypted keys for all co-authors \(S\). It may or may not also contain controller IP address \(c\_ip\) and port number \(c\_port\) depending on whether the doc subnet has a controller initialized.

4.4.2 Document Decrypt \((docDecrypt)\)

This operation takes place before the start of an editing session. A peer executes the \(docPull\) operation and decrypts the document using the \(hdsDecrypt\) function, as illustrated in Figure 4.3. The \(hdsDecrypt\) function takes the following inputs: co-author email identifiers \(E_u\), the private key \(K_{Pr}\) and the encrypted document \(doc\_con\), and it outputs the decrypted document in plaintext \(m\). This allows the peer to gain access to the contents of the document in plaintext form.
4.4.3 Document Encrypt \textit{(docEncrypt)}

The \textit{docEncrypt} operation takes place as a part of three control operations, \textit{docCntCmt}, \textit{docCmt} and \textit{peerInit}, described in the next section. The controller logs on to the CSP with its \textit{u\_id} and \textit{pwd} and receives an \textit{auth\_tok}. The controller then encrypts the latest contents of the document \textit{m} using \textit{hdsEncrypt} to get the new cipher text \(c'\) and \(S\). This is illustrated in Figure 4.4.

4.4.4 Document Control Commit \textit{(docCtrlCmt)}

As shown in Figure 4.5, this operation is responsible for updating the contents of a document with the information required to connect to the controller. The controller connects to the CSP and fetches the document it wants to work on, same as in Section 4.4.1. It then appends its IP address (\textit{c\_ip}) and the port (\textit{c\_port}) it’s listening on, to \textit{doc\_con} and sends it back to the CSP, at which point the document is updated using \textit{updateDoc} function. The document text remains untouched during this operation.
4.4.5 Document Content Commit (docCntCmt)

The controller executes docEncrypt to get $c'$. It then recompiles doc_con with $c'$, $S$, $c_ip$ and $c_port$. The updated doc_con is sent back to the CSP. The controller information does not change during this operation. This is illustrated in Figure 4.6.

4.4.6 Document Commit (docCmt)

Figure 4.7 illustrates the doc_cmt operation that takes place at the end of an editing session. The controller performs the docEncrypt operation. The updated document text is encrypted and controller information is removed. The controller then sends the contents doc_con and $f_id$ to the CSP. The document is then saved. This operation marks the end of an editing session.
4.5 HDS Peer and Controller Operations

4.5.1 Controller Initialization ($\text{ctrlInit}$)

In this phase, the controller thread gets initialized using the $\text{startCtrlThread}$ function, followed by $\text{docCtrlCmt}$ (refer to Figure 4.8). This ensures that the controller information becomes available to all other peers to connect to the controller. This marks the start of an editing session. All peers that perform a $\text{docPull}$ subsequently know that a controller for the document has been initialized and an editing session is active.

4.5.2 Peer Initialization ($\text{peerInit}$)

This phase is responsible for setting up peers for an editing session. In this phase, a peer thread/program is initialized using the $\text{initializePeerThread}$ function, followed by $\text{docDecrypt}$, as shown in Figure 4.9. The peer now gains information about the controller and connects to it. If the connection is successful, the controller executes a $\text{peerAdd}$. The absence of controller information after $\text{docDecrypt}$ signifies that the peer is the first one to pull the document from the CSP and a controller has not been initialized. If this is the case, $\text{ctrlInit}$ takes place, followed by $\text{peerInit}$. 
4.5.3 Peer Add (*peerAdd*)

After acquiring *c_ip* and *c_port* from *doc_con*, the peer connects to the controller. The controller maintains a list of all peers connected to it. Upon a successful connection with the peer, the controller generates a connection token, *c_tok* using the *connectPeerThread* function and adds this peer to the *peer_list*. The order of insertion is maintained. After a peer is added to the list, the controller relays it to all the peers. This is illustrated in Figure 4.10.

4.5.4 Peer Exit (*peerExit*)

When a peer wishes to end its editing session, it sends a termination request token (*t_req*) to the controller (refer to Figure 4.11). The controller makes sure that all changes made by the peer are communicated to all the other peers in the doc subnet and proceeds to terminate its connection to the peer using the *terminatePeerThread* function. The controller then generates a successful termination acknowledgment, *t_tok*, containing the updated peer list. The updated peer list is then sent to all the currently connected peers.
4.5.5 Controller Exit (\textit{ctrlExit})

At the end of an editing session, the controller performs a \textit{docCmt}. After the commit, all changes made by all the peers during the editing session are committed to the CSP and the controller information is removed from the document. The removal of controller information from the document signifies that the document is not in an active editing session anymore. This is followed by the termination of the controller thread, using the \textit{stopCtrlThread} function, as shown in Figure 4.12.

4.5.6 Controller Hand Off (\textit{ctrlHandOff})

As can be seen in Figure 4.13, during an editing session, the controller may become unavailable, or be disconnected from the network. In this scenario, the peers are unable to communicate with their controller, and a new controller needs to be initialized. In this scenario, the peers make use of the \textit{peer list}. The peer highest on the list performs a \textit{docCntCmt}. This ensures that the latest contents of the document are committed to the CSP. The peer then executes \textit{ctrlInit}. This makes this highest priority peer, the controller of the doc subnet. The other peers wait a specified amount of time and perform \textit{peerInit}. After all the peers connect to the new controller, the doc subnet is restored and the editing session can proceed. This completes a \textit{ctrlHandOff}.
4.6 Asynchronous Key Distribution

HDS ensures that all changes made during an editing session are propagated between peers and are not exposed on the CSP. However, after an editing session, when the document is to be saved to the CSP, an additional layer of security is required to ensure that the contents of the document are not exposed. This work explores the use of symmetric encryption to provide this layer of security. The idea is to encrypt the contents every time the document is committed to the CSP and decrypt it on pull. The control operations affected by introducing security are \textit{docPull}, \textit{docCmt} and \textit{docCntCmt}. These changes and other intricacies are detailed below.

Symmetric encryption with HDS provides the benefit of fast encryption and decryption times during document commits and document pulls. However, it introduces an additional overhead of key distribution. This challenge is resolved by an asynchronous key distribution mechanism that uses public-key cryptography, described as follows.

4.6.1 Content Embedded Key

Symmetric encryption requires the use of a document encryption key $K_{sym}$, which is used for encryption and decryption. After a document is encrypted, $K_{sym}$ is appended to the header of the document in the form of permission strings with the format $E_u||E_{PKC}(K_{Pu}, K_{sym})$. A permission string is generated for each co-author of the document with its own email id $E_u$ and public key, $K_{Pu}$. Individual permission strings for each author ensure that they are able to independently decrypt the contents of the document.
4.6.2 **hdsEncrypt**

The encryption takes place at the end of `docCntCmt` and `docCnt` control operations. The document $m$ is encrypted using an AES function, $E_{AES}(m, K_{sym})$. This is followed by generation and embedding of permissions for each co-author into the document. $E_{PKC}(K_{Pu}, K_{sym})$ is a public key encryption function which encrypts $K_{sym}$ with $K_{Pu}$, the public key of the co-author, and outputs $K'_{sym}$, the encrypted version of $K_{sym}$.

**Algorithm 2: hdsEncrypt**

<table>
<thead>
<tr>
<th>Input</th>
<th>$m, K_{sym}, peer_list$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>$c, S$</td>
</tr>
<tr>
<td></td>
<td>$c \leftarrow E_{AES}(m, K_{sym})$</td>
</tr>
<tr>
<td></td>
<td>$S \leftarrow \text{null}$</td>
</tr>
<tr>
<td></td>
<td>for each peer in the peer_list do</td>
</tr>
<tr>
<td></td>
<td>$E_{id} \leftarrow \text{get email id of the peer}$</td>
</tr>
<tr>
<td></td>
<td>$K_{Pu} \leftarrow \text{get public key of } E_{id}$</td>
</tr>
<tr>
<td></td>
<td>$K'<em>{sym} \leftarrow E</em>{PKC}(K_{Pu}, K_{sym})$</td>
</tr>
<tr>
<td></td>
<td>$S \leftarrow S</td>
</tr>
<tr>
<td></td>
<td>end</td>
</tr>
<tr>
<td></td>
<td>return $c, S$</td>
</tr>
</tbody>
</table>

4.6.3 **hdsDecrypt**

The decryption takes place during `docDecrypt` control operation. To start off, all permissions are extracted into a map with co-author email identifiers, $E_{id}$, the keys and the $K'_{sym}$ as the values. The encrypted contents are also separated during this step. The permissions are iterated over and when the permission for the current co-author is found, the corresponding cipher value is decrypted with $D_{PKC}(K'_{sym}, K_{Pr})$ using the private key, $K_{Pr}$. When the co-author gains access to the symmetric key $K_{sym}$, the contents of the document are decrypted using an AES decryption function, $D_{AES}(c, K_{sym})$ and returned as $m$. If the permission is not found, `hdsDecrypt` returns ⊥.

4.7 **SecureC2Edit Protocol**

The SecureC2Edit protocol makes use of HDS control operations and HDS peer and controller operations to create an unintrusive experience for the user. Both HDS control
Algorithm 3: hdsDecrypt

Input: $E_{id}, K_{Pr}, doc_{con}$  
Output: $m/\bot$

$S \leftarrow$ extract key sets from $doc_{con}$  
$c \leftarrow$ extract ciphertext of the document from $doc_{con}$

for each key set in the $S$ do

if $E_{id}$ matches the current peer’s email id then

$K_{sym} \leftarrow D_{PKC}(K'_{sym}, K_{Pr})$

$m \leftarrow D_{AES}(c, K_{sym})$

return $m$

else

continue;

end

end

return $\bot$

---

Figure 4.14: A simplified illustration of the SecureC2Edit workflow.
operations and peer and controller operations work in the background without the requirement of any user intervention. The protocol can be divided into three phases: session setup, editing session, and session commit. A simplified illustration of the SecureC2Edit workflow is shown in Figure 4.14 followed by the details of various blocks in Figure 4.15 to Figure 4.20.

4.7.1 Session Setup

In this phase, a user can choose to either create a document or open one. In case a document needs to be created (Figure 4.15), the user is presented with a blank document, and at the same time, a corresponding black document is created on the CSP. After successful creation, the peerInit operation along with ctrlInit takes place, followed by peerAdd. At this point, SecureC2Edit enters the editing session phase. In case a user wants to open a document, the protocol fetches the document and checks whether the document is in an active editing session. The presence of controller information indicates an active editing session, whereas its absence indicates the lack of one. If the document is not in an active editing session, it means that the current peer also needs to be initialized as the controller.
(Figure 4.16), which takes place using the operation \textit{ctrlInit} in conjunction with \textit{peerInit}. This is followed by \textit{peerAdd} and \textit{docDecrypt} operations. Following \textit{docDecrypt}, the user gains access to the plaintext contents of the document and enters an active editing session. Now, if a document is opened and controller information is found indicating that the document is in an active editing session, \textit{peerInit}, \textit{peerAdd} and \textit{docDecrypt} operations are executed (Figure 4.17) and the user joins the active editing session.

### 4.7.2 Editing Session

During an active editing session, all peers connected to the doc subnet periodically synchronize their edits with the controller enabling users to see each other’s edits in real-time. SecureC2Edit watches for controller drop off (Figure 4.18). If the controller is reachable, the editing session proceeds as normal. In case the controller drops off during an active editing session, another active peer is given controller privileges and the editing session resumes. This is called Controller Hand-off. The process of controller hand-off is as follows: when a controller becomes unreachable, the peer pulls the document from the CSP and checks if it contains controller information. If controller information is available but the controller is not reachable using that information, it means that there was an unintentional disconnect from the controller’s side. The peer then initializes itself as the controller and updates the new controller information on the CSP. Subsequent peers also pull the document from the CSP but now they find the new controller active. They then connect to the new controller and the editing session proceeds as normal. During the editing session, the controller may choose to save the document (Figure 4.19), the latest version of the contents are saved to the CSP using \textit{docCntCmt} operation. Note that this does not disrupt the editing session.

### 4.7.3 Session Commit

At the end of an editing session, the non-controller peers may choose to exit the editing session (Figure 4.20). They save the document acknowledging that there will be no further edits coming from them. Any pending unincorporated edits from that peer are then processed by the controller indicating that the peer is safe to leave the network. The peer then sends \textit{t_req} to the controller and SecureC2Edit then executes the \textit{peerExit} operation. The controller removes the peer from \textit{peer_list} and sends a successful termination token, \textit{t_tok}. The peer
then exits the doc subnet. Similarly, all the other peers disconnect from the doc subnet. The last peer, who also possesses controller privileges then chooses to save the document by committing all the changes made to it to the CSP. It also removes controller information from the document indicating that it is no longer in an active editing session by executing `peerExit` and `ctrlExit` operations. This marks the end of the SecureC2Edit protocol.

A complete one-shot illustration of the SecureC2Edit workflow can be seen in Figure 4.21.
4.8 Evaluation

In this section, we present the following analyses: operability analysis in terms of doc subnet reliability and content loss (section 4.8.1); cost analysis in terms of storage, communication and computation costs (section 4.8.2); performance analysis for varying edit lengths and peer scaling (section 4.8.3); and security analysis against various attacks (section 4.8.4).

4.8.1 Operability Analysis

4.8.1.1 Doc subnet reliability

The reliability of a doc subnet in the proposed SecureC2Edit framework is analyzed using the following lemma.

**Lemma 4.1** In SecureC2Edit framework, the probability of failure of a doc subnet of n peers is \(1 - (1 - \gamma)^n\), where \(\gamma\) is network reliability.

**Proof:** Let \(p\) be the number of peers, \(\gamma\) be the network reliability assuming the network
reliability is the same for all peers. Now, we define the chance of a peer disconnecting from a sub-network to be \(1 - \gamma\). Consequently, the chances of all peers disconnecting becomes \((1 - \gamma)^p\). Now, the doc subnet is up as long as there is at least one peer connected to it. Therefore, the probability of it not failing is \(1 - (1 - \gamma)^p\).

**Example:** Say for a doc subnet with 5 peers has a reliability of 70\%. So, \(p = 5\), \(\gamma = 0.7\). Probability of one peer disconnecting is \(1 - \gamma = 0.3\). Probability of all peers disconnecting is \(0.3^5 = 0.00243\). Consequently, the probability of the doc subnet remaining up is \(1 - 0.00243 = 0.99757\).

### 4.8.1.2 Content loss

Content loss in the proposed framework may occur when one or more peers get disconnected from the doc subnet. In order to this potential analyze content loss, two scenarios need to be examined: (1) there is no controller drop off (2) in case of controller drop-off. We analyze the content loss in these two scenarios, as follows.

**Scenario 1 - No controller drop off:** In this scenario, we assume that the controller always stays connected to the doc subnet and only the peers drop off from the network. To analyze the expected content loss in this scenario, we provide the following lemma (Lemma 4.2).

**Lemma 4.2** In SecureC2Edit framework, the expected content loss when there is no controller drop off is \((1 - \gamma)^r \times \delta \times \sum_{i=1}^{r} d_{c_i}\), where \(i\) is the number of peers in a doc subnet.

**Proof:** Let \(\delta\) be the average typing speed in characters per second and \(\Gamma_p\) be the time required to synchronize an edit. If a peer remains connected, to the doc subnet, there is no content loss. Now, in case a peer does disconnect, let \(d_{c_i}\) be the number of seconds after a synchronization cycle begins when peer \(i\) drops off. Leveraging on the results of Lemma 4.1, the expected content loss for peer \(i\), denoted by \(ECL_i\) will be:

\[ ECL_i = (1 - \gamma) \times \delta \times d_{c_i} \]

We assume in this case that the controller never drops off, consequently, the corresponding peer does not drop off either. Thus, the maximum number of peers that can drop off is \(n - 1\). Let \(r\) be the number of peers that drop off, the expected content loss due to the drop off of
r peers, denoted by $ECL^r$ thus becomes:

$$ECL^r = (1 - \gamma)^r \times \delta \times \sum_{r=0}^{n-1} d_{ci} \quad (4.1)$$

where $0 < r < n - 1$ is the number of peers disconnected.

**Scenario 2 - Controller drop off exits:** In the case where the controller is disconnected, the SecureC2Edit framework hands off the controller role to one of the peers in the editing session. The following lemma (Lemma 4.3) is provided to quantify the expected content loss in this scenario.

**Lemma 4.3** In SecureC2Edit framework, the maximum content loss when there is controller drop-off is $p \delta \Gamma_p$, where $p$ is the number of peers in a doc subnet.

**Proof:** Let $\Gamma_c$ be the synchronization cycle of the controller $C$. This value however is dependent on the peers connected to this controller. Figure 4.22 shows an example timeline of two peers connected to a controller. Let $t_{p_1}$ be the time when peer $p_1$ is initialized. $p_1$ being the first peer, would also result in controller initialization. From the controller’s perspective, let this time be $t = 0$. Let $t_{p_2}$ and $t_{p_3}$ be the time when peers $p_2$ and $p_3$ are initialized with $t_{p_3} > t_{p_2} > t_{p_1}$. There can be three similar cases (1) The controller drops off just before receiving edits from a peer. (2) The controller drops off after receiving edits from a peer but before sending them out. (3) The controller drops off after sending out edits received from peers. The expected content loss in these three cases will be as follows:

- **Case 1:** In this case, the peer is not able to send its edits to the controller since the controller becomes unreachable as shown in Figure 4.22 by the first X mark at time
The controller drops off just before \( p_2 \) sends its edits to \( C \). Since \( p_2 \) is unable to connect to \( C \), this scenario triggers controller hand off and a new controller is initialized. Consequently, no data is lost.

- **Case 2**: In this case, all data sent by the peer is lost shown by the second X mark in Figure 4.22. Since a peer sends \( \delta \Gamma_p \) characters every synchronization cycle, the data loss becomes \( \delta \Gamma_p \) characters per-peer whose edits are not processed and broadcasted by the \( C \).

- **Case 3**: In this case, the controller drops off after processing all edits sent by peers, marked by the third X in Figure 4.22. Since the edits are processed and sent out by the controller, no data is lost. The next peer that tries to send its edits finds the controller unreachable and triggers controller hand-off, same as case 1.

As shown above, while in case 1 and case 3 there is no data loss, in case 2 the expected content loss is \( \delta \Gamma_p \) characters per peer. Hence, for \( n \) peers in an editing session, the maximum expected content loss is \( n \delta \Gamma_p \) characters.

These lemmas form the basis for the following theorem (Theorem 4.4) that provides the bounds on the content loss.

**Theorem 4.4** In the SecureC2Edit framework, the minimum content loss is 0 and the maximum content loss is \( p \delta \Gamma_p \) characters.

**Proof**: As stated in Lemma 4.2, the expected content loss due to the drop-off of \( r \) peers when there is no controller drop off is given by Eq. 4.1. The content loss will be minimum when there is no drop-off of peers as well as the controller. Therefore, substituting \( r = 0 \) in Eq. 4.1 we get,

\[
ECL^0 = (1 - \gamma)^0 \times \delta \times \sum_0^0 d_{c_i} = 0
\]

as the term \( \sum_0^0 d_{c_i} \) reduces to zero.

We now compare the maximum content loss in both scenarios, when the controller does not drop off and when it does. In Scenario 1, the maximum content loss occurs when \( p = 1 \).
peers disconnect from the network. Thus, substituting $r = p - 1$ in Eq. 4.1 (Lemma 4.2), the expected content loss is given by:

$$ECL^{p-1} = (1 - \gamma)^{(p-1)} \times \delta \times \sum_{i=0}^{p-1} d_{c_i}$$

For maximum content loss, we assume that the peers disconnect just before sending their edits to the controller. Thus, $d_{c_i} = \Gamma_p$.

$$ECL^{p-1} = (1 - \gamma)^{(p-1)} \times \delta \times (p - 1) \times \Gamma_p$$  \hspace{1cm} (4.2)$$

In Scenario 2, as stated in Lemma 4.3, the maximum expected content loss is $p\delta\Gamma_p$. In this case, since the controller always drops off, the network reliability $\gamma$ is assumed to be 0. Therefore the maximum expected content loss can be rewritten as:

$$p\delta\Gamma_p = (1 - \gamma) \times p \times \delta \times \Gamma_p$$

$$= (1 - \gamma)^{(p-1)} \times p \times \delta \times \Gamma_p$$

$$= (1 - \gamma)^{(p-1)} \times (p - 1) \times \delta \times \Gamma_p +$$

$$+ (1 - \gamma)^{(p-1)} \times \delta \times \Gamma_p$$

We can see that the above formulation has an extra $(1 - \gamma)^{(p-1)} \times \delta \times \Gamma_p$ term compared to Eq. 4.2 resulting in more content loss when the controller drops off, which is $p\delta\Gamma_p$ characters.

4.8.2 Cost Analysis

4.8.2.1 Storage cost

The cost of storing an encrypted document is computed in terms of the number of characters in $doc_{con}$. Let the number of characters be $doc_{len}$ and each character occupies 1 byte. An encrypted document undergoes some inflation in the number of characters due to the nature of AES, defined as the expansion factor $\|$. An AES-encrypted document exhibits an expansion factor of 1.004. Thus, the storage cost of an encrypted document can be computed to be proportional to $1.004 \times doc_{len}$. 

55
4.8.2.2 Communication cost

The communication cost is computed in terms of length and number of messages required to propagate an edit to all peers. Let peer \( p_1 \) makes an edit, which is propagated through the system in two phases: from peer \( p_1 \) to controller \( C \), and \( C \) to the other peers. The cost of propagating an edit for each phase separately.

\( p_1 \) to \( C \). Let the edit length of \( p_1 \) be \( l_1 \). The length of the corresponding message then becomes \( l_1 + \text{const} \). The \( \text{const} \) term being comprised of index values corresponding to the edit, length of the edit, etc. The length of metadata corresponding to an edit is not dependent on the actual \( l_1 \) and hence can be considered constant, and can be dropped. Thus, the cost of the message is computed as \( l_1 + \text{const} \).

\( C \) to peers. After an edit is incorporated in \( C \), let the new edit length be \( l_2 \). The length of the edit might change as now, the contents of \( C \) are compared to the rest of the peers. The cost of the corresponding message can be computed as \( l_2 + \text{const} \). Assuming that there are \( p \) peers in the doc subnet, the number of messages needed to be propagated is \( p - 1 \). Thus, the total communication cost in this phase becomes \( (l_2 + \text{const}) \times (p - 1) \).

Total cost. The total communication cost can thus be computed by adding the cost of messages from \( p_1 \) to \( C \) and \( C \) to peers. The total cost thus becomes \( (l_1 + \text{const}) + (l_2 + \text{const}) \times (p - 1) \). Asymptotically, after dropping the constant term, the cost becomes \( l_1 + l_2 \times (p - 1) \). Let \( l = \max(l_1, l_2) \). Thus the cost can be written as \( l \times (p - 1) \). Thus, the total cost of communicating an edit comes out to be \( O(l \times p) \).

4.8.2.3 Computation cost

For evaluation of the proposed framework, the computation cost is defined as the total cost of operations required to be performed to incorporate an edit. Each edit requires a diff and multiple patch operations to be performed to be incorporated. We now discuss both operations in detail.

diff. The diff operation converts the most recent edit made to the document into a com-
municable form. The most popular \textit{diff} algorithm is the Myer’s diff \cite{77} algorithm and is used in the implementation of the proposed framework. The computation cost of the original Myer’s diff algorithm is $O(N \times D)$, where $N$ is the sum of lengths of both inputs and $D$ is the size of the minimum edit script that converts one input to another. Linear space refinements of Myer’s algorithm exhibit a time complexity of $O(N \log N + D^2)$ and space complexity of $O(N)$. For our purposes, we can assume $D << N$ since typically, the documents are of significantly larger length than the length of one edit operation. Thus, the time complexity of a diff operation can be assumed to be $O(N \log N)$.

The \textit{patch} operation uses the output of a \textit{diff} to incorporate the edit into the current document. To compute the asymptotic time complexity of a patch operation, it can be assumed that the patch operation can be reduced to a string insert operation. The time complexity of a string insert operation is $O(b + l)$, where $b$ is the size of the second document and $l$ is the edit length. Since asymptotically, $l << b$, the cost of one string operation becomes $O(b)$. In order to incorporate an edit in the doc subnet, $p - 1$ patch operations take place. Asymptotically, the cost of the patch operations comes out to be $O(p \times b)$.

\textit{Total cost.} The total cost of incorporating an edit can be computed as the total cost of the diff and patch operations, i.e., $O((N \log N) + (p \times b))$. Since asymptotically, $b \approx N$, the cost can be written as $O((N \log N) + (p \times N))$.

\section*{4.8.3 Performance Analysis}

\subsection*{4.8.3.1 Implementation details and experimental setup}

SecureC2Edit was implemented entirely using Java. The client text editors were developed using Java Swing. Java Socket Programming was used to develop the doc subnet. The Control and Peer and Controller Operations were also developed using Java. Java Threads were used to implement threads in SecureC2Edit to optimize performance. The Google Drive API was used to establish connection with the CSP, pull and push collaborative documents, and add co-authors to a collaborative document.

All experiments were performed on an Early 2015 Macbook Pro with Intel Core i5 2.7 GHz Dual-Core processor, 8 GB DDR3 memory, and Intel Iris 6100 Graphics Card. The
peers were Java Swing desktop applications and the controllers were implemented as Java Threads.

To simulate tests, both initialization of peers and generation of edits were performed programmatically. All edit operations generated were additions (ADD) and deletions (DELETE) of varying lengths. The idea behind generating operations of such varying lengths was that ADD and DELETE operations of shorter lengths simulate keystrokes and operations of larger lengths simulate operations like cut, paste, select and delete. The experiments performed were classified into two categories (1) Edit length and (2) Peer scaling. Edit length operations were performed between two peers, with different combinations of ADDs and Deletes. Peer scaling operations were performed keeping the edit operations constant but varying the number of peers.

*Edit length experiments.* For this set of experiments, the number of peers was set to 2, with one peer also acting as the controller of the document sub-network. The follow-
ing experiments were conducted: (1) Adds Only (Figure 4.23a), (2) Deletes Only (Figure 4.23b), (3) Equal Adds And Deletes (Figure 4.23c) (4) More Adds Less Deletes (Figure 4.23d) (5) More Deletes Less Adds (Figure 4.23e). For Adds Only, we generated 20 instances of ADD operation with edit lengths varying as 1, 10, 20, 30, 40, ..., 500. Further, for Deletes Only, 20 instances of DELETE operations were generated with lengths varying as 1, 10, 20, 30, 40, ..., 500. Equal Adds And Deletes experiment involved the generation of 10 instances of ADD, followed by 10 instances of DELETE for lengths varying as 1, 10, 20, 30, 40, ..., 500. For More Adds Less Deletes, we generated 13 instances of ADD and 7 instances of DELETE for the same set of lengths. Finally, for More Deletes Less Adds, 7 instances of ADD were generated followed by 13 instances of DELETE. For consistency across all these experiments, initial size of the document was set to 126510.

Peer scaling experiments. For peer scaling experiments, a similar approach was used but keeping the operation type and length the same and varying the number of peers involved. The following experiments were conducted: (1) Adds Only (Figure 4.24a), (2) Deletes Only (Figure 4.24b), (3) Equal Adds And Deletes (Figure 4.24c) (4) More Adds Less Deletes (Figure 4.24d) (5) More Deletes Less Adds (Figure 4.24e). For Adds Only, we generated 20 instances of ADD of length 10 increasing the number of peers from 2 to 10. For Deletes Only, 20 instances of DELETE of length 10 were generated for peers ranging from 2 to 10. Further, for Equal Adds And Deletes, 10 instances of ADD 10 and 10 instances of DELETE 10 were generated. For More Adds Less Deletes, 13 instances of ADD 10 and 7 instances of DELETE 10 were generated, and for More Deletes Less Adds, 7 instances of ADD 10 and 13 instances of DELETE 10 were generated.

For every experiment, we calculate the average time to add and delete. The execution time is calculated by adding the time it takes for one peer to process an edit, the time it takes for the controller to process that edit and send it to all peers, and time it takes for other peers to incorporate the edit. For Edit length experiments, the averages are calculated for each edit length. For peer scaling experiments, the averages were calculated for each different number of peers.
For edit length experiments, the execution time of Adds Only (Figure 4.23a) increases greatly with the edit length. Consequent additions to the contents of a document greatly increase the length of the document. Any addition to the contents requires the content string to be split into two parts followed by the insertion of the new content and then the merger of all three sub-strings. On the other hand, in Deletes Only (Figure 4.23b), the length of the document keeps on decreasing as the experiment continues resulting in the underlying operations of string split and merge taking less and less time resulting in the reduction of the execution times. In the third experiment involving Equal Adds And Deletes (Figure 4.23c), the overall length of the document remains the same throughout the experiment, hence the overall execution time also remains mostly constant, which is reflected in the
Table 4.1: Average time (in ms) to edit a character in the two cases. Case 1: For varying edit length with two peers; and Case 2: For varying number of peers, when a block of 20 characters is added or deleted.)

<table>
<thead>
<tr>
<th>Experiment Type</th>
<th>Case 1 (Edit length)</th>
<th>Case 2 (Peer scaling)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adds Only</td>
<td>2.03</td>
<td>13.98</td>
</tr>
<tr>
<td>Deletes Only</td>
<td>32.7</td>
<td>8.31</td>
</tr>
<tr>
<td>Equal Adds And Deletes</td>
<td>1.60</td>
<td>7.93</td>
</tr>
<tr>
<td>More Adds Less Deletes</td>
<td>1.64</td>
<td>8.30</td>
</tr>
<tr>
<td>More Deletes Less Adds</td>
<td>1.34</td>
<td>8.08</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>7.86</strong></td>
<td><strong>9.32</strong></td>
</tr>
</tbody>
</table>

In our fourth experiment, More Adds Less Deletes (Figure 4.23d), the overall length of the document increases but not as much as Adds Only. Hence we do see an increase in execution time, but not to the same magnitude as seen in Adds Only experiment. The fifth experiment, More Deletes Less Adds (Figure 4.23d) follows the same pattern as delete only, where the execution time of the proposed method becomes lower and lower, but not to the same magnitude as Deletes Only.

In all the peer scaling experiments (Figure 4.24), we see that the execution time is affected by the number of peers. We do observe that the execution time of all the experiments scales linearly with the number of peers or co-authors working concurrently on the same shared document. Note that the scaling is linear with the number of active co-authors, and not the total number of co-authors of the document.

Table 4.1 shows the average time to edit a character across all our experiments for edit length and peer scaling. For edit length experiments, the average is calculated for varying edit lengths scaled down to average time per character (graphically illustrated in Figure 4.23f). For Adds Only experiment, the average time is observed to be 2.03 ms, which is a little higher than the other experiments involving deletes due to deletes reducing the overall length of the document, except for Deletes Only. The average time of Deletes Only is significantly higher than the other experiments due to the initial length of the document being set to 506040 characters instead of 126510 as for the other experiments. We need to set the length of the document greater for Deletes Only in order to have enough characters to delete from the document. The average time to edit a character for Equal Adds and Deletes, and More Adds Less Deletes is approximately the same with Equal Adds and Deletes being a little lower at 1.60 ms than More Deletes Less Adds at 1.64 ms. More Deletes Less Adds
average time turns out to be 1.34 ms due to deletes reducing the length of the document over the course of the experiment.

The average time to edit a character in peer scaling is calculated per-peer scaled down to one peer, as shown in Figure 4.24. We observe similar patterns in edit scaling experiments with Adds Only a little higher at 13.98 ms. For Deletes Only Peer scaling experiment, the length of the document does not need to be greater than for other experiments due to us deleting only 20 characters at a time. As a result, the average time comes out to be 8.31 ms, which is in a similar ballpark as the other experiments, with the other averages being 7.93 ms for Equal Adds And Deletes, 8.30 ms for More Adds Less Deletes and 8.08 ms for More Deletes Less Adds.

From our experiments, we can conclude that the average time to edit a character is observed to be 7.86 ms for varying edit lengths under the most common edit operations and an average of 9.32 ms for peers varying between 2 to 10.

The observed execution time falls in line with the cost analysis. The computation cost was determined to be $O((N \log N) + (p \times N))$. For edit length experiments, the number of peers was kept constant to $p = 2$. Thus for edit length experiments, the computation cost can be estimated to be $O((N \log N)$, or linearithmic time, which is reflected in Figure 4.23. The execution time increases linearithmically in Figures 4.23a and 4.23d and decreases in Figures 4.23b and 4.23c. The execution time stays approximately the same for Equals Adds Deletes since the overall length of the documents remains roughly the same, as shown in Figure 4.23c.

4.8.4 Security Analysis

The contents of a collaborative document exist in plaintext form on the controller and peers. We assume in our threat model that peers and controllers are honest entities. Thus the content existing in plaintext form on these entities is not a point of concern.

The operations that take place in SecureC2Edit on document contents are diff, patch, encryption and decryption. These operations take place on the controller and peers. The diff and patch operations facilitate propagation of edits made on one peer to the others. The diff operations generate patches which are propagated to all peers in the doc subnet. The
network connections across peers in a doc subnet are assumed to be secure. Thus, no content is revealed during propagation.

Thus, we can establish that the edit synchronization module of SecureC2Edit is secure. Furthermore, to ascertain the security of the encryption mechanisms employed, we present a theoretical security analysis of the proposed SecureC2Edit framework using the following theorem and propositions.

**Theorem 4.5** The system is considered \( \epsilon \)-secure if the adversary can know the document contents with negligible probability, \( \epsilon \).

**Proof**: If an adversary gains unauthorized access to a document and tries to decrypt it, \( \text{hdsDecrypt} \) returns \( \bot \) since permissions for external adversary do not exist. Permissions for a co-author are added during \( \text{hdsEncrypt} \) which is executed when a current co-author shares the document with a new user. The adversary may try to add a permission, however, that involves guessing \( K_{sym} \). For AES-256, the length of \( K_{sym} \) is 256 bits. Thus the probability of the adversary guessing the correct key, \( P(k = k^*) = 1/2^{256} \approx \epsilon \). Alternatively, the adversary may try to guess the contents of a document directly. Assuming there are \( \text{doc\_len} \) characters in a document, the probability of the adversary guessing the correct contents \( P(m = m^*) = 1/\text{doc\_len}^{256} \approx \epsilon \), assuming the extended ASCII character set is used. Either case results in the adversary guessing the correct contents of a document with negligible probability, and thus, the system is considered \( \epsilon \)-secure.

**Proposition 4.6** SecureC2Edit is secure against unauthorized reads from semi-honest entities.

**Proof**: According to the SecureC2Edit threat model, the CSP is a semi-honest entity. The CSP has read access to the documents but not write and delete access. The SecureC2Edit framework encrypts the contents of the document using \( \text{docEncrypt} \) before saving it to the CSP. When a document is pulled from the CSP for an editing session, it is decrypted on the peer using \( \text{docDecrypt} \) and encrypted again at the end, before saving the changes to the CSP \([26]\). This ensures that the contents of the documents are always encrypted on the CSP side \([43]\). Thus, according to Theorem 5.1, despite having read access to the documents, the CSP can never make sense of the document contents.
Proposition 4.7 SecureC2Edit is secure against malicious access control from semi-honest entities.

Proof: According to the SecureC2Edit threat model, the CSP is curious about user content. The CSP is also not honest in maintaining access control to collaborative documents. The CSP may revoke access of a collaborator to a document or may share the document with another user covertly. If access from a collaborator is revoked, the document can be re-shared with the collaborator resulting in regular operation. However, if the CSP chooses to grant access to an unauthorized user, since the document is not shared using the proposed method, due to the $\epsilon$-security property of the system, the new user would not be able to access the contents of the document.

Proposition 4.8 SecureC2Edit is secure against malicious external adversaries.

Proof: An external adversary may try to attack SecureC2Edit at various points of the workflow, the CSP, the peers, and the network. Firstly, to get access to the document on the CSP, it must subvert the security of the CSP. With the assumption established in Section 4.2, it is assumed that CSP has adequate security mechanisms in place from external adversaries (assumption A1). The security of the CSP is considered out of the scope of this work. However, even if an external adversary does get read access to the document, the contents exist in encrypted form and thus cannot infer anything [26]. Secondly, the external adversary may also try to attack the users’ machines. According to assumption A2, the peers possess adequate provisions to defend against both.

Proposition 4.9 SecureC2Edit is secure against unauthorized access from network attacks.

Proof: A malicious external adversary may try to attack the doc subnet to gain access to the communication between the peers and controller. The adversary may try to perform network scanning and mapping [50] to try to observe traffic from peers in a doc subnet. The adversary may be able to identify who the controller is and who and how many peers are in the particular doc subnet. However, network mapping does not reveal the content of the document and thus the adversary would not be able to make sense of the actual content of the document. All the communication between the peers and controller and between
controller and CSP is secure [24]. The adversary may also execute a Denial of Service (DoS) [60] or a Distributed Denial of Service (DDoS) [81]. From assumptions stated in Section 4.2 (assumptions A1 and A2), it can be assumed that the peers and CSP have the state-of-the-art means to detect and thwart attempts of DoS and DDoS and using SecureC2Edit does not make the system any more vulnerable to DoS and DDoS. The adversary may also try to attempt to carry out a packet forging attack in order to attempt to adversely affect the integrity of the document. However, secure P2P connections employ checks against malformed packets and prevent them from going through [24].

4.9 Chapter Summary

This chapter presented the secure online editing framework that ensures the confidentiality of users’ documents by performing user-controlled encryption before storing them to the CSP. Also, the proposed asynchronous key distribution mechanism allows multiple users to secure access and edit the documents without communicating with each other. Adoption of an HDS-based structured P2P architecture enables collaborative access and concurrent editing of the documents to multiple users. The extensive evaluation in terms of operability, cost, performance, and security analyses demonstrated the effectiveness of the proposed framework.

While in this chapter, we fulfilled Objective 2 of the thesis, the next chapter will explore the use of an alternate encryption scheme that does not require any key, i.e., for Objective 3 of the thesis.
CHAPTER 5
SecureC2Edit Using Secret Sharing

Advanced Encryption Standard (AES), while providing robust security, has a significant disadvantage of requiring the entire document to be encrypted, each time it has to be saved to the cloud service provider (CSP). This does not become a bottleneck in traditional applications requiring encryption, since, usually, a resource is encrypted once and then decrypted on need basis. However, in a collaborative document editing scenario, the document needs to be encrypted before every commit to the CSP. This can lead to high encryption time compared to editing time especially when the content of the document becomes significantly large. Further, there is an overhead of asynchronous key management with AES. Lastly, since the AES is a block cipher, the keyword search into an encrypted document remains limited to encrypted blocks, which limits its usage in practice.

The approaches presented in Chapter 3 (SecureCEdit [16]) and Chapter 4 (in SecureC2Edit [15]), which involved encrypting documents before saving them to CSPs. The encryption algorithm used in these works was AES. Using AES for the purposes of secure document editing has certain limitations, as discussed earlier. Thus, to overcome these limitations, in this chapter, we propose the use of secret sharing as an alternative encryption approach to SecureC2Edit. Secret sharing facilitates incremental encryption better than AES as it allows the creation of shares of individual edits. The use of secret sharing also mitigates the need for employing key distribution, while providing a significant performance increase. Moreover, this lays a foundation for facilitating the secure domain search feature in secure collaborative frameworks to make them more practical [78].

In [8], a secret sharing scheme, widely known as Shamir’s Secret Sharing (SSS) was introduced. A lot of research has been done on the application of secret sharing in various applications [47], [115], but to the best of our knowledge, this is the first work that explores the use of secret sharing for secure collaborative editing. The work presented in this chapter is aligned with Objective 3 of this thesis.

4The work presented in this chapter was published at the IEEE International Workshop on Information Forensics and Security (WIFS) 2021 [14].
In the rest of this chapter, we begin by reviewing the architecture of SecureC2Edit with AES as the choice of encryption algorithm, in Section 5.1. Next, in Section 5.2 we discuss the feasibility and challenges associated with using SSS with secure collaborative editing in a peer-to-peer (P2P) model. We also compare and contrast two different approaches to selecting coefficients for SSS. In Section 5.3 we discuss the entities involved with using SSS with SecureC2Edit and describe the workflow. Further, in Section 5.4 we present the `diffToSharePatch` algorithm, responsible for processing edits, computing patches and preparing the updated shares. Next, in Section 5.5 we discuss the changes that have to made in some HDS control operations and peer/controller operations previously presented in sections 4.4 and 4.5 to accommodate the use of SSS. In Section 5.6 we evaluate the proposed method in terms of performance, and operability and present a security analysis. Finally, Section 5.7 provides the chapter summary.

5.1 Review of the Architecture of SecureC2Edit with AES

Figure 4.1 (in Chapter 4) showcases a generic architecture of a P2P secure collaborative editing framework. Each collaborative document forms its own doc subnet. Each doc subnet consists of peers, which function as the collaborators of the document with one peer having controller privileges. All peers are connected to the controller. Each peer possesses its own copy of the document, in which they make their edits. The controller is responsible for the synchronization of edits between all peers of a doc subnet and communicating with the CSP to fetch and commit collaborative documents. If the algorithm of choice for employing security is AES, the controller communicates with one CSP. The controller encrypts the
contents of a collaborative document before committing it to the CSP and decrypts them after fetching the document. The decrypted contents are then propagated to all the peers.

5.2 Using SSS with Peer-to-Peer Secure Collaborative Editing

The proposed architecture for P2P secure collaborative editing framework with SSS is shown in Figure 5.1. In the traditional $(k, n)$ SSS scheme, $k - 1$ coefficients are chosen from a finite field defined over $Z_q$, where $q$ is a prime. The document-shares (i.e., encrypted documents) of a secret document are created using the following equation (expanded form of Equation 2.1 introduced in Chapter 2):

$$f(x) = a_0 + a_1 \times x + a_2 \times x^2 + \cdots + a_{k-1} \times x^{k-1} \mod q$$  \hspace{1cm} (5.1)

When $k$ or more document-shares are brought together, the secret document can be reconstructed using Lagrange’s interpolation. The shares of a word (called word-shares) can be created by sharing each character in the word. Here, we assume that the characters are ASCII characters having values less than 128. Thus, in the share creation process, the value of $q$ in the above equation can be taken as 127.

A significant advantage of the SSS scheme is that it possesses information-theoretic security, meaning that an adversary cannot get any information about the secret document file until all the $k$ shares are known. Also, it offers fault-tolerance in the sense that even if a few (i.e., up to $n - k$) servers are non-functional or unavailable, the documents can still be recovered [23]. However, when applied on the text, the SSS scheme can be prone to frequency analysis attack [17], [42], [52]. To counter this attack, we propose necessary changes in the deployment of the SSS scheme. For integration of SSS scheme into the framework, the selection of polynomial coefficients $a_i$, $1 \leq i \leq n$, in Equation (5.1) is one of the most important aspects. We discuss it as follows.

5.2.1 SSS with Single Set of Coefficients

The coefficients for share generation and reconstruction can be kept constant. If the coefficients are kept constant, this enables the system to not require their storage and just one reconstruction for it to function appropriately. Coefficients can be reconstructed on the
controller from the controller shares after retrieving them from the CSPs. The reconstructed coefficients can then be used to generate shares of new edits made to the document contents. The advantage of using constant coefficients is, as previously mentioned, just one reconstruction, and overall faster execution of the proposed algorithm. However, the disadvantage of using one set of coefficients is that the contents become prone to frequency analysis. If static coefficients are used to create shares, any letter appearing multiple times in the document contents will result in the same share letters. In order to prevent frequency analysis, shares can be created using multiple sets of coefficients, as described below.

5.2.2 SSS with Multiple Sets of Coefficients

Using multiple coefficients for the purpose of text secret sharing has been explored in [92]. Shares of each word can be created by randomly choosing \( k - 1 \) coefficients. Since shares of different words are created using different coefficients, different occurrences of the same letter lead to different shares, thus preventing frequency analysis. Moreover, a keyword search functionality over the encrypted shares can be incorporated with the use of SSS, as explored in [92]. For share creation of a word, \( k - 1 \) coefficients can be randomly chosen from a set of \( v = \phi \times (k - 1) \) coefficients, \( \phi \) being the number of sets \( k - 1 \) coefficients. When a keyword search has to be performed, shares of the keyword can be then created using all possible combinations of the coefficients, resulting in \( \binom{v}{k-1} \) shares. The search operation can then be performed by matching words in the shares with the keyword shares. In [92], it was determined that using \( \phi = 5 \) results in an acceptable Index of Coincidence, which is the probability that two randomly chosen letters from a cipher are the same, and thus the value of \( \phi \) is chosen to be 5 for this work.

The usage of randomly chosen multiple sets of coefficients in a collaborative and concurrent editing scenario is challenging. If an edit, \( e \), made to document contents is a part of an existing word \( w \), its shares have to be created using the same set of coefficients that were used to create shares of \( w \), originally. This challenge can however be overcome when patches for \( Sc_1, \ldots, Sc_n \) are being created. While computing patches (will be discussed in Section 5.4), it can be determined whether an \( e \) is a part of an existing word. Under the assumption that words are separated by blank spaces, if \( e \) is surrounded by blank spaces (both leading and trailing), it is not a part of an existing word. If however, \( e \) does not have
a leading or trailing blank space, either the start or the end of $e$ is a part of an existing
word. If an edit is not a part of an existing word, then, $k - 1$ coefficients can be randomly
chosen from the set to create shares for it. If however, an edit is a part of an existing word,
its location in the document contents can be extracted from the \textit{diff} using the index values,
and subsequently, the word shares from $S_{c_1}, \ldots, S_{c_n}$. This is possible because the word in
question and its shares exist at the same index values, and thus can be extracted from the
controller shares. Once shares are extracted, the coefficients that were used to create the
shares can be reconstructed and subsequently used to create shares of the edit. If an edit
consists of multiple words, then it is also possible to determine whether the edit starts from
an existing word, or ends in one, and accordingly, coefficients can either be reconstructed or
randomly chosen for each word.

5.3 Entities and Workflow

As shown in Figure 5.1 in order to use SSS for the purpose of secure document editing,
the controller instead has to communicate with multiple CSPs, each possessing a share of
the collaborative document. Using SSS with P2P secure collaborative editing also requires
the use of the following entities:

5.3.1 Controller Text

The controller text holds the latest contents of the document that is synchronized with
the other peers. Each edit made by any of the peers is incorporated into the controller text
and then propagated through to the other peers.

5.3.2 Save Shadow

The save shadow holds the latest saved version of the document. The purpose of
the save shadow is to determine the latest set of changes made to the document since the
last save. Having a save shadow provides the flexibility of having a varied save frequency.
Whenever a document has to be saved to the CSP, a \textit{diff} is computed between the controller
text and the save shadow.
5.3.3 Controller Shares

The controller maintains $n$ shares, $Sc_1, ... Sc_n$, corresponding to each CSP. The purpose of the controller shares is to incorporate the shares of the edits made to the document. The controller shares remain synchronized with the save shadow.

5.3.4 CSPs

SSS requires the use of $n \geq 2$ CSPs with each CSP responsible for storing a share of the document. CSP$_1, \ldots, $CSP$_n$ store controller shares $Sc_1, ... Sc_n$ respectively. Each time a save request is made, the contents of $Sc_1$ are saved to CSP$_1$, $Sc_2$ to CSP$_2$ and so on.

Workflow: In order to save an edit made to the collaborative document, the following steps are executed:

- compute diff between the controller text and the save shadow.
- using the diff, compute patches for each of the controller shares, $Sc_1, ... Sc_n$.
- patch the edits to their corresponding controller share.
- push the content of $Sc_1, ... Sc_n$ to CSP$_1, \ldots, $CSP$_n$ respectively.

5.4 Computing Patches for Controller Shares

Propagating document edits from the save shadow to the controller shares requires computing a diff between the controller text and the save shadow, and then computing patches for each of the controller shares. Algorithm 4 describes the process of computing these patches. The algorithm takes as input, $n$, the number of shares to be created and diff, the diff between the controller text and save shadow and outputs $n$ patches, for each of the controller shares. The makePatch() method creates a patch, diff_patch from the diff. Patches are used to incorporate recent edits to the shares. We then create $n$ copies of the patch, one corresponding to each controller share. Then, diff_patch is tokenized into string and index values using the tokenize() method. The implementation of the tokenize() method depends on the underlying diff-patch library used. For this work, we use the diff-match-patch [2] library. Typically, all patches contain the index values at which modifications
to the content have taken place and the strings that need to be modified. The purpose of the \textit{tokenizer()} method is to isolate string tokens from the patch for further processing. Following the tokenization, we then create $n$ shares for each string token and then replace the corresponding string with its $i^{th}$ share in the $i^{th}$ patch. This finalizes the creation of patches for the controller shares. The controller shares are then patched with the latest edits and become ready to be committed to their respective CSPs.

\begin{algorithm}
\hspace*{0.02in} \textbf{Algorithm 4: } \texttt{diffToSharePatch} \\
\hspace*{0.02in} \textbf{Input} : $n, \texttt{diff}$ \\
\hspace*{0.02in} \textbf{Output:} \texttt{patch}[1...n] \\
\hspace*{0.02in} \hspace*{0.12in} \texttt{diff\_patch} $\leftarrow$ \texttt{makePatch}(\texttt{diff}) \\
\hspace*{0.02in} \hspace*{0.12in} \textbf{for} $i \colon 1...n$ \textbf{do} \\
\hspace*{0.02in} \hspace*{0.14in} \texttt{patch}[i] $\leftarrow$ \texttt{deepCopy}(\texttt{diff\_patch}) \\
\hspace*{0.02in} \hspace*{0.02in} \textbf{end} \\
\hspace*{0.02in} \hspace*{0.12in} \texttt{patch\_tokens}[] $\leftarrow$ \texttt{tokenize}(\texttt{diff\_patch}) \\
\hspace*{0.02in} \hspace*{0.12in} \textbf{for} $t : \texttt{patch\_tokens}$ \textbf{do} \\
\hspace*{0.02in} \hspace*{0.14in} \textbf{if} \texttt{isString}(t) \textbf{then} \\
\hspace*{0.02in} \hspace*{0.16in} \texttt{token\_shares}[] $\leftarrow$ \texttt{createShares}(n, t) \\
\hspace*{0.02in} \hspace*{0.16in} \textbf{for} $i : 1...n$ \textbf{do} \\
\hspace*{0.02in} \hspace*{0.18in} \texttt{patch}[i].replace(t, \texttt{token\_shares}[i]); \\
\hspace*{0.02in} \hspace*{0.14in} \textbf{end} \\
\hspace*{0.02in} \hspace*{0.14in} \textbf{end} \\
\hspace*{0.02in} \hspace*{0.02in} \textbf{end} \\
\end{algorithm}

5.5 Changes in Operations

In this section, we describe changes that have to be made to some HDS control operations and peer/controller operations. Some of the operations described in the previous chapter have to be modified to accommodate the use of SSS instead of AES. $\texttt{doc\_con}$ does not contain $S$ anymore since we do not need the encrypted key set due to the keyless nature of SSS. While employing the use of SSS, we also do not make use of $\texttt{hdsEncrypt}$ and $\texttt{hdsDecrypt}$ algorithms as is since they are developed for AES. Specifically, there are changes in the following operations:

- HDS Control Operations
  - $\texttt{docPull}$
• Peer and Controller Operations
  
  – peerAdd

In the following subsections, we describe the changes made to each operation.

5.5.1  **docPull**

The *docPull* operation is responsible for retrieving a collaborative document from the CSP and delivering it to the peer. The operation described in the previous chapter does not change when using SSS. At the end of the operation, the peer receives *doc_con* from the CSP. When using AES, *doc_con* consists of *c*, *S*, and optionally, *c_ip* and *c_port*. While employing SSS, we do not require *S* to be a part of *doc_con*. Other than the previously mentioned change, *docPull* operates the same as before as highlighted in Figure 4.2. Instead of *S*, a list of identifiers of all collaborators, *U* is maintained.

5.5.2  **docDecrypt**

The *docDecrypt* operation is responsible for decrypting the collaborative document retrieved using the *doc_pull* operation. It then makes use of *hdsDecrypt* to decrypt contents of the retrieved document. However, when using SSS, we use the *lagPol()* method to reconstruct the document. The *lagPol()* method makes use of lagrange’s interpolation to reconstruct the secret document from its shares using Equation 2.2. The operation is shown in detail in Figure 5.2.

5.5.3  **docEncrypt**

The *docEncrypt* operation for secret sharing requires the controller to retrieve an *auth_tok* from all CSPs. The controller then executes the *diffToSharePath()* method to
prepare patches for all the CSPs. Each patch is then appended with $U$ to get $c'_1, \ldots, c'_n$. The patches are then ready to be committed to their respective CSPs. The complete *docEncrypt* is showcased in Figure 5.3.

### 5.5.4 *docCntCmt*

The controller executes *docEncrypt* to get $c'_1, \ldots, c'_n$. It then recompiles *doc_con$_1$, ..., doc_con$_n$* with $c'_1, \ldots, c'_n$, $U$, *c_ip* and *c_port*. The updated *doc_con$_1$, ..., doc_con$_n$* are sent back to the respective CSPs. The controller information does not change during this operation. This is illustrated in Figure 5.4.
5.5.5 \textit{docCntCmt}

Figure 5.5 illustrates the \textit{docCntCmt} operation that takes place at the end of an editing session. The controller performs the \textit{docEncrypt} operation for each CSP. The updated document text is encrypted and controller information is removed. The controller then sends the contents $doc\_con_1, ..., doc\_con_n$ and $f\_id_1, ..., f\_id_n$ to the CSP. The document is then saved. This operation marks the end of an editing session.

5.5.6 \textit{peerAdd}

The \textit{peerAdd} operation same as highlighted in Figure 4.10. Same as in \textit{docPull}, instead of $S$, a list of identifiers of all collaborators, $U$ is maintained. Otherwise, \textit{peerAdd} behaves the same.
Table 5.1: Results of AES experiment.

<table>
<thead>
<tr>
<th>Doc Length</th>
<th>Expansion Factor</th>
<th>Execution Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1.01</td>
<td>0.0438852</td>
</tr>
<tr>
<td>100</td>
<td>1.007</td>
<td>0.0518032</td>
</tr>
<tr>
<td>500</td>
<td>1.004</td>
<td>0.0542582</td>
</tr>
<tr>
<td>1000</td>
<td>1.003</td>
<td>0.0593121</td>
</tr>
<tr>
<td>5000</td>
<td>1.001</td>
<td>0.0773224</td>
</tr>
<tr>
<td>10000</td>
<td>1.0007</td>
<td>0.1126484</td>
</tr>
<tr>
<td>20000</td>
<td>1.0004</td>
<td>0.1510292</td>
</tr>
</tbody>
</table>

Table 5.2: Results of SSS experiment with $\phi = 1$.

<table>
<thead>
<tr>
<th>Doc Length</th>
<th>Expansion Factor</th>
<th>Exec Time (ms, $\phi = 1$)</th>
<th>% Decrease ($\phi = 1$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>0.0237788</td>
<td>45.8159</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>0.0246764</td>
<td>52.3651</td>
</tr>
<tr>
<td>500</td>
<td>1</td>
<td>0.0234438</td>
<td>56.7922</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>0.0260782</td>
<td>56.0322</td>
</tr>
<tr>
<td>5000</td>
<td>1</td>
<td>0.0326093</td>
<td>57.8268</td>
</tr>
<tr>
<td>10000</td>
<td>1</td>
<td>0.0436944</td>
<td>61.2117</td>
</tr>
<tr>
<td>20000</td>
<td>1</td>
<td>0.0573002</td>
<td>62.0602</td>
</tr>
</tbody>
</table>

5.6 Experiments, Results and Analyses

In this section, we evaluate the performance of the proposed framework, i.e., SecureC2Edit with SSS, in terms of execution time and data expansion factor. We also analyze the operability aspect covering the content loss and overall fault tolerance. This is followed by a security analysis.

5.6.1 Performance Analysis

5.6.1.1 Experimental setup

All experiments were performed on a Windows Laptop with Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz 2.59 GHz processor and 16 GB RAM. The peers are implemented as Java Swing desktop applications and the controller is implemented as Java Threads.

5.6.1.2 Experiments and results

For execution time and expansion factor evaluation, 500 add operations were generated with varying sizes of the initial length of document contents to measure the average execution
Table 5.3: Results of SSS experiment with $\phi = 5$.

<table>
<thead>
<tr>
<th>Doc Length</th>
<th>Expansion Factor</th>
<th>Exec Time (ms, $\phi = 5$)</th>
<th>% Decrease ($\phi = 5$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>0.038883</td>
<td>11.3984</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>0.038914</td>
<td>24.8811</td>
</tr>
<tr>
<td>500</td>
<td>1</td>
<td>0.041623</td>
<td>23.2872</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>0.041942</td>
<td>29.2859</td>
</tr>
<tr>
<td>5000</td>
<td>1</td>
<td>0.049306</td>
<td>36.2332</td>
</tr>
<tr>
<td>10000</td>
<td>1</td>
<td>0.063128</td>
<td>43.9601</td>
</tr>
<tr>
<td>20000</td>
<td>1</td>
<td>0.085313</td>
<td>43.5357</td>
</tr>
</tbody>
</table>

Figure 5.6: Execution time comparison between AES and SSS with $\phi = 1$ and $\phi = 5$.

time and expansion factor. Three sets of experiments were conducted for each document length, one with AES, the second using SSS with a static set of coefficients ($\phi = 1$), and the third using SSS with multiple sets of coefficients ($\phi = 5$). The execution time reported is a simple average of the execution time to incorporate an edit and perform AES on the document, in the case of the AES experiment, and in the case of SSS, the time to create shares of each edit and incorporate them in their respective controller shares. The expansion factor represents a comparison between the length of the resulting ciphertext or the share text and the length of the original plaintext. Tables 5.1 and 5.3 show the results of the three experiments. $\phi = 1$ secret sharing provides an average performance increase of 56.01% over
varying document lengths, whereas $\phi = 5$ secret sharing provides an average performance increase of 30.37%. The execution time of AES increases significantly at higher document lengths, whereas secret sharing stays relatively fast. The use of AES results in an average expansion factor of 1.004 per CSP across various document lengths, whereas secret sharing experiments exhibit an expansion factor of 1 per CSP. Figure 5.6 shows a comparison between execution time of AES versus $\phi = 1$ and $\phi = 5$ secret sharing across varying document lengths. In a collaborative document editing setting, where multiple co-authors make frequent edits and the document needs to be committed to the CSP frequently, secret sharing can provide a significant performance increase compared to AES.

5.6.2 Operability Analysis

The operability of the system is analyzed for reliability in terms of the frequency of incorporating edits and committing them to CSPs from the controller, in a secure manner. When a peer generates an edit, it is propagated to the controller and subsequently to all other peers. The controller then synchronizes edits it receives between the controller text and the save shadow. Whenever a new edit is incorporated into the save shadow, shares for that edit can be created and incorporated into the controller shares and then committed to their respective CSPs. This entire process can be conducted for each character edit made to the contents of the collaborative document since creating shares of one character has a low cost when compared with using AES to employ security to user contents. The cost of using AES, especially when the contents of a document become significantly large becomes extremely high. Thus, in solutions using AES, documents are saved at the end of the editing session or on need basis during an editing session. However, when using secret sharing, this is not the case. The system can remain synchronized on every character edit and also have all the edits be synchronized with the CSPs. Another advantage of employing secret sharing is that since contents are kept in sync with CSPs, the chances of losing content become minimal. While using AES, and saving documents either at the end of editing sessions or on-demand, the chances of losing edits made since the last save become higher. In case the controller is not able to save edits made to contents, all the edits made since the last save can potentially be lost, which is prevented when using secret sharing.
5.6.3 Security Analysis

For security analysis, in addition to the assumptions (A1 and A2) made in Chapter 4, the following assumptions are made in the proposed framework’s model:

A3 The CSP is semi-honest in the sense that it maintains access control honestly but may be curious about user data.

A4 The CSPs do not collude with each other.

A5 The peers are honest entities and do not use the framework maliciously.

Assumption A1 is also applicable to current collaborative editing applications. The underlying assumption of secret sharing is that the parties holding shares do not collude, thus assumption A4 is established. With the aforementioned assumptions, we establish the following threat model:

Semi-honest CSP. The CSP is assumed to be semi-honest. It provides storage services to users and adequate security measures to defend against external adversaries. The CSP is also responsible for maintaining access control for the documents it stores. It, however, may choose to act maliciously in the sense that it may become curious about user data and try to read it.

Honest Peers. In the proposed framework, the co-authors/peers of a collaborative document are considered to be honest. As mentioned in Assumption A2, peers possess adequate security mechanisms to defend against external adversaries and attacks. The peers are authorized users of the system and are assumed to have access to the document they collaborate on.

External Adversaries. External adversaries may try to get unauthorized access to a collaborative document and are assumed to be malicious entities. An external adversary may try to subvert the security protocols established by the CSP to gain access to stored documents.

Security analysis of the proposed framework is presented using the following theorem and propositions:
**Theorem 5.1** The system is considered $\epsilon$-secure if the adversary can know the contents of the document with a negligible probability, $\epsilon$.

**Proof**: According to the information theoretic security property of SSS, the probability that a given character $c$ in the plaintext document is $c'$ in the encrypted document is equiprobable, with $P(c = c') = 1/q$, where $q = 127$. Also, let a word $w$ contain $y$ number of characters on an average. The probability that a given word $w$ in the plaintext document is $w'$ in the encrypted document is also equiprobable, with $P(w = w') = 1/(q^y)$. Furthermore, let a document $d$ contain $g$ number of words on an average. The probability of a given plaintext document $d$ being $d'$ when encrypted also ends up being equiprobable, and is given by, $P(d = d') = (1/q^{gy}) \approx \epsilon$.

Example: A document $d$ containing $g = 100$ words with an average $y = 5$ characters per word, $P(d = d') = (1/127^{5\times100}) \approx \epsilon$.

**Proposition 5.2** The proposed framework is secure against unauthorized reads from semi-malicious entities.

**Proof**: According to the assumptions stated above, the CSP is a semi-malicious entity. The CSP possesses read access to the stored documents, but not write and delete access. In the proposed framework, before the contents of a document are committed to the CSP, are converted into shares by the controller. Upon a save request, these shares are then saved to the CSPs. Upon retrieval of a document, the contents of the document are reconstructed from shares on the controller. Thus, the contents of the document are never exposed to the CSP in plaintext form. This results in CSPs never gaining access to plaintext data and hence cannot make sense of the contents.

**Proposition 5.3** The proposed framework is $\epsilon$-secure against unauthorized reads from semi-honest CSP.

**Proof**: In the proposed framework, before the contents of a document are committed to the CSP, are converted into shares by the controller. Upon a save request, these shares are then saved to the CSPs. Thus, the contents of the document are never exposed to the CSP in plaintext form. Following Theorem 5.1, the CSP can know the content with a negligible ($\epsilon$) probability only, enabling it to be $\epsilon$-secure.
**Proposition 5.4** The proposed framework is secure against malicious external adversaries.

**Proof:** An external malicious adversary may try to gain unauthorized access to a collaborative document at the CSP or at a peer. As per Assumption A1, the CSPs possess adequate security mechanisms to prevent external adversaries from gaining access to documents stored on their cloud. Thus, an external adversary cannot gain access to a document attacking the CSPs. Secondly, it may also try to gain access to the contents of a document by attacking peers. However, according to Assumption A2, the peers possess sufficient security mechanisms to protect themselves against attacks from malicious external adversaries.

### 5.6.4 Encryption Process Comparison

In the framework presented in this chapter, we employ SSS as the security mechanism, whereas, in SecureC2Edit (the work presented in Chapter 4), the AES was used. We now compare and contrast the two security mechanisms. The use of SSS at least doubles the storage cost of collaborative documents due to the requirement of using a minimum of two CSPs. However, SSS allows the system to be synchronized on every character edit. This is beneficial in minimizing the content loss but at the cost of extra storage requirements. The use of AES exhibits an average expansion factor of 1.004, as determined in [14], thus requiring less storage. Both security mechanisms provide adequate security of user contents and the choice depends on application-specific requirements and time and space constraints.

### 5.7 Chapter Summary

In this chapter, we explored the use of secret sharing to maintain the confidentiality of user data in a collaborative document and presented the third framework, i.e., SecureC2Edit with SSS as an encryption mechanism, and hence, accomplished **Objective 3** of the thesis. We established that using SSS provides an average increase of 56.01% in performance over AES with a single set of coefficients and an average performance increase of 30.37% with multiple sets of coefficients, while not requiring maintenance and distribution of symmetric keys as in the case of AES. This chapter also discussed the incorporation of keyword-based search with the proposed framework and presented an operability and security analysis.
CHAPTER 6
Summary, Conclusions and Future Research Directions

This chapter presents a summary of the work presented in this thesis (in Section 6.1). Further, conclusions drawn from this research are outlined in Section 6.2. Finally, the chapter ends with a deliberation on research directions that can be explored in the future, in Section 6.3.

6.1 Thesis Summary

The thesis begins with discussing the problem of secure collaborative editing and the challenges associated with it in Chapter 1. We discuss the rise of cloud service providers (CSPs) and the privacy concerns associated with it. Potential attacks and terms of service that can compromise user data are also discussed. In addition, this chapter states the overarching goal and three specific objectives of the thesis and outlines the contributions. This is followed by a background and literature review in Chapter 2 where we discuss application architectures, encryption algorithms, and collaborative editing frameworks. We then discuss challenges associated with cloud data security followed by a detailed analysis of the state-of-the-art in the area of secure collaborative editing.

Subsequently, corresponding to the three objectives set in this thesis, three frameworks are proposed in the next three chapters. For Objective 1, in Chapter 3 we present SecureCEdit, which is a client-server-based secure collaborative editing framework that bridges some of the research gaps and provides collaborative access. We also present an experimental evaluation of the framework along with security analysis. Next, for Objective 2, in Chapter 4 we present, SecureC2Edit, which is a structured peer-to-peer (P2P) secure collaborative and concurrent editing framework that is capable of both collaborative access and concurrent editing. We also provided a detailed evaluation of the proposed framework. Lastly, for Objective 3, in Chapter 5 we present a variation on SecureCE2Edit that uses Shamir’s Secret Sharing (SSS) instead of AES. We also discuss the pros and cons of both variations and discuss potential use cases for both variations. Furthermore, we evaluate the
framework for security, performance, and operability and present a security analysis.

6.2 Conclusions

Based on the work presented in this thesis, we draw the following conclusions:

1. **Security-focused re-design of collaborative editing frameworks is required keeping in view the semi-honesty of CSPs.** Existing collaborative editing frameworks were designed to work in the plaintext domain and need to be either heavily adopted to work in the encrypted domain or re-designed from the ground up with security in mind. Browser-extension-based solutions tried to do this by reverse engineering communication protocols between online editors and cloud storage. However, the solutions were inadequate as any change in the communication protocols resulted in the solutions becoming obsolete.

2. **A structured P2P framework for collaborative editing has more general-purpose feasibility than client-server.** Secure collaborative editing solutions are feasible in both client-server and peer-to-peer architectures, as explored in this thesis. Both architectures have their own merits and limitations, as discussed in detail. The choice of architecture can be made depending on the use case. That being said, a structured P2P architecture has more general-purpose feasibility.

3. **SSS is a better security mechanism than AES for collaborative editing frameworks in terms of operability and performance if having multiple CSPs is feasible.** Key-based encryption mechanisms like symmetric encryption and keyless mechanisms like secret sharing are feasible choices for methods of employing security in secure collaborative editing solutions. We have demonstrated the use of AES, and SSS in the two variations of SecureC2Edit in this thesis and proved the security of both variations. We have also discussed the advantages and disadvantages of using both classes of security mechanisms and provided suitable use cases for both. The advantage of using symmetric encryption is that although the execution time of AES and SSS is comparable on smaller documents, it requires only one CSP; however, SSS scales extremely well with larger document lengths but requires the use of more than one CSP.
6.3 Future Research Directions

As an extension of the work done in this thesis, the following list of works can be explored in the future to make it more secure and eliminate some of the established assumptions.

- **Feasibility and Viability of Other Collaborative Editing Frameworks.**
  Since Operational Transformation (OT) and Conflict-free Replicated Data Types (CRDTs) possess some underlying disadvantages for collaborative and concurrent editing, we chose Differential Synchronization (DS) as the collaborative editing framework for SecureC2Edit. OT does not support sending batch edits in a single request while CRDTs provide eventual consistency thus not being suitable for fast concurrent editing for multiple users. However, there are merits to using both of these schemes. OT is the most popular editing framework in the area of collaborative editing. CRDTs are implemented in such a way that they take care of reaching a consistent state themselves and thus do not require explicit communication with all the peers. We would like to explore the feasibility and viability of other editing frameworks by eliminating the mentioned weaknesses of the schemes.

- **Secure Communication Across Semi-secure P2P Connections and Semi-malicious Peers.**
  One of the assumptions established in SecureC2Edit is that the peers are honest and the network connections between them and the controller are assumed to be secure. Any peer connected to the doc-subnet is assumed to be honest. This eliminates the need to authenticate the peers within the doc-subnet and verify the messages that are sent by them. We would like to explore the idea of relaxing this assumption by assuming peers to be semi-malicious in our new threat model as well as introduce the idea of P2P connections that are not secure and address the challenges that are associated with them.

- **Other Encryption Frameworks.** In SecureC2Edit we use AES to encrypt the content of collaborative documents. A side effect of using AES is that the entire document needs to be encrypted every time it is saved to the CSP, in order to preserve its avalanche effect. This introduces additional overhead. We also explore the idea of using SSS instead of AES and compare its performance to the existing model. Since using
SSS introduces the requirement of another CSP, we would also like to explore other mathematical functions that would enable us to have only one CSP in the framework while giving us the capability of not encrypting the entire document each time it is saved while providing similar or potentially better security.
BIBLIOGRAPHY


[61] Saptaparni Kumar, Haochen Pan, Roger Wang, and Lewis Tseng, *LiteDoc: Make collaborative editing fast, scalable, and robust*, Proceedings of the IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops) (Austin, TX, USA), 2020, pp. 1–6.


[96] Priyanka Singh, Balasubramanian Raman, Nishant Agarwal, and Pradeep K. Atrey, *Secure cloud-based image tampering detection and localization using pod number sys-


[99] Chengzheng Sun, David Sun, Agustina, and Weiwei Cai, Real differences between OT and CRDT under a general transformation framework for consistency maintenance in co-editors, CoRR abs/1905.01518 (2019).

[100] David Sun, Chengzheng Sun, Agustina, and Weiwei Cai, Real differences between OT and CRDT in building co-editing systems and real world applications, CoRR abs/1905.01517 (2019).


