THESIS

DIGITAL SIGNATURES TO ENSURE THE AUTHENTICITY AND INTEGRITY OF SYNTHETIC DNA MOLECULES

Submitted by

Diptendu Mohan Kar

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Spring 2019

Master's Committee:

Advisor: Dr. Indrajit Ray
Co-Advisor: Dr. Indrakshi Ray

Dr. Leo R Vijayasarathy
Dr. Jean Peccoud

Copyright by Diptendu Mohan Kar 2019

All Rights Reserved

ABSTRACT

DIGITAL SIGNATURES TO ENSURE THE AUTHENTICITY AND INTEGRITY OF SYNTHETIC DNA MOLECULES

DNA molecules are increasingly being synthesized in the laboratory some of which are licensed intellectual properties (IP). Synthetic samples are shared between academic labs, ordered from DNA synthesis companies, and manipulated for a variety of different purposes, mostly to study their properties and improve upon them. However, it is not uncommon for a sample to change hands many times with very little proof of origin information encoded in the sample. This poses significant challenges to the original inventor of the DNA molecules, trying to protect her IP rights. More importantly, following the anthrax attacks of 2001, there is an increased urgency to employ microbial forensic technologies to trace and track agent inventories. However, in the realm of synthesized DNA such attribution is next to impossible with existing technology. We look into how a digital signature of the DNA molecule can be embedded into the molecule itself to solve this problem (if at all). We encounter several challenges that we do not face in the digital world. These challenges arise primarily from the fact that the physical size of the DNA molecule can affect its properties, random mutations can accumulate in the DNA samples over time, DNA sequencers can sequence (read) DNA erroneously and is still relatively expensive (which means that laboratories would prefer not to read and re-read a DNA to get error-free sequences).

In this research, we try to address these challenges and present a digital signature technology that can be applied to synthetic DNA molecules in living cells.

ACKNOWLEDGEMENTS

I also would like to express my sincere gratitude to my advisors, Dr. Indrajit Ray, and Dr. Indrakshi Ray, for their continuous support of my master's study and research, for their patience, motivation, kindness, and immense knowledge. Their guidance helped me in all the time of research and writing of this thesis. I could not have imagined having better advisors for my graduate study. Their advice and support on both research as well as on my career have been invaluable.

I am also grateful to Dr. Leo Vijayasarathy, for his reviews, insightful comments, patience, kindness, and encouragement.

My sincere thanks also go to Dr. Jean Peccoud and Jenna Gallegos for providing me an opportunity to work on this wonderful area of research. Without their precious advice and support, it would not be possible to conduct this research. I appreciate all their contributions of time, ideas, domain knowledge and funding to make my graduate experience productive and stimulating.

I am immensely grateful to Ibrahim Meftah Lazrig for being such a wonderful mentor, guide, collaborator and friend. He has helped me to gain a lot of knowledge in the area of applied cryptography. Without his guidance, this would not have been possible. Thank you for being patient and answering any question that I would ask you.

Last but not least, my fellow students with whom I have had the pleasure to hang-out or work with, our security research group (DBSec).

DEDICATION

To my parents

To my love

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

DNA synthesis is becoming more commonplace, and samples containing DNA frequently change hands within the life sciences community. Samples are shared between academic labs, ordered from DNA synthesis companies, and manipulated for a variety of different purposes. It is becoming increasingly difficult to properly attribute DNA molecules to their original sources.

Example use cases where attribution of a DNA molecule to its originator is important are as follows: (i) Following the anthrax attacks of 2001, there has been an increased urgency to employ microbial forensic technologies to trace and track agent inventories. Many of these biological agents are created or manipulated in laboratories. (ii) Academic laboratories and biotech companies frequently treat synthetic DNA as licensed intellectual property that needs to be protected; the first step towards such protections is successfully ensuring attribution. (iii) In the future, if synthetic genes are used in gene therapy based medical treatment, such attribution could (a) readily inform the user about matters related to the therapy, and/or (b) serve as some measure of the quality of the therapy, Ãă la brand name versus generic drugs. (iv) Recently, a DNA-based security exploit was demonstrated as a proof of concept, where synthetic DNA was used to attack a DNA sequencer that was deliberately modified with a vulnerability [1]; ensuring source attribution could help mitigate similar attacks on DNA sequencers.

The exact sequence of synthetic DNA molecules is generally documented electronically. Thus, one potential way (although indirect) of achieving such attribution is to include the origin information in the electronic document. However, the association between physical sequences and their electronic documentation is very loose [2, 3]. It is not uncommon for a sample to change hands many times with only vague descriptions of where it originated or the exact sequences being shared. On the other hand, simply documenting samples more effectively can create security and intellectual property risks [4]. For instance, if the full sequence of pathogenic viral vectors was freely available, that information could be used with malicious intent. Or, for example, if a

bio-tech company wants to keep some sequence information confidential to protect their IP rights, they may choose to exclude such information from the documentation. Regardless, the source of a given sample cannot be verified using only a careful documentation of the DNA sequence.

In the digital realm, the problem of authenticating a document or software while still withholding proprietary or sensitive information is achieved by digital signatures. Digital signatures are used to authenticate the source of a digital file and to confirm that the file has not been changed since the originator applied the signature. Our hypothesis is that digital signature technology can be used to solve the DNA attribution problem. Additionally, the same technology, can be used to detect mutations in DNA samples. Mutations are known to occur randomly at low frequencies, and they can compromise the function of DNA molecules. Recent works in DNA cryptography, such as, encoding arbitrary information in DNA [5], and embedding watermarks in synthetic DNA [6–10], gives us some degree of confidence in the validity of this hypothesis.

To solve the problem of tracing the source of synthesized DNA molecules and confirming their identity and integrity, we have developed a system for generating digital signatures for molecules of DNA in living cells. Specifically, we have used Shamir's Identity-based Signature (IBS) scheme [11]. For the unique identifier string of the originator, we use Open Researcher and Contributor ID's (ORCID) – https.orcid.org. ORCID is a non-profit organization which uniquely identifies researchers using a 16 digit number. Many funding agencies require researchers to register for an ORCID, and scholarly journals request that authors identify themselves using their ORCID. The generated signature bits are converted to the four letters A, C, G, and T, which represent the four nucleotide building-blocks of DNA. The sequence can then be synthesized and inserted into the original DNA molecule. When this signed molecule is shared, a receiver can sequence the signed molecule to verify that it was shared by an authentic sender and that the sequence of the original molecule has not been altered or tampered with.

While the use of these techniques in the digital world is quite common, applying them to DNA required several creative adjustments. The foremost challenge we faced arose from the physical size of the DNA sequence encoding the signature. Adding extraneous sequences to a DNA

molecule can impact it's function or stability. It was, thus, important to minimize the size of the added sequence in order to decrease the likelihood that the biological function of the signed molecule would be effected and to decrease the cost of synthesizing the signature. This restricted our ability to use more well-known signature schemes as well as larger key sizes for signatures.

The second challenge was accounting for DNA mutations. In a DNA sample, mutations occur randomly at low frequencies, and, as a result, there is a non-trivial possibility that a signed molecule could undergo a mutation between the time it is signed and when it is validated. Mutations could affect not only the original DNA molecule but also the signature. In both cases, the signature validation will fail even if the molecule is sent by the correct authority and the original sequence was correct during the process of signature generation. Mutations are beyond the control of any authority and the relative impact of any given mutation can vary. In order to address this concern, we included error correction codes to detect mutations in the signed DNA molecule. Error correction codes are prevalent in digital storage such as CD/DVD. It is possible to use the same techniques to provide a reliable reconstruction of the original sequence for comparison, provided a small number of changes have occurred. Our application of error correction codes to DNA could also be used to ensure the integrity of digital information stored in DNA molecules.

Finally, while digital signatures provide a way to verify the source and integrity of a DNA sample, there is additional information about the DNA sequence that would be very useful to the recipient of a signed DNA sample. For example, the exact location of features within the sequence, such as a certain gene, will still be unknown to the recipient. We thus developed a method to link the physically signed DNA molecule with its digital representation, which contains the sequence and its features with explanations.

Threat Model: For this work, we assume a polynomial-time adversary, Mallory, who is trying to forge the signature of a reputed synthesized DNA molecule creator, Alice. Alice is trying to protect her IP rights / reputation as she distributes DNA molecules synthesized by her to researcher Bob. If the attacker, Mallory, is able to forge the signature of Alice then: (a) Mallory can replace the actual DNA created by Alice with her own but keep the signature intact. (b) Mallory can create her

own DNA molecule and masquerade as Alice to sign it. (c) Mallory can modify parts of the signed DNA molecule created by Alice. Therefore, for our proposed signature scheme to be secure, we need to show that no polynomial-time adversary can forge a genuine signature without knowing the secret used to sign.

The rest of the thesis is organized as follows. Chapter 2 gives a small overview of the biological domain which forms the backdrop of our scheme, a brief overview of digital signatures, identity-based signatures and Reed-Solomon Code. Chapter 3 comprises of the related works in this field of research. Chapter 4 contains the challenges we faced and our design decisions. In Chapter 5 Section 5.1 we present our vanilla signature scheme and how the digital signature of the DNA is embedded in the physical molecule. Section 5.2 describes the proof of security of our proposed signature scheme. Section 5.3 describes the workflow that we have to follow in order to validate the signed physical DNA molecule. Section 5.4 discusses how error correction code is used to address the problem of DNA mutation. In Section 5.5, we describe the process of strongly tying the digital document corresponding to the signed DNA molecule. In chapter 6 we describe the signature generation and validation procedure of the developed application. In chapter 7 we describe the experimental procedures conducted and their results to verify if the proposed solution works in real world. In chapter 8 we describe some improvements upon our initial work. Finally, we conclude in chapter 9 by briefly talking about validation of our scheme and discussing some open problems that we can pursue in this area.

Chapter 2

Background

In this chapter, we discuss in brief the biological domain that is the backbone of our work. Next, we describe an overview of digital signatures and identity-based digital signatures. Finally, we discuss an overview of Reed-Solomon Codes.

2.1 Domain Background

As a first step, we have applied digital signatures to plasmids. A plasmid is a small DNA molecule within a cell that is physically separated from the chromosomal DNA and can replicate independently. They are most commonly found as small circular, double-stranded DNA molecules in bacteria. In nature, plasmids often carry genes that may benefit the survival of the organism, for example, antibiotic resistance. While the chromosomes are big and contain all the essential genetic information for living under normal conditions, plasmids usually are very small and contain only additional genes that may be useful to the organism under certain situations or particular conditions. Artificial plasmids are widely used as vectors in molecular cloning, serving to drive the replication of recombinant DNA sequences within host organisms. The reason for starting with plasmids are -

- Plasmids can be isolated in large quantities.
- We can cut and splice them, adding whatever DNA we choose.
- We can put them back into bacteria, where they'll replicate along with the bacteria's own DNA.
- We can isolate them again - getting billions of copies of whatever DNA we inserted into the plasmid!

Plasmids are generally limited to sizes of 2.5-20 kilobases (each letter of the genetic code A-C-G-T is 1 base).

The sequences that make up a plasmid are often documented electronically. Molecular biologists are now equipped with automated DNA sequencers that identify the pattern of bases in a physical DNA sample and document the sequence in a digital file called a fasta file (`.fasta`). A sample fasta file is shown in Figure 2.1. The sequences can then be converted to annotated files such as `.dna`, or `.gb` files that include information about what genetic features are included in the plasmid. A sample genbank (`.gb`) file is shown in Figure 2.2. Each of these files has a specific format which denotes the sequences together with other pieces of information including the location of features such as coding sequences (CDS), origin of replication, etc. Sequence-manipulation softwares such as SnapGene (http://www.snapgene.com) convert sequences into maps of plasmid features. Figure 2.3 depicts a map generated in SnapGene of the same digital DNA file depicted in Figure 2.2. Features (shown as colored block arrows in Figure 2.3) can be added manually or identified automatically by searching within the SnapGene database for common features. Not all of the sequences contain features. There are substrings or subsequences that do not have any known biological function. Biologists can add other DNA sequences in these areas with reasonable confidence that the added sequences will not disrupt the activity of any existing features.

Although the sequences that make up a plasmid can be documented electronically, the electronic sequence file associated with a physical DNA sample is seldom shared along with the sample it represents [12]. In life sciences manuscripts, plasmids are generally described one of four ways. Most often, the main features of the plasmid relevant to the publication are broadly explained (ie. "A plasmid containing gene X was used..."). Sometimes there is a more thorough description of how the plasmid was constructed included in the methods section (ie. "Gene X was inserted into a commercial plasmid between Origin Y and antibiotic resistance gene Z"). Full plasmid maps (as shown in Figure 2.3) are very rarely included in published manuscripts, and inclusion of the full sequence – which you would need to validate the plasmid – is even more rare. Additionally, it is not uncommon for a plasmid to be shared multiple times between many labs until it is no

longer clear who made the original plasmid. Even within a lab, it is often difficult to track down the digital sequence file associated with a plasmid if the person who constructed it is no longer an active member. The ability to validate a physical DNA sample without having access to the digital sequence file associated with it would, thus, be extremely valuable to the life sciences community.

We propose digital signatures as a strategy for encoding the ability to validate a physical DNA sample within the DNA itself. Once the sequences are in a digital file, we can apply digital signatures on the extracted sequence (message). The signature bits are then converted to ACGT sequence as A-00, C-01, G-10, and T-11 and added to the original sequence.

```
>Sequence_1 assembly1
CCCTAAACCCTAAACCCTAAACCCTAAACCTCTGAATCCTTAATCCCTAAATCCCTAAAT
CTTTAAATCCTACATCCATGAATCCCTAAATACCTAATTCCCTAAACCCGAAACCGGTTT
CTCTGGTTGAAAATCATTGTGTATATAATGATAATTTTATCGTTTTTATGTAATTGCTTA
TTGTTGTGTGTAGATTTTTTAAAAATATCATTTGAGGTCAATACAAATCCTATTTCTTGT
GGTTTTCTTTCCTTCACTTAGCTATGGATGGTTTATCTTCATTTGTTATATTGGATACAA
GCTTTGCTACGATCTACATTTGGGAATGTGAGTCTCTTATTGTAACCTTAGGGTTGGTTT
ATCTCAAGAATCTTATTAATTGTTTGGACTGTTTATGTTTGGACATTTATTGTCATTCTT
>Sequence_2
CCCTAAACCCTAAACCCTAAACCCTAAACCTCTGAATCCTTAATCCCTAAATCCCTAAAT
CTTTAAATCCTACATCCATGAATCCCTAAATACCTAATTCCCTAAACCCGAAACCGGTTT
CTCTGGTTGAAAATCATTGTGTATATAATGATAATTTTATCGTTTTTATGTAATTGCTTA
TTGTTGTGTGTAGATTTTTTAAAAATATCATTTGAGGTCAATACAAATCCTATTTCTTGT
GGTTTTCTTTCCTTCACTTAGCTATGGATGGTTTATCTTCATTTGTTATATTGGATACAA
GCTTTGCTACGATCTACATTTGGGAATGTGAGTCTCTTATTGTAACCTTAGGGTTGGTTT
ATCTCAAGAATCTTATTAATTGTTTGGACTGTTTATGTTTGGACATTTATTGTCATTCTT
```

Figure 2.1: Sample fasta (.fasta) file

Once the signed sequence is obtained by adding the signature to the original sequence, it can be outsourced to a gene synthesis company that will synthesize the signed DNA and send it. The signature alone can be synthesized and inserted into the original molecule, or the entire plasmid can be synthesized including the signature to eliminate the need for any downstream assembly.

```
LOCUS       Exported                2686 bp ds-DNA     circular SYN 24-NOV-2013
DEFINITION  Standard E. coli vector with a multiple cloning site (MCS) for DNA
            cloning. The MCS is reversed in pUC18.
ACCESSION   .
VERSION     .
KEYWORDS    pUC19
SOURCE      synthetic DNA construct
  ORGANISM  synthetic DNA construct
REFERENCE   1  (bases 1 to 2686)
  AUTHORS   Yanisch-Perron C, Vieira J, Messing J.
  TITLE     Improved M13 phage cloning vectors and host strains: nucleotide
            sequences of the M13mp18 and pUC19 vectors.
  JOURNAL   Gene 1985;33:103-19.
  PUBMED    2985470
REFERENCE   2  (bases 1 to 2686)
  AUTHORS   New England Biolabs
  TITLE     Direct Submission
  JOURNAL   Exported Nov 10, 2017 from SnapGene 4.1.0
            http://www.snapgene.com
COMMENT     See also GenBank accession L09137.
FEATURES             Location/Qualifiers
     source          1..2686
                     /organism="synthetic DNA construct"
                     /lab_host="Escherichia coli"
                     /mol_type="other DNA"
     CDS             complement(146..469)
                     /codon_start=1
                     /gene="lacZ"
                     /product="LacZ-alpha fragment of beta-galactosidase"
                     /label=lacZ-alpha
                     /translation="MTMITPSLHACRSTLEDPRVPSSNSLAVVLQRRDWENPGVTQLNR
                     LAAHPPFASWRNSEEARTDRPSQQLRSLNGEWRLMRYFLLTHLCGISHRIWCTLSTICS
                     DAA"
     primer_bind     379..395
                     /label=M13 fwd
                     /note="common sequencing primer, one of multiple similar
                     variants"
     misc_feature    396..452
                     /label=MCS
                     /note="pUC19 multiple cloning site"
                     EPELNEAIPNDERDTTMPVAMATTLRKLLTGELLTLASRQQLIDWMEADKVAGPLLRSA
                     LPAGWFIADKSGAGERGSRGIIAALGPDGKPSRIVVIYTTGSQATMDERNRQIAEIGAS
                     LIKHW"
     promoter        complement(2487..2591)
                     /gene="bla"
                     /label=AmpR promoter
ORIGIN
        1 tcgcgcgttt cggtgatgac ggtgaaaacc tctgacacat gcagctcccg gagacggtca
       61 cagcttgtct gtaagcggat gccgggagca gacaagcccg tcagggcgcg tcagcgggtg
      121 ttggcgggtg tcggggctgg cttaactatg cggcatcaga gcagattgta ctgagagtgc
      181 accatatgcg gtgtgaaata ccgcacagat gcgtaaggag aaaataccgc atcaggcgcc
      241 attcgccatt caggctgcgc aactgttggg aagggcgatc ggtgcgggcc tcttcgctat
      301 tacgccagct ggcgaaaggg ggatgtgctg caaggcgatt aagttgggta acgccagggt
      361 tttcccagtc acgacgttgt aaaacgacgg ccagtgaatt cgagctcggt acccggggga
      421 cctctagagt cgacctgcag gcatgcaagc ttggcgtaat catggtcata gctgtttcct
      481 gtgtgaaatt gttatccgct cacaattcca cacaacatac gagccggaag cataaagtgt
      541 aaagcctggg gtgcctaatg agtgagctaa ctcacattaa ttgcgttgcg ctcactgccc
      601 gctttccagt cgggaaacct gtcgtgccag ctgcattaat gaatcggcca acgcgcgggg
      661 agaggcggtt tgcgtattgg gcgctcttcc gcttcctcgc tcactgactc gctgcgctcg
      721 gtcgttcggc tgcggcgagc ggtatcagct cactcaaagg cggtaatacg gttatccaca
      781 gaatcagggg ataacgcagg aaagaacatg tgagcaaaag gccagcaaaa ggccaggaac
      841 cgtaaaaagg ccgcgttgct ggcgtttttc cataggctcc gcccccctga cgagcatcac
      901 aaaaatcgac gctcaagtca gaggtggcga aacccgacag gactataaag ataccaggcg
      961 tttcccccctg gaagctccct cgtgcgctct cctgttccga ccctgccgct taccggatac
     1021 ctgtccgcct ttctcccttc gggaagcgtg gcgctttctc atagctcacg ctgtaggtat
     1081 ctcagttcgg tgtaggtcgt tcgctccaag ctgggctgtg tgcacgaacc ccccgttcag
     1141 cccgaccgct gcgccttatc cggtaactat cgtcttgagt ccaacccggt aagacacgac
     1201 ttatcgccac tggcagcagc cactggtaac aggattagca gagcgaggta tgtaggcggt
     1261 gctacagagt tcttgaagtg gtggcctaac tacggctaca ctagaaggac agtatttggt
     1321 atctgcgctc tgctgaagcc agttaccttc ggaaaaagag ttggtagctc ttgatccggc
     1381 aaacaaacca ccgctggtag cggtggtttt tttgtttgca agcagcagat tacgcgcaga
     1441 aaaaaaggat ctcaagaaga tcctttgatc ttttctacgg ggtctgacgc tcagtggaac
     1501 gaaaactcac gttaagggat tttggtcatg agattatcaa aaaggatctt cacctagatc
     1561 cttttaaatt aaaaatgaag ttttaaatca atctaaagta tatatgagta aacttggtct
     1621 gacagttacc aatgcttaat cagtgaggca cctatctcag cgatctgtct atttcgttca
     1681 tccatagttg cctgactccc cgtcgtgtag ataactacga tacgggaggg cttaccatct
     1741 ggccccagtg ctgcaatgat accgcgagac ccacgctcac cggctccaga tttatcagca
     1801 ataaaccagc cagccggaag ggccgagcgc agaagtggtc ctgcaacttt atccgcctcc
     1861 atccagtcta ttaattgttg ccgggaagct agagtaagta gttcgccagt taatagtttg
     1921 cgcaacgttg ttgccattgc tacaggcatc gtggtgtcac gctcgtcgtt tggtatggct
     1981 tcattcagct ccggttccca acgatcaagg cgagttacat gatcccccat gttgtgcaaa
     2041 aaagcggtta gctccttcgg tcctccgatc gttgtcagaa gtaagttggc cgcagtgtta
     2101 tcactcatgg ttatggcagc actgcataat tctcttactg tcatgccatc cgtaagatgc
     2161 ttttctgtga ctggtgagta ctcaaccaag tcattctgag aatagtgtat gcggcgaccg
     2221 agttgctctt gcccggcgtc aatacgggat aataccgcgc cacatagcag aactttaaaa
     2281 gtgctcatca ttggaaaacg ttcttcgggg cgaaaactct caaggatctt accgctgttg
     2341 agatccagtt cgatgtaacc cactcgtgca cccaactgat cttcagcatc ttttactttc
     2401 accagcgttt ctgggtgagc aaaaacagga aggcaaaatg ccgcaaaaaa gggaataagg
     2461 gcgacacgga aatgttgaat actcatactc ttcctttttc aatattattg aagcatttat
     2521 cagggttatt gtctcatgag cggatacata tttgaatgta tttagaaaaa taaacaaata
     2581 ggggttccgc gcacatttcc ccgaaaagtg ccacctgacg tctaagaaac cattattatc
     2641 atgacattaa cctataaaaa taggcgtatc acgaggccct ttcgtc
//
```

Figure 2.2: Sample genbank (.gb) file

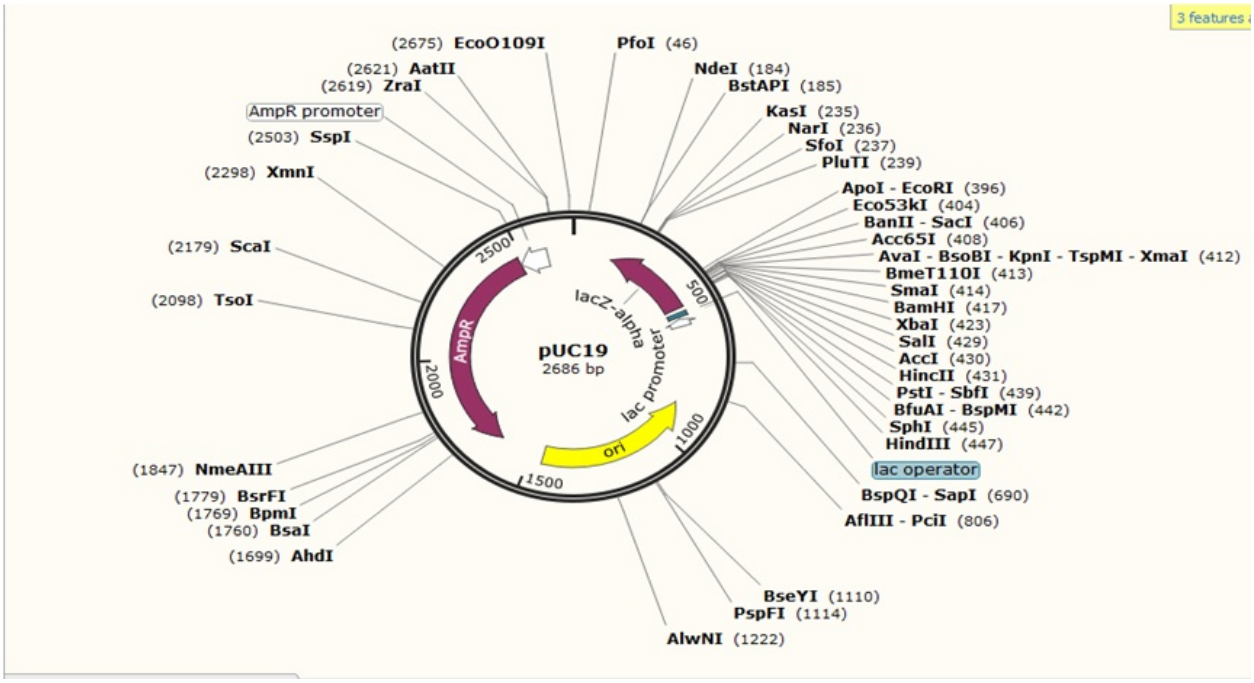

Figure 2.3: SnapGene view of a genbank file

2.1.1 Digital DNA file formats

An automated DNA sequencer provides the digital representation of the sequences present within the physical sample. The output of a DNA sequencer is a fasta (`.fasta`) file as shown in Figure 2.1. This file contains only the raw sequences in the sample. A genbank file (`.gb`) contains the same raw sequences along with annotations. In Figure 2.2, after the word "ORIGIN" the raw sequences are denoted and before that the features are annotated.

Sequence manipulation software such as SnapGene can be used to convert a fasta file to a genbank file and vice versa. When a fasta file is converted to a genbank file, the software searches its database for common annotations. The generated annotations may not be complete or correct every time. Hence, the user has the flexibility to manually add additional annotations that may be required to describe the sample sequence. These manually added annotations are only available to the creator. When the same sample is sent to others, they will sequence it and obtain the fasta file but the genbank file will contain only those annotations that can be automatically generated. In order for the receiver to extract all the feature information for a given plasmid, the creator would need to share the genbank file containing the manually added annotations. In this work, we assume that the sender of the DNA sample will also share the genbank file, although this is very often not the case. (We are looking into ways of eliminating this step such that the genbank file can be generated from the fasta file itself.)

2.1.2 Circular DNA and reverse complement

Plasmid DNA is circular and double-stranded. The sequences represented in a fasta file are the **linear representation of a circular structure**. As a consequence, there is no single set representation of the sequences in a sample. Following sequencing, any cyclic permutation of the sequence is possible. For example, in a fasta file if the sequence is - "ACGGTAA", when the same sample is sequenced again, the fasta file might read as - "TAAACGG".

Furthermore, since DNA is composed of two complimentary, anti-parallel strands, a sequencer can read a sample in both the "sense" or "antisense" direction. The sequence may be represented

in a fasta file in either direction. When the sample is sequenced again, the output might be in the other direction, or what is known as the reverse complement. The reverse complement of "A" is "T" and vice-versa, and the reverse complement of "C" is "G" and vice-versa. The DNA molecule has polarity with one end represented as $5'$ and the other represented as $3'$. One strand adheres to its reverse compliment in anti-parallel fashion. So if the sequence is - " $5'$-ACGGTAA- $3'$ ", the reverse complement is " $3'$-TGCCATT- $5'$ ". The fasta file will represent one strand of the DNA sequence in the $5'$ to $3'$ direction; so the fasta file could read as "ACGGTAA" or "TTACCGT".

By combining these two properties, for a DNA that contains N number of bases, the correct representation of the same sample is $2N$: N cyclic permutations plus each reverse complement.

2.2 Digital Signatures

A digital signature is a mathematical technique used to validate the authenticity and integrity of a message, software or digital document. It is the digital equivalent of a handwritten signature or stamped seal, but offering far more inherent security, a digital signature is intended to solve the problem of tampering and impersonation in digital communications. Digital signatures can provide the added assurances of evidence to origin, identity and status of an electronic document, transaction or message, as well as acknowledging informed consent by the signer.

A digital signature can be used with any kind of message, whether it is encrypted or not, simply so the receiver can be sure of the sender's identity and that the message arrived intact. Digital signatures make it difficult for the signer to deny having signed something (non-repudiation), assuming their private key has not been compromised as the digital signature is unique to both the document and the signer, and it binds them together. Most modern email programs support the use of digital signatures and digital certificates, making it easy to sign any outgoing emails and validate digitally signed incoming messages. Digital signatures are also used extensively to provide proof of authenticity, data integrity, and non-repudiation of communications and transactions conducted over the Internet.

Digital signatures are based on public key cryptography. It was invented in 1976 by Whitfield Diffie and Martin Hellman [13]. It is also called asymmetric encryption because it uses two keys instead of one key (symmetric encryption). The two keys are named **public key** known to everyone and a **private key**, the private key has full control to the key owner and has to keep in a secure environment. A unique element to the public key system is that the public and private keys are related in such a way that only the public key can be used to encrypt messages and only the corresponding private key can be used to decrypt them. Moreover, it is virtually impossible to deduce the private key if you know the public key.

When Bob wants to send a secure message to Alice, he uses Alice's public key to encrypt the message. Alice then uses her private key to decrypt it. For digitally sign documents, Alice needs to obtain a Private and Public Key - a one-time process, it's done by Secured Signing Service while

user registered. The private key isn't shared and is used only by Alice while signing documents. The public key is available for all, used to validate the signatory's digital signature. This public key is again validated or stamped by a certificate authority e.g. Verizon, Google etc. so that Bob can be certain that this is Alice's public key. The certificate authority creates a digital certificate for Alice which certifies that the public key belongs to Alice along with other details. Certificates are similar to ID Document. When you want to identify a user in the system you check his certificate. When Alice sends the signed document to Bob, Bob uses Alice's public key (which is included in the signature within the Digital Certificate) to authenticate Alice's signature and to ensure the document didn't alter after it was signed. A public key infrastructure (PKI) is a system for the creation, storage, and distribution of these digital certificates which are used to verify that a particular public key belongs to a certain entity. The PKI creates digital certificates which map public keys to entities, securely stores these certificates in a central repository and revokes them if needed. The following figure depicts the working principle of a digital signature.

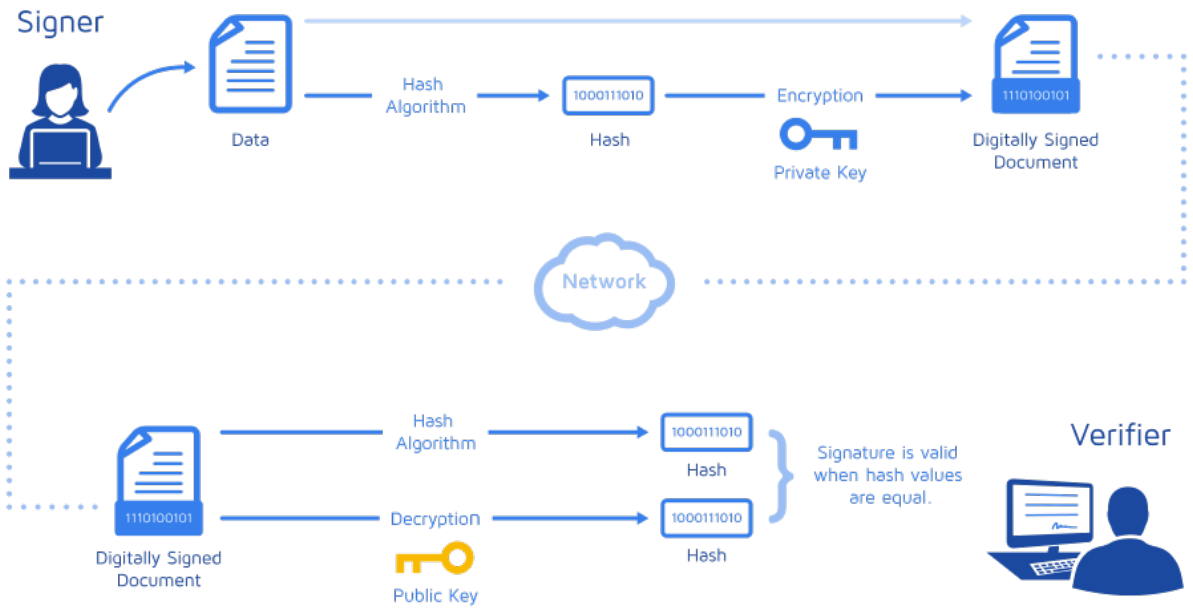

Figure 2.4: Overview of a digital signature

2.3 Identity-Based Digital Signatures

One significant impediment to the widespread adoption of public-key cryptography is its dependence on a public-key infrastructure that is shared among its users. Before secure communications can take place, both sender and receiver must generate encryption and signature key pairs, submit certificate requests along with proof of identity to a Certificate Authority (CA), and receive CA-signed certificates, which they can then use to authenticate one another and exchange encrypted or signed messages. This process can be both time-consuming and error-prone and is especially prohibitive for novice computer users. Although this is simple within a company, senders outside the organization don't have any access to a central directory, so it's more difficult for them to send encrypted/signed messages. Moreover, the process works only if the recipient has already decided to use it and has made a key available. And most people don't have public keys.

In 1984, Adi Shamir, of RSA notoriety, introduced the concept of identity-based cryptography [11]. Its primary innovation was its use of user identity attributes, such as email addresses or phone numbers, instead of digital certificates, for encryption and signature verification. This feature significantly reduces the complexity of a cryptography system by eliminating the need for generating and managing users' certificates. It also makes it much easier to provide cryptography to unprepared users, since messages may be encrypted for users before they interact with any system components. In an identity-based cryptosystem, a unique identifier of the recipient (such as his e-mail address) is used to calculate a public key. A trusted third-party server, called the private-key generator, uses a cryptographic algorithm to calculate the corresponding private key from the public key. In this way, recipients can generate their own private keys directly from the server as needed, and they don't have to worry about distributing their public keys. Ferris Research looked at the costs of one commercial Identity Based Encryption system (from Voltage Security Inc. in Palo Alto, California) and found that total cost of ownership of a typical system is one-third that of a typical public-key system.

Figure 2.5 depicts the steps involved in an identity-based signature scheme. When Alice wants to share a signed message with Bob, the steps involved are as follows:

1. Alice connects to the central authority (PKG) over a secure channel and provides her identity.
2. The central authority after successfully authenticating Alice provides her a private key that is tied to her identity and the master private key.
3. Alice uses this received private key to sign the message. She shares this signed message with Bob.
4. Bob upon receiving the message connects to the central authority and requests for the master public key.
5. The central authority after successfully authenticating Bob provides him with the public key.
6. Bob uses this public key to validate Alice's signature.

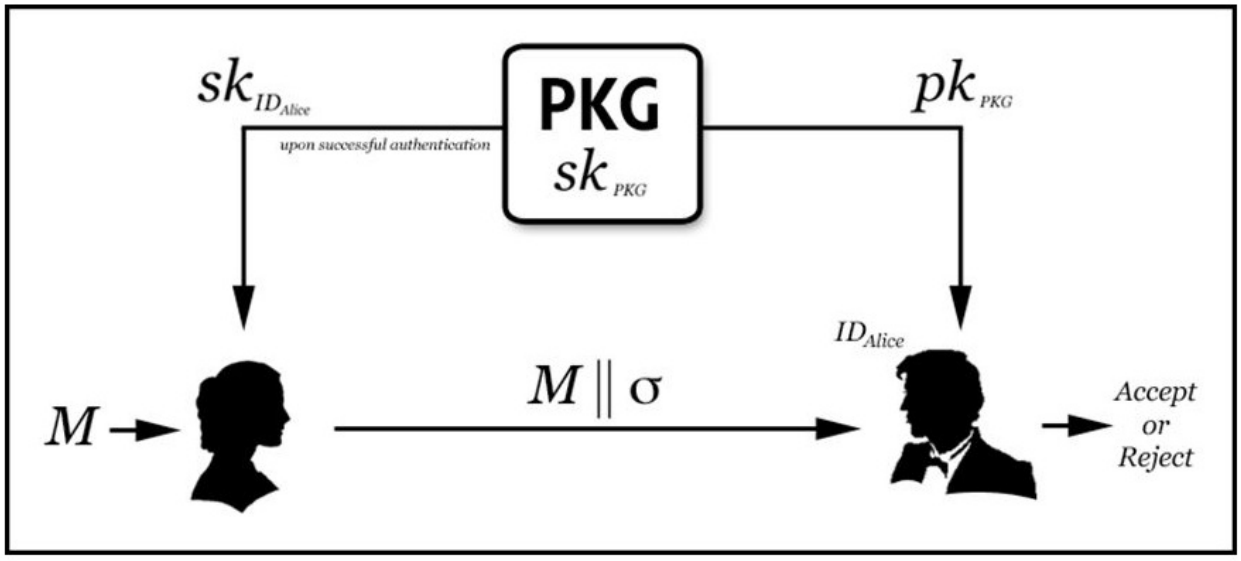

Figure 2.5: Overview of an identity-based digital signature

2.4 Reed-Solomon Codes

Reed-Solomon codes introduced by Irving S. Reed and Gustave Solomon in 1960 [14], are block-based *error correcting codes* in which redundant information is added to data so that it can be recovered reliably despite errors in transmission or storage and retrieval. The error correction system used on CD's and DVD's is based on a Reed-Solomon code. These codes are also used on satellite links and other communications systems. The Reed-Solomon encoder takes a block of digital data and adds extra redundant bits. Errors occur during transmission or storage for a number of reasons (for example noise or interference, scratches on a CD, etc). The Reed-Solomon decoder processes each block and attempts to correct errors and recover the original data. The number and type of errors that can be corrected depend on the characteristics of the Reed-Solomon code.

A Reed-Solomon code is specified as RS(n,k) with s-bit symbols. This means that the encoder takes k data symbols of s bits each and adds parity symbols to make a n symbol codeword. There are $n - k$ parity symbols of s bits each. A Reed-Solomon decoder can correct up to t symbols that contain errors in a codeword, where $2t = n - k$.

The following diagram shows a typical Reed-Solomon codeword (this is known as a Systematic code because the data is left unchanged and the parity symbols are appended):

Figure 2.6: A typical Reed-Solomon codeword

Example: A popular Reed-Solomon code is RS(255,223) with 8-bit symbols. Each codeword contains 255 codeword bytes, of which 223 bytes are data and 32 bytes are for parity. For this code: n = 255, k = 223, s = 8, $2t$ = 32, t = 16. The decoder can correct any 16 symbol errors in the code word: i.e. errors in up to 16 bytes anywhere in the codeword can be automatically corrected. Given a symbol size s, the maximum codeword length (n) for a Reed-Solomon code is $n = 2^s - 1$.

For example, the maximum length of a code with 8-bit symbols (s=8) is 255 bytes. The amount of processing power required to encode and decode Reed-Solomon codes is related to the number of parity symbols per codeword. A large value of t means that a large number of errors can be corrected but requires more computational power than a small value of t.

One symbol error occurs when 1 bit in a symbol is wrong or when all the bits in a symbol are wrong. RS(255,223) can correct 16 symbol errors. In the worst case, 16-bit errors may occur, each in a separate symbol (byte) so that the decoder corrects 16-bit errors. In the best case, 16 complete byte errors occur so that the decoder corrects 16 x 8-bit errors. Reed-Solomon codes are particularly well suited to correcting burst errors (where a series of bits in the codeword are received in error).

Reed-Solomon codes are based on a specialist area of mathematics known as Galois fields or finite fields. A finite field has the property that arithmetic operations (+,-,x,/ etc.) on field elements always have a result in the field. A Reed-Solomon encoder or decoder needs to carry out these arithmetic operations. For a detailed description of the construction and working principle of Reed-Solomon encoding and decoding, an interested reader can refer to [15, 16].

Chapter 3

Related Work

By converting the binary system of 0's and 1's used by computers into the four-letter genetic code (A-C-G-T), encryption algorithms have been developed to store the content of a book, an operating system, and even a movie in synthetic DNA [5, 17]. Computer scientists working in the field of DNA computing have also proposed developing asymmetric encryption schemes using DNA sequences [18]. These pioneering projects demonstrate that DNA synthesis and DNA sequencing technologies are able to translate between DNA molecules and digital information. Our proposed approach is the first application of a digital signature scheme to DNA in a living organism, which will provide a physical significance to a digital signature.

There is a growing body of research concerned with traceability and unique identification of the source of DNA sequences. For instance, it has been proposed that unique watermarks be inserted in the genome of infectious agents to increase their traceability [6]. The synthetic genomics community has demonstrated the feasibility of this approach by inserting short watermarks into DNA without introducing significant perturbation to genome function [7–10]. The use of watermarks has also been proposed in order to identify genetically modified organisms (GMOs) or proprietary strains. Our digital signature-based approach will provide a much greater level of security than watermarking because a watermark is independent of the sequence it is attached to (only changes to the watermark itself would be detectable), and watermarks are easily counterfeited.

Heider et al. [19] describe DNA-based watermarks using DNA-Crypt algorithm. This technique is applicable to provide proof of origin to a DNA molecule. However, there are some limitations. First, the watermark is generated from any binary data and added to the original sequence. The watermark is independent of the original sequence hence provides no integrity of the actual DNA sequence. If the watermark locations can be found, the original molecule sequence can be changed by others keeping the watermark sequence unaltered and sent to the receiver. The receiver will trust that it came from the sender whose watermark is present in the DNA. Second, if

an attacker or other legitimate competing user/organization knows the binary data that is used to generate the watermark, they can generate their own arbitrary DNA and put in the watermark to malign the actual user/organization. Due to these same reasons, in the digital realm, we do not just rely on watermarks because of the limited security that it can provide. Instead, we use digital signatures as it provides much better and stronger security guarantees. Finally, DNA-Crypt uses some symmetric key encryptions like AES, Blowfish to encrypt the binary data that is used to create the watermark. Hence, these keys have to be transmitted to the receiver who will validate the watermark. DNA-Crypt has options to export and import keys and exchange them with receivers. But the receiver now has the secret key that was used to generate the watermark and can masquerade as the originator of the DNA. Non-repudiation becomes a problem too. Although DNA-crypt mentions that RSA algorithm can be used as an asymmetric scheme, the developers do not talk about the challenges involved when using RSA since the length of the watermarks will be dependent on the RSA security parameter.

Chapter 4

Challenges and Design Decisions

In the digital realm, application of digital signatures nowadays is trivial. But when a signature is embedded within a physical DNA molecule we face a couple of challenges. We describe them below in brief and how we handled them.

4.1 Signature length

For any digital asset, e.g. a digital document, the length of the signature does not affect the asset that is being signed. When applying digital signatures to DNA, we cannot use any arbitrary length for the signature as a very long signature sequence could impact the properties of the DNA molecule. Also, it is not expected to use weak security parameters to shorten the signature length as this might keep the properties of the molecule intact but instead compromise the security of the signature itself. For a digital document, a signature of 384 bytes (say) is trivial. But the same 384 bytes translates to 1536 bases(384 * 8 / 2) in a DNA. If a DNA sample originally contains say 2000 bases(not unusual for a plasmid), the addition of a 1536 nucleotide signature would nearly double the size of the DNA molecule. As a consequence, we were not able to apply identity-based signatures that use bilinear pairings. The signature length would affect the properties of the plasmid. Hence, we decided to utilize Shamir's IBS scheme with a minor modification. The details of our signature scheme are described in section 5.1.

4.2 Signature identification

In a digitally signed document, the original message and the signature can be easily identified and separated since we have delimiters that separate them. In the DNA domain, there exists another problem of embedding the signature inside the original molecule. Because the site of insertion will vary depending on the architecture of the plasmid, delimiters are needed to identify where the signature starts and ends. So we have used an algorithm that identifies subsequences. Any

subsequence of 10 base pairs (substring of length 10) that is not present in the original sequence can be used as a start and end delimiter which will contain the signature. During verification, all subsequences of 10 base pairs will be identified and only those subsequence that occur twice within the entire sequence are the delimiters.

Although the above technique can be useful, to keep things simple we came up with an alternate solution. Instead of the algorithm choosing the delimiters, the tool we developed lets the user input their own delimiters of 10 base pairs. This approach can be beneficial as it lets the biologist design delimiters that are relevant to their specific project. For instance, the delimiters can be designed in such a way as to simplify synthesis/assembly of the DNA. The tool checks if the sequences are permitted i.e. the 10 base pair subsequence does not already exist elsewhere in the plasmid. The sequences that we have used as start and end delimiters are ACGCTTCGCA and GTATCCTATG respectively. These sequences are relatively easy to identify visually, they are unlikely to develop secondary structures and they contain a balanced number of A's C's G's and T's.

4.3 Error tolerance

When any digitally signed message is shared and verification fails, the sender just resends the message again. But in the domain of DNA sharing, since we are primarily shipping samples, this implies resending the sample (sometimes even batches of samples). This will incur a lot of cost. The presence of a signature inside the molecule will ensure that any change in the signed DNA will result is failed verification. But DNA molecules are prone to naturally occurring mutations. Hence after a failed verification, it can be useful to check the location of mutation which caused the verification to fail. If the mutation is in any important feature, the receiver can reorder the sample. If the mutation is in any relatively unimportant part of the DNA, the receiver can choose to proceed to work with it. In order to achieve this error tolerance, we have used error correction codes specifically Reed-Solomon Codes [14]. But we assume that the start and end tags described above do not mutate. Without the start and end tags we cannot identify where the signature is located. The details about error correction codes are described in section 5.4.

4.4 Strong association between physical DNA molecule and its digital representation

We discuss two ways of achieving this association between the physical DNA sample and its digital representation. The first way is to embed a dual signature in the digital representation. This dual signature combines the signed sequence, the description and the identity of the signer and generates a signature on this combined message. This signature is placed in the digital representation of the DNA such as the Genbank file which is shared with the receiver along with the physical signed DNA sample. The receiver will be able to associate the received physical DNA sample and this file easily due to the presence of the dual signature. This ensures that the explanation of the sequences and the sequences in the plasmid are correct and related. Any change is the descriptions without changing the molecule will invalidate this signature. Also, any change in the molecule without updating the descriptions will invalidate the signature. In this approach, the signer has to share two items with the receiver, the signed physical DNA and the associated digital file. The details of this procedure are described in section 5.5.1.

The second way is to embed the features and descriptions of the physical DNA in the physical DNA itself. In this approach, the signer does not need to send the digital file additionally since the descriptions are already embedded in the DNA sequence. The receiver when sequencing the received DNA sample can reconstruct the features and descriptions related to the sample. The descriptions are extracted and converted into ACGT sequence, the plasmid DNA sequence is already present. The signature is generated from the combination of the description sequence and plasmid sequence. This ensures that the description sequence and the plasmid sequence are correct and unaltered. The signature and description sequence are then embedded in the original DNA sequence. We also use error correction codes in conjunction to provide some error tolerance from mutations. The details of this procedure are described in section 5.5.2

Chapter 5

Proposed Solutions

5.1 DNA Signature scheme

We use the identity-based signature scheme proposed by Shamir [11] in 1984. However, we have simplified it to suit the DNA sharing domain. The unique identifier in our case is the ORCID(Open Researcher and Contributor ID). Shamir's IBS is based on the RSA cryptosystem and its security depends on the hardness of integer factorization in the RSA problem.

Setup: The setup is similar to the standard RSA cryptosystem setup. For a given security parameter k, proceed with the following steps -

1. Generate two distinct primes p and q at random with $2^{\frac{k-1}{2}} < p, q < 2^{\frac{k}{2}}$
2. Calculate the modulus n as $n = p \cdot q$
3. Calculate the totient $\phi(n) = (p-1)(q-1)$. Choose the master public key *e* as $1 < e < \phi(n)$, such that *e* is relatively prime to $\phi(n)$.
4. Calculate the master private key, d, as $e^{-1}\ mod\ \phi(n)$ in order to satisfy the congruent relation $d \cdot e \equiv 1\ mod\ \phi(n)$
5. Publish the public parameters $<e, n>$ and the keep the private key d.

In our setup k is 1024 bits.

Key Extraction: The private key, s_{ID} for a user with the identity ID is generated as:

$$s_{ID} = H(ID)^d\ mod\ n$$

where H is a secure hash function. We are using SHA-256 as the hash function.

Signature Generation: For generating the signature for a message $m \in \{0,1\}^*$, generate the signature(σ) as :

$$\sigma = s_{ID}^{H(m)} \ mod \ n = H(ID)^{d \cdot H(m)} \ mod \ n$$

Signature Verification: To verify a signature σ for a message m and user identity ID, check if the following equation holds:

$$\sigma^e \stackrel{?}{=} H(ID)^{H(m)} \ mod \ n$$

We have simplified the existing Shamir's IBS scheme by removing the random. The random will prevent two same messages to have the same signatures. Using the original scheme with the random, the signature length(σ) will be 2048 bits or 1024 base pairs for the security parameter(k) of 1024 bits. Removing the random will make the signature length 1024 bits or 512 base pairs for the same security parameter(k) of 1024 bits. In the realm of DNA sharing, users will be primarily shipping physical samples, not sending digital information over the internet.A signer will not be sharing the same signed DNA molecule with a receiver more than once. Hence in this domain, we can remove the random to get a shorter signature length.

Figure 5.1 depicts a map of the plasmid features of a digitally signed DNA file.

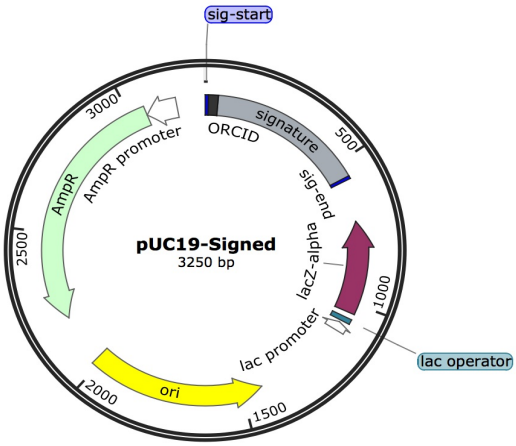

Figure 5.1: Snapgene view of a signed digital DNA file.

An application software called **DNA-Sign** has been developed which users can utilize to generate signatures from a genbank file and also validate a signature on a genbank file. The details of the signature generation and verification procedure are described in Chapter 6.

5.2 Proof of security for DNA signature scheme

Our DNA signature scheme is a simplified version of Shamir's IBS scheme. To show that our scheme is secure we first present the original Shamir's IBS scheme:

Shamir's IBS Scheme

Setup: Same as above.

Key Extraction: Same as above.

Signature Generation: For generating the signature for a message $m \in \{0,1\}^*$:

1. Choose $r \in_R \mathbb{Z}_n^*$.
2. Compute $R = r^e \ mod \ n$.
3. Compute $c = H(R||m) \ mod \ n$.
4. Compute $t = s_{ID} \cdot r^c \ mod \ n$.
5. Output signature $\sigma = (R, t)$

Signature Verification: To verify a signature σ for a message m and user identity ID, check if the following equation holds:

$$t^e \stackrel{?}{=} H(ID) \cdot R^{H(R||m)} \ mod \ n$$

We show that the simplified Shamir's IBS scheme is able to provide the same level of security guarantees as the original scheme.

The signature in the original scheme is a tuple - (R, t). If the modulus chosen is 1024 bits, the signature output will be 2048 bits which is 1024 base pairs. Based on our threat model, Shamir's

IBS scheme is secure if no polynomial-time adversary can forge the signature on a given message. It is readily shown that this is equivalent to the difficulty of breaking the RSA public-key cryptography. Here is how. To forge a signature, the adversary needs to find s_{ID} from the equation $t = s_{ID} \cdot r^c \ mod \ n$. Let, $r^c = w$. Therefore, $s_{ID} = t \cdot w^{-1}$. In order to find any inverse modulo n, one has to know $\phi(n)$, where $\phi(\cdot)$ is the Euler totient function. Calculating $\phi(n)$ from n is equivalent to factoring n into two distinct primes – a known hard problem. Next, to calculate w^{-1}, the random r has to be calculated. If r can be found, then r^c can be found as c is public. $c = H(R||m) \ mod \ n$. R is first part of the signature and m is the message which bears the signature. To find the random r, one has to know $\phi(n)$ or the secret key d, since $R = r^e$, $r = R^d$.

With that brief background, let us now consider the simplified Shamir's IBS scheme. We have simplified the original scheme by removing the random R. In our scheme, the signature $\sigma = s_{ID}^{H(m)}$. Therefore, $s_{ID} = \sigma^y$, where $y = H(m)^{-1}$. Hence to find y, one has to know $\phi(n)$ which is equivalent to the RSA problem. Therefore, no polynomial-time adversary can forge a signature in the simplified scheme.

Note that our scheme will generate the same signature for the same message every time. This is a threat where replay attacks on signatures is of concern. In our domain, the threat of replay attacks is negligible since replaying the signed message implies sending the actual signed DNA to the receiver again. As this is not any digital message which can be generated by packet crafting or similar techniques, the attacker would have to actually synthesize the DNA molecule and send it to the receiver. On the other hand, removing the random will make the signature length 1024 bits or 512 base pairs if the modulus(n) is 1024 bits. Hence, in our domain, although the same original DNA plasmid, originating from the same source, will have the same signature every time, the practical risk is minimal and is outweighed by the benefit of minimizing the signature length so as to decrease the likelihood that the functionality or stability of the plasmid is disrupted.

We are exploring other identity based signature schemes which generate shorter signatures as the signature sequences have to be ordered from gene synthesis companies and this can be financially expensive. The cost of synthesis is about 7 to 9 cents per base pair.

5.3 Sign-Share-Validate Workflow

In our system, there are three players: 1) The signer will develop the DNA signature and sign a sequence. 2) The verifier will use the signature to verify whether the received DNA sequence was sent by the appropriate sender and was unchanged after signing. 3) A Central Authority will provide the signer with a token that is associated with their identity. We assume that the central authority is secure and trusted by all participants in the system.

There are also three steps to the sign-share-validate workflow, summarized in Figure 4. In this example, Alice is developing a new plasmid. She starts in a sequence editor application by combining sequences from different sources. When she has finalized the sequence of the plasmid she wants to assemble in the lab, she uses the signature generating service hosted in a server to create a DNA signature sequence she will add to her design. This DNA sequence is the digital signature. It is generated using the signature algorithm described in section 5.1. Alice provides the digital DNA file to sign, her unique identifier (ORCID), a six-digit plasmid ID, a start and end sequence that will contain the signature sequence. The digital signature is inserted in the plasmid sequence between two conserved sequences used to identify the signature from the rest of the plasmid sequence. Alice will then assemble the signed plasmid by combining DNA fragments from different sources. She will have to order the DNA fragment corresponding to the signature from a gene synthesis company. She describes her plasmid in a paper and refers to it using the six-digit number ID which she used to identify the plasmid in the signature. She did not include the entire plasmid sequence in the online supplement of the article. She sends the plasmids to a few collaborators.

Ellen is interested in using Alice's plasmid. She gets the plasmid from another graduate student who got it from his advisor a few years ago. Ellen has limited confidence in the plasmid because it came in a hand-labeled tube. So, she decides to get it sequenced completely before doing anything with it. She uploads the assembled sequence of the plasmid to the server to verify the plasmid. The signature validation service in the server identifies the signature inserted between the two signature tags. It will identify a block of 32 bp. to the right of the signature start signal to extract

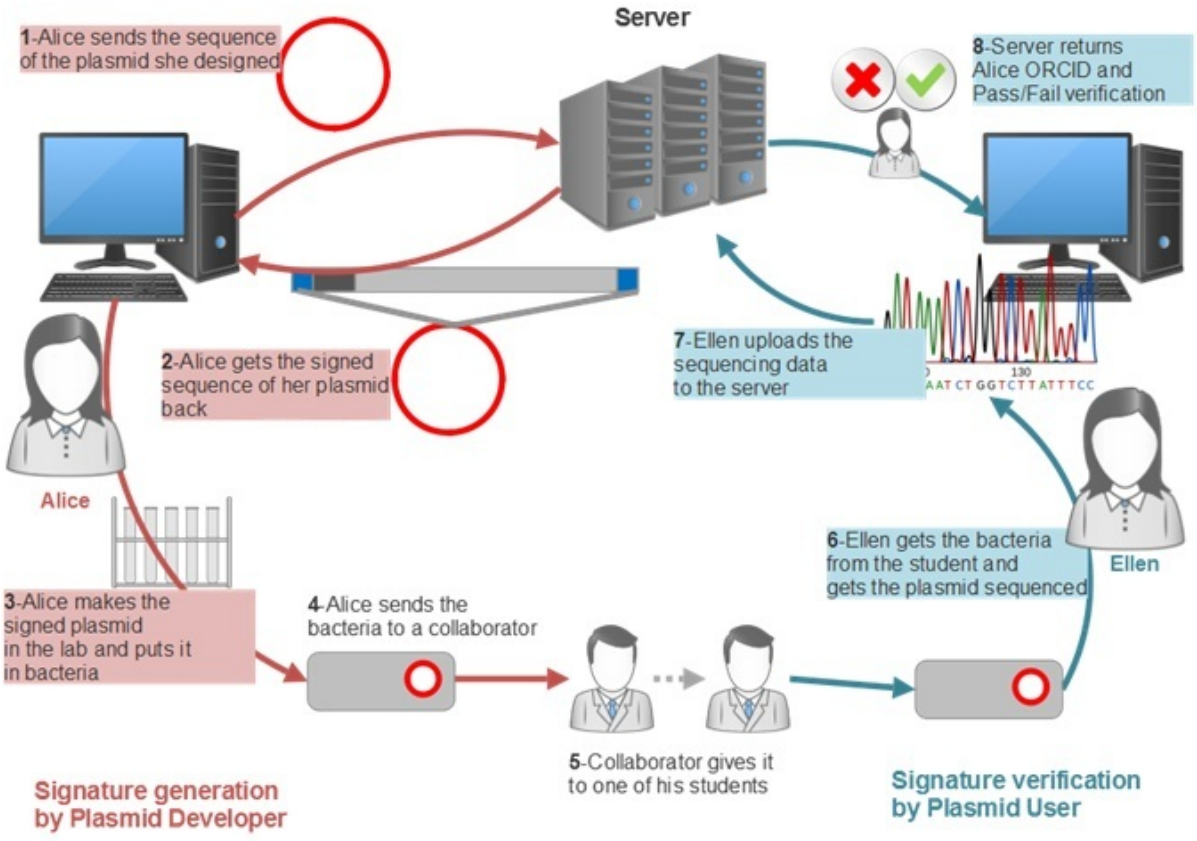

Figure 5.2: Example of a sign-share-validate workflow

the plasmid developer ORCID. Using the ORCID value as identity, the server decrypts the 512 bp signature block. Then validation service will verify the signature as described in section 5.1. If the two values match, then Ellen will know that the plasmid was signed by Alice and that the physical sequence of her plasmid corresponds exactly to Alice's design. She had asked Alice for the plasmid sequence to align with her sequencing data. Unfortunately, Alice had moved on with her life and she no longer had access to the plasmid sequence files. Nonetheless, because she was careful enough to sign her plasmid, Ellen can be assured that the plasmid she intends to use is the one described in the publication.

It is also possible that validation service did not validate the plasmid signature. Several hypotheses could lead to this situation. It is possible that Alice was sloppy and did not manage to assemble the plasmid corresponding to the sequence she had designed. It is also possible that her

advisor handed Ellen a derivative of the plasmid described by Alice. One could also not rule out the possibility of spontaneous mutations or a stupid labeling error. In this situation, Ellen may decide to proceed with the plasmid based on the similarity of the plasmid sequence and the information available in the Methods section of the paper describing the plasmid.

5.4 Error Correction Codes

Limitations of using signed DNA: The presence of a digital signature within a DNA will guarantee that the original sequence, identity sequence and the signature sequence itself has not been tampered with since the signer sent the DNA sample. If any of these change, intentionally or unintentionally, the receiver will not be able to verify the DNA sequence. The sequences inside a DNA are prone to mutation. Mutation is a naturally occurring phenomenon. Whenever there is any mutation within the signed DNA, the receiver will not be able to verify and this is not the signer's fault.

Mutations can be of three types: 1) **Point mutation** - Where one base changes to another base e.g. AAGGAA -> AAGAAA. 2) **Addition** - When a subsequence gets added to the original sequence e.g. AAGGAA -> AAGAGAA. 3) **Deletion** - When a subsequence gets deleted from the original sequence. e.g. AAGGAA -> AAGG. In any of the above scenarios, the verification process will result in failure. In a digital realm, if any message is not verified we can always resend the message. But in the DNA sharing domain, this means that the sample has to be transported again which incurs a lot of cost. Along with the problem of mutation lies the problem of sequencing. When the DNA is processed by the automated DNA sequencer, the output is not always one hundred percent correct. It is dependent on the depth of sequencing.

In order to overcome these limitations, we are using error correction codes along with signatures. The presence of error correction codes will help the receiver to locate and correct some errors in the sequence. In information theory and coding theory, error detection and correction are techniques that enable reliable delivery of digital data over unreliable communication channels. Many communication channels are subject to channel noise, and thus errors may be introduced

during transmission from the source to a receiver. Error detection techniques allow such errors to be detected, and error correction is used to reconstruct the original, error-free data. In error correction, redundant data, or parity data, is added to a message, such that it can be recovered by a receiver even with a number of errors (up to the capability of the code being used). Error-correcting codes are frequently used in lower-layer communication, as well as for reliable storage in media such as CDs, DVDs, and hard disks. It is possible to use the same techniques to provide a reliable reconstruction of sequences provided a small number of changes have occurred. Our application of error correction codes to DNA can also be used to ensure the integrity of digital information stored in DNA molecules.

We have used Reed-Solomon codes [14] for error detection and correction of DNA sequences. As of now we have utilized it to correct point mutations only. Reed-Solomon codes are block-based error correcting codes with a wide range of applications in digital communications and storage. It is used extensively in storage devices e.g. CD/DVD, barcodes, QR codes, wireless communication, satellite communication (voyager space probe used RS codes) etc. We will not explain the construction and working principle of a Reed-Solomon code here. An interested reader can refer to [15, 16] for details.

The most common convention of a Reed-Solomon code is (255,223,32) in which 223 is the number of data symbols, 32 is the parity symbols and 255 is the total number of symbols that can be processed at a time or block size. Using this convention the total number of errors that can be corrected anywhere in the 255 symbols is 32/2 = 16. This convention uses 8-bit symbols. Since the symbols are 8-bits, the block size is $2^8 - 1 = 255$ and with respect to a programming language, each symbol is treated as a byte. So the Reed-Solomon code of (255,223,32) can be simply put as 255 bytes block size, 223 data bytes, and 32 parity bytes. The total number of errors that can be corrected is 16 bytes. The parameters are generated from Galois field GF(257).

Here also we face an obstacle while applying this to DNA domain. Note that a plasmid contains 2500 to 20,000 base pairs. If we treat each base as a character or byte, we will not be able to process the entire sequence in a single block. We have to make blocks of 255 bases. This implies that in a

block of 255 bases, 223 bases are the actual sequence and 32 bases are the parity sequence for that block. As we mentioned earlier, we do not have the privilege of using any delimiters in a DNA as we had in a digital message. Therefore, the parity sequences of every block cannot be identified. Also, let us consider the following scenario - a user wants to correct 5 bases in a plasmid which contains a total of 800 bases. By convention, this 800 bases cannot be processed at a time and will be processed in four blocks (255 bases in each). We cannot be certain about the distribution of errors in the four blocks i.e. there is no guarantee where the 5 errors might be. It can be all in one block or other possible ways but certainly not uniform (1.25 in each block). So assuming the worst case, we need to correct 5 errors in each of the blocks. Therefore the number of parity bytes for total 800 bases is now $10 \cdot 4 = 40$. Whereas, if we could process the entire 800 sequence at once, we would have only $5 \cdot 2 = 10$ parity bytes.

In order to adapt to this scenario, we used 16-bit symbols or shorts. Now the block size is $2^{16} - 1 = 65535$. The sequence characters which were bytes are now shorts. The parameters are generated using Galois field GF(65537). This gives us the flexibility to process entire plasmid sequence at once. The user when signing provides the number of errors that they would like to correct. The original plasmid sequence and the generated sequence are passed to the Reed-Solomon encoder. The Reed-Solomon encoder generates $2 \cdot k$ shorts for k error tolerance. The $2k$ shorts are then converted to sequences. Each parity short is converted to an 8 base sequence (16/2). Previously the final signature sequence consisted of - $< start >< ORCID + Plasmid\_ID + Signature >< end >$ where start and end were 10 base pairs each, ORCID was 32 base pairs, Plasmid_ID was 12 base pairs and Signature was 512 base pairs. Now the parity sequences are inserted between the signature sequence and end sequence. Updated signature sequence - $< start >< ORCID + Plasmid\_ID + Signature + Parity >< end >$. During the correction phase, the parity sequence is retrieved using the start and end sequences and the length of other three parts which is already known. The number of errors that can be corrected can be determined by the length of the parity sequence. Since each parity short is 8 bases, 16

bases are two shorts and two shorts can correct one error, hence the number of errors that can be corrected are - (parity sequence length) / 16.

Using the error correction code, the verifier can correct some number of errors (limit is set by signer). Upon correcting the sequence, the verification is invoked again on the corrected sequence. The position of the errors and the corrected value are conveyed to the verifier. The verifier can then decide if the errors are in any valuable feature or not. If a valuable feature has been corrupted, the verifier can ask for a new shipment, else if the error was in a non-valuable area in the plasmid, the verifier can proceed to work with it.

The sign-share-verify workflow will be updated accordingly as follows. Ellen will upload the digital DNA file to the server which she obtained after sequencing the plasmid shared by Alice. The validation service will try to validate the sequence. If this validation results in failure, the error correction part will be invoked will try to correct the sequence depending on how many errors Alice chose to be tolerated during signing. If no corrections can be made (may be due to more mutations in the sample than Alice predicted) Ellen will be notified with an alert. If corrections can be made, the verification will start again on the corrected sequence. Upon successful verification on the corrected sequence, Ellen will be notified about the errors(mutations) that occurred in the sample she received.

5.5 Associating the DNA sample with its digital representation

The sequences within a physical DNA molecule can be obtained digitally using an automated DNA sequencer. The sequencer outputs a digital file - .fasta which contains the ACGT sequence of the sample. The fasta file can be converted into annotated files like .dna , .gb etc which contains the genetic features along with its descriptions. Gene-manipulating software like Snapgene aid in converting these subsequences into maps of plasmid features. Users can add feature descriptions manually or search within the Snapgene database for any matching features. When the physical DNA sample is shared, the receiver will also sequence the sample and obtain its sequence. Now the receiver will be able to view only the features that Snapgene can automatically interpret from

its database. The extra features that the sender manually annotated will not be available to the receiver. Hence the sender also needs to share the annotated file with the receiver such that the additional annotations are available to the receiver. Hence, it is essential to tie the digital file with the physical sample in some way such that the receiver can be certain that the annotations belong to the particular physical sample. We discuss two possible ways of tying the digital file with the physical sample.

5.5.1 Dual signature in the digital DNA file

In this method, the sender needs to share the digital genbank file along with the physical DNA sample. In order to associate the digital DNA file which contains the additional descriptions(let us call this F_{sent}) with the digital DNA file that receiver generates after sequencing the sample(let us call this F_{gen}), we associate them together with a combined signature. The association between the digital file and the physical sample is created in the following way:

Create association

1. Signer provides the digital DNA file containing the appropriate sequence and descriptions. Extract the sequence and the descriptions. Only the sequence is used for signature creation as described in section 5.1. Let this sequence be m_{seq} and the descriptions be m_{desc}.

2. Generate signature on m_{seq} as before and place this within the original sequence. Let this final signed sequence be m_{sig}.

3. Combine m_{desc} and m_{sig} by calculating the following:

$$m_{comb} = H(H(m_{sig})||H(m_{desc}))$$

   where H is a secure hash function e.g. SHA-256 and || is concatenation operation.

4. Create signature for this m_{comb} using the same procedure -

$$\sigma' = s_{ID}^{m_{comb}} \; mod \; n = H(ID)^{d \cdot m_{comb}} \; mod \; n$$

5. Add σ' to the genbank file with a keyword "ASSOC".

6. Share the file with the recipient.

Validate association

1. Recipient obtains F_{sent} and generates F_{gen} from the received sample. The tool takes both files as input.

2. Extract σ', m_{desc} from F_{sent} and m_{sig} from F_{gen}. The ID is extracted from m_{sig}.

3. Calculate m_{comb} as:

$$m_{comb} = H(H(m_{sig})||H(m_{desc}))$$

4. Check if the following equation holds:

$$(\sigma')^e \stackrel{?}{=} H(ID)^{m_{comb}} \; mod \; n$$

Using this combined signature, the recipient can validate that the description file was sent by the authentic sender, the manually added descriptions have not been changed and these descriptions belong to the same DNA sample that was shared. The sign-share-verify workflow will work as follows. Ellen will upload the digital DNA file she generated after sequencing the sample shared by Alice and also the digital DNA file that Alice shared (which contains the additional descriptions). The server will match the combined signature and Ellen will be notified about the association between the two files.

The limitation of this method is that any user who wants to share a DNA sample must also share the approriate digital file. But often there are many researchers who are working together

on a sample and each of them makes changes to the physical sample independently. These subtle changes are not always documented in the digital file. Hence the digital file might not be a correct representative of the physical sample. The advantage is that the physical sample is unaltered and hence its properties will remain intact.

5.5.2 Self Documenting DNA

In this method, the sender needs to share only the physical DNA sample. The annotations are embedded within the physical sample itself. The receiver can sequence the physical DNA as before and from the fasta file, the genbank file(all descriptions and sequence) can be generated. However, the disadvantage is that since we are embedding some additional sequence within the original plasmid, its size will increase and hence it might not retain all of its original properties. But if the sample can retain its original properties this method might be a better option than the previous. Experiments are required to check the validity and applicability of this method. If the experimental validations are successful this will be integrated with the signature scheme to produce a physical DNA sample which contains proof of origin, sequence integrity validation and which contains the description about itself within its sequence. As of now this is a separate application, later if the experiments succeed this will be integrated with the signature application. In that case the workflow will be slightly modified than this. In this workflow, given a digital genbank file, the descriptions are extracted, compressed using any compression technique, converted to ACGT sequence and added to the existing sequence of the DNA. The workflow of this method is as follows:

1. User provides the digital DNA file containing the appropriate sequence and descriptions. The user also provides three conserved tag sequences. These tag sequences are not present within the original plasmid sequence i.e. they are unique. The user then provides the location where to put the description sequence and also the number of errors/mutations that will be tolerated.

2. The sequence and the descriptions are extracted from the provided digital(genbank) file. Only the sequence is used for signature creation as described in section 5.1. The descriptions are utilized here, the descriptions are compressed using a compression algorithm and are converted to ACGT sequence. This description sequence is placed between tag#1 and tag#2.

3. The original plasmid sequence is then combined with this description sequence by placing it within the location specified by the user. We cannot place this in any arbitrary location as there might be feature existing there and that will change the properties of the plasmid. If the user chooses to insert in the begining of the original sequence the output is - $< tag\#1 >< descriptions >< tag\#2 >< original >$. If the user chooses to insert in the end of the original sequence, the output is - $< original >< tag\#1 >< descriptions >< tag\#2 >$. Otherwise, if the user chooses a location within the original sequence, the output is $< originalpart1 >< tag\#1 >< descriptions >< tag\#2 >< originalpart2 >$.

4. A checksum is generated on this intermediate sequence.Any existing checksum generating algorithm can be used. We have used CRC32 checksum [20]. As we are using CRC32 checksum, the checksum length is always 32 bits or 16 base pairs. This checksum sequence is placed within tag#2 and tag#3 by appending the checksum and tag#3 after tag#2. The error correction code is then generated from this sequence(i.e. the original, description and checksum sequence) and appended after this checksum, also within tag#2 and tag#3. We call this subsequence - $< tag\#1 >< descriptions >< tag\#2 >< CHECKSUM >< ECC >< tag\#3 >$ as "annotation sequence" . Similar to the signature sequence, we use error correction code(ECC) to tolerare some amount of mutation that can occur within the DNA. Without this error correction code, mutations will lead to corrupted original sequence and description sequence which will result in a incorrect digital file at the receiver's end.

5. The final combined sequence output is - $< originalpart1 >< tag\#1 >< descriptions >< tag\#2 >< CHECKSUM >< ECC >< tag\#3 >< originalpart2 >$. Depending on the users choice of insertion, either $< originalpart1 >$ or $< originalpart2 >$ can be empty.

The final combined sequence, i.e. the original and the annotation sequence is written to a text file or .fasta file. This file contains only ACGT sequences. The following figure depicts an example .fasta file.

```
>Sequence_1 assembly1
CCCTAAACCCTAAACCCTAAACCCTAAACCTCTGAATCCTTAATCCCTAAATCCCTAAAT
CTTTAAATCCTACATCCATGAATCCCTAAATACCTAATTCCCTAAACCCGAAACCGGTTT
CTCTGGTTGAAAATCATTGTGTATATAATGATAATTTTATCGTTTTTATGTAATTGCTTA
TTGTTGTGTGTAGATTTTTTAAAAATATCATTTGAGGTCAATACAAATCCTATTTCTTGT
GGTTTTCTTTCCTTCACTTAGCTATGGATGGTTTATCTTCATTTGTTATATTGGATACAA
GCTTTGCTACGATCTACATTTGGGAATGTGAGTCTCTTATTGTAACCTTAGGGTTGGTTT
ATCTCAAGAATCTTATTAATTGTTTGGACTGTTTATGTTTGGACATTTATTGTCATTCTT
>Sequence_2
CCCTAAACCCTAAACCCTAAACCCTAAACCTCTGAATCCTTAATCCCTAAATCCCTAAAT
CTTTAAATCCTACATCCATGAATCCCTAAATACCTAATTCCCTAAACCCGAAACCGGTTT
CTCTGGTTGAAAATCATTGTGTATATAATGATAATTTTATCGTTTTTATGTAATTGCTTA
TTGTTGTGTGTAGATTTTTTAAAAATATCATTTGAGGTCAATACAAATCCTATTTCTTGT
GGTTTTCTTTCCTTCACTTAGCTATGGATGGTTTATCTTCATTTGTTATATTGGATACAA
GCTTTGCTACGATCTACATTTGGGAATGTGAGTCTCTTATTGTAACCTTAGGGTTGGTTT
ATCTCAAGAATCTTATTAATTGTTTGGACTGTTTATGTTTGGACATTTATTGTCATTCTT
```

Figure 5.3: A sample .fasta file

6. The combined sequence is then outsourced to a gene synthesis company who will create a DNA fragment containing the combined sequence which can be shared. Otherwise, only the annotation sequence can be outsourced and the DNA fragment containing the annotation sequence will be created. The user then combines the original plasmid sequence and the annotation sequence in the lab and then it is ready to be shared.

7. The receiver upon receiving the shared plasmid, sequences it using an automated DNA sequencer. The generated .fasta file will have the combined sequence. The receiver then provides the three tag sequences that the sender provides. Using this information, the digital DNA file is created along with the descriptions directly from this fasta file. The receiver does not need to use any gene manipulation tools to interpret the features and descriptions.

8. The description sequence is present within tag#1 and tag#2, the checksum and error correction sequence is present within tag#2 and tag#3. First, the checksum, error correction, descriptions and original sequence are extracted and isolated. Then the original sequence and description sequence are combined and a CRC32 checksum is generated. It is then validated against the extracted checksum. If these two sequences are equal, this implies the sequences have not been altered i.e. no mutations occurred. In this case, the genbank file is generated after this step. If this checksum validation fails, the error correction sequence comes into action and tries to correct any mutations that may have occurred, provided the number of error is within the tolerance limit set by the sender. If the number of errors are within the tolerance limit, the genbank file is generated with the corrected information and the user is provided with the position and content of the error. If the number of errors are more than the tolerance limit, the user is notified.

In this approach, the assumption is tag#2 and tag#3 is not corrupt or mutated. Since the corruption of any or both of these tags will result in irrecoverability of the error correction sequence. Consequently, due to loss of tag#2, the description sequence end cannot be located and hence the digital genbank file cannot be reconstructed. As a possible solution to this scenario, the use of string silimarity metrics can be used to make a guess where the mutated tags might be located. There are many techniques which address string silimarity e.g. Jaccard Similarity, Dice's Coefficient, Levenshtein distance, Jaro-Winkler edit distance etc. that can be used to solve this problem.

Chapter 6

Signature generation and verification procedure

The tool we have developed allows a user to generate and validate signatures. The parameters i.e. e, d, N are fixed in the prototype where the modulus N is 1024 bits.

Signature generation: The user provides the following inputs for signature generation -

1. The genbank (.gb) file.
2. ORCID - a 16 digit number in xxxx-xxxx-xxxx-xxxx format.
3. Plasmid ID - a 6 digit number.
4. Location of signature placement.
5. Number of errors to be tolerated.

All the necessary input checks e.g. the file has extension .gb, ORCID format is correct, ORCID is integers etc. are done. The signature generation procedure begins by splitting the genbank file by the keyword $ORIGIN$. Refer to figure 2.2. After the keyword $ORIGIN$ is the actual sequence and before it are the descriptions. The sequence is the message to sign and the descriptions are kept for verifying if the user provided location is colliding with an existing feature. Let us assume the sequence to sign is **SEQUENCE** and there exists a feature from location 1 to 3 which corresponds to **SEQ**. Next,the location of signature placement is checked. If the location collides with a feature, the user is alerted to change the location. For our example, if the user had provided 2, the tool will alert the user that there is already a feature **SEQ** there and ask for a new location. If the user chooses 4 which is after the letter **Q**, it will be allowed. Next, the OCRID and Plasmid ID are converted to ACGT sequence by the following conversion method - [0 - AC, 1 - AG, 2 - AT, 3 - CA, 4 - CG, 5 - CT, 6 - GA, 7 - GC, 8 - GT, 9 - TA]. The reason for choosing this conversion type is that if any ORCID or plasmid ID has repetitions e.g. 0000-0001-4578-9987, the converted sequence will not have a long run of a single base. If we used 0 - AA, the example ORCID

would have AAAAAAAAAAAAA in the beginning, and long runs of a single nucleotide can result in errors during sequencing. In the chosen conversion method the ORCID would start with ACACACACACACAC. Let the converted ORCID and Plasmid ID sequence be **ORCID** and **PID**. The signature is generated according to the scheme described in section 5.1. The signature bits are then converted to ACGT sequence. Let this signature sequence be **SIGN**. Also, recall that the start and end tag where this signature is to be placed is predefined. Let this start tag be **START** and end tag be **END**. The signature sequence is concatenated with ORCID and plasmid ID and then placed between the start and end - **START ORCID PID SIGN END**. This entire string is placed at the position specified by the user. As we chose 4 in our example, the total sequence looks like - **SEQ START ORCID PID SIGN END UENCE**. Now this string is passed into the error correction encoder. According to the number of tolerable errors specified by the user, the parity bits are generated. The parity bits increase with the number of errors to be tolerated. These parity bits are then converted to ACGT sequence. Let this be **ECC**. When the encoder output is generated, the string looks like - **SEQ START ORCID PID SIGN END UENCE ECC**. Next, the **ECC** is separated and this is placed before the signature and end tag. So the final output string is - **SEQ START ORCID PID SIGN ECC END UENCE** . Note that the error correction code is generated after generating the signature sequence and combining with original sequence. Hence any error in that string can be corrected provided it is within the tolerable limit. For our example, if we put 2 as our error tolerance limit, then any 2 errors within the string **SEQ START ORCID PID SIGN END UENCE ECC** can be tolerated. For example if there is 1 error in **SEQ** and 1 error in **SIGN**, or 2 errors in **SIGN**, or 1 error in **SIGN** and 1 error in **ECC**, it can be corrected. But if there are more than two errors it cannot be tolerated. The final output string - **SEQ START ORCID PID SIGN ECC END UENCE** is written into another genbank file. The descriptions are updated i.e. the locations of the signature, start, end, ecc are added and if there are features after the signature placement locations they are updated. The output genbank file is shared with the recipient.

Signature verification: The user provides the following inputs for signature verification -

1. The shared genbank (.gb) file.

2. The fasta (`.fasta`) file which the receiver obtained after sequencing the shared, signed DNA.

The sequence in the fasta file might not be the in the same order as the receiver sent it. That is, after sequencing the shared DNA, the fasta file may look like - **ORCID PID SIGN ECC END UENCE SEQ START** which is a cyclic permutation of the final sequence the receiver obtained after signature generation. The genbank file contains the correct order. The tool aligns the genbank sequence and the fasta sequence. If there is any mutation in the shared DNA the fasta file will have some errors but most of it will be aligned correctly. If there are no mutations the file will be aligned perfectly. For now let us assume there is no mutation and hence the fasta sequence and the genbank sequence will be aligned perfectly - **SEQ START ORCID PID SIGN ECC END UENCE**. The tool looks for start and end tags which we had predefined. After obtaining the start tag, 32 bases are counted, this is the ORCID sequence, next 12 bases are counted, this is the plasmid ID sequence, then 512 bases are counted, this is the signature sequence. Next the substring after this signature sequence to the end tag is retrieved, this is the error correction sequence. Finally, the portion before start tag and the portion after end tag is concatenated to reconstruct the message for signature verification. So as of now we have retrieved **SEQUENCE**, **START**, **ORCID**, **PID**, **SIGN**, **ECC** and **END**. The **SEQUENCE**, **ORCID** and **SIGN** is used for signature verification according to the scheme described in section 5.1. If there is no mutation, the signature verification will succeed and the user is alerted for successful verification. If there is any mutation ,the verification will fail. In this case the extracted parts are used to construct the string - **SEQ START ORCID PID SIGN END UENCE ECC**. Recall that this was the output of the error correction encoder. If the error is within the tolerable limit, it will be corrected. If the error is more than the tolerable limit, the user is alerted that the verification and the error correction both failed. If the error is corrected, we again use the counting method to retrieve the corrected parts - **SEQUENCE**, **START**, **ORCID**,

PID, **SIGN**, **ECC** and **END**. The verification is invoked on the corrected **SEQUENCE**, **ORCID** and **SIGN**. If the verification succeeds the second time the user is notified about success. Also, the corrected parts and the previously extracted parts (before first verification) are compared to display where the error was. If the verification fails on the corrected parts, the user is notified about failure after correction and the corrected errors are displayed.

It is to be noted that although the corrected errors are displayed to the user, the actual content of the DNA did not change, only the fasta representation changed. The physical DNA still contains the error i.e. if the sample is sequenced again, the freshly obtained fasta file will again be erroneous. Hence, the error correction works more like error detection. The user gets the correction information, and if he thinks the errors are not in any important part of the DNA, the user can choose to work with the shared sample. If he thinks the errors are in an important part of the DNA, the sample can be re-ordered from the sender.

Chapter 7

Results

To validate the ability to verify digital signatures from sequencing data and ensure that digital signatures do not interfere with the function of plasmids, a series of experiments were conducted in three different phases.

For Phase I and II, two plasmids were designed for assembly by the Gibson cloning method. One of these was the commonly used commercial vector pUC19. The other was a minimal expression vector consisting only of two antibiotic resistance genes and an origin of replication. For each of these, the sequences, including the signatures, were ordered in four separate parts from one of two DNA synthesis companies, TWIST or Integrated DNA Technologies. TWIST offers DNA synthesis at a more affordable rate ($0.07/bp), but they were not able to synthesize all of the sequence fragments needed. The total cost of DNA synthesis for the two plasmids was $397.88 and $395.16 with the signature accounting for $48.16 of the total cost in each case.

Once the four building blocks for each plasmid were received, they were put together by Gibson assembly and transformed into Escherichia coli cells. The cells were plated on media containing antibiotics. Three colonies from each plate were grown up in liquid cultures containing antibiotics, the DNA was extracted, and the expected structure of the plasmids was confirmed by restriction enzyme digests. The ability of the cells to grow in liquid and solid media containing antibiotics indicates that the signature did not interfere with the origin of replication or the antibiotic resistance genes for either plasmid. The entire process from receiving the plasmid building blocks to confirming the plasmid structures was accomplished in 2 weeks. For one strain transformed with each plasmid, DNA was extracted and sequenced by Sanger Sequencing, and the resulting reads were manually assembled into a single Fasta file for each plasmid. The Fasta files were verified with the digital signature software.

In Phase III, the potential impact of digital signatures on sequence function was further tested using a different strategy. Sequences were designed for a minimal expression construct for the re-

porter gene Lac-Z, which, in the presence of the chemicals Isopropyl β-D-1-thiogalactopyranoside (IPTG) and 5-bromo-4-chloro-3-indolyl-β-D-galactopyranoside (X-gal), results in the production of a blue precipitate. A plasmid including the Lac-Z expression construct was ordered from TWIST with and without the addition of a digital signature. The sequences were synthesized by TWIST within one of their predefined vector backbones. The cost of the sequences was $76.82 without the signature and $131.72 with the signature.

E. coli transformed with the plasmid with the signature "SIGNED" and the plasmid without the signature "CONTROL" were grown on media with and without antibiotics, IPTG, and X-gal. Both plasmids resulted in equivalent numbers of blue colonies suggesting that the performance of the origin of replication, the Lac-Z construct, and the antibiotic resistance marker were all unaffected by the presence of the digital signature. The "SIGNED" plasmid was also extracted from cells, sequenced by Sanger Sequencing, the sequence was manually assembled into a single Fasta file from the reads, and the fasta file was verified using the digital signature software. This entire process took approximately one week. Figure 7.1 shows that the addition of signature does not change the behavior or function of the plasmid. The two samples on the left are the signed plasmids and the two samples on the right are the control plasmids.

The next step is to order plasmids with mutation in different parts e.g. mutation in signature, mutation in ORCID, mutation in the original sequence and then proceed with the same test cases and check if the verification fails at first and if the error correction code can help to detect mutations and provide information to the user about the mutations.

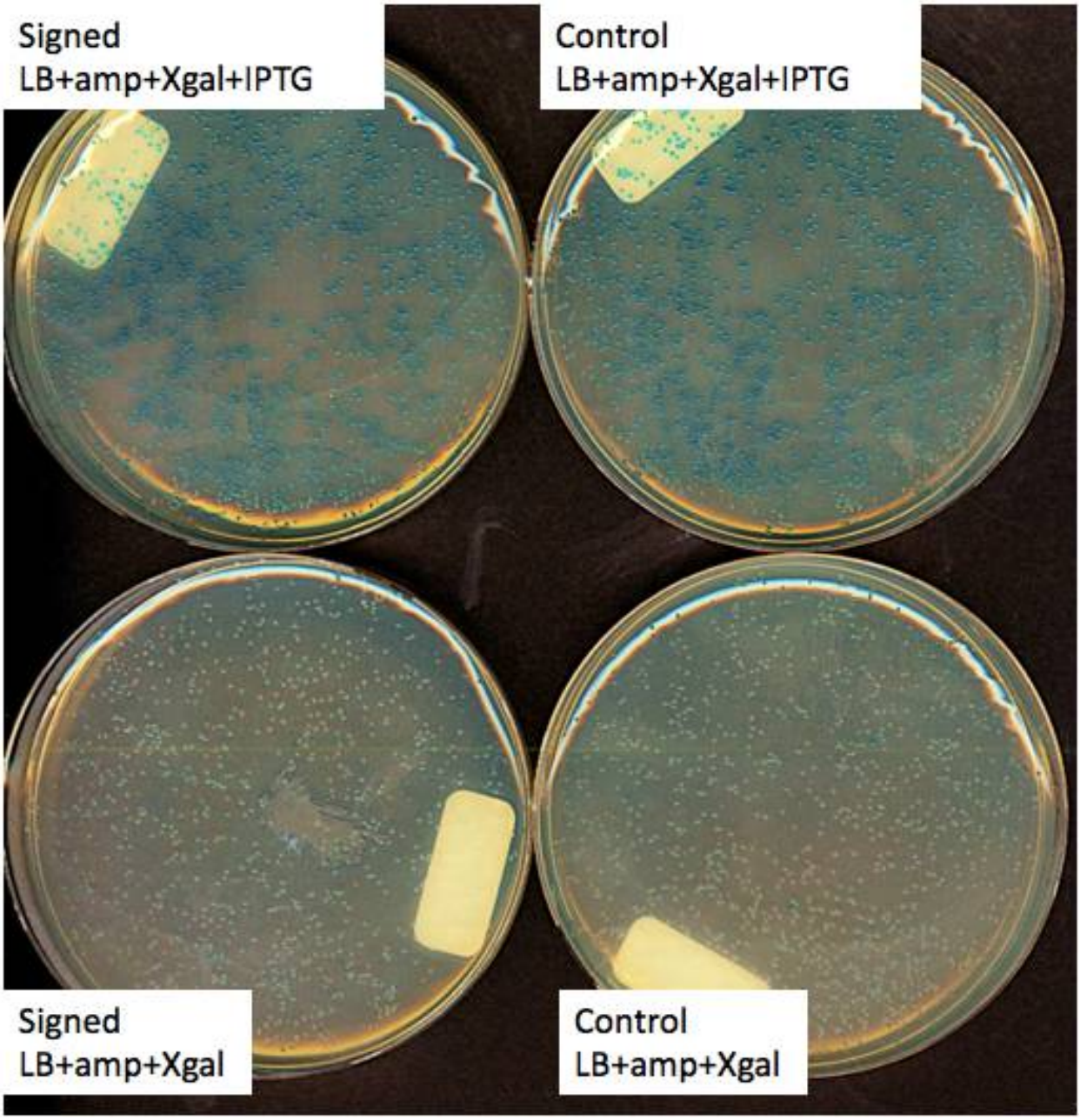

Figure 7.1: Comparing expression of a reporter construct with and without a digital signature.

Chapter 8

Improvements

In this chapter, we discuss some of the improvements that we have made to the work described above. One of the improvements include overcoming one assumption that the identifying tags i.e. the start and end tag which contains the signature or in the self-documenting plasmid case the annotations, do not mutate. The second improvement is using modern elliptic curve and pairing-based cryptographic schemes to generate even shorter signatures with stronger security.

8.1 Allowing mutations in identifying tags

In chapter 4, section 4.2, we defined two identifying tags which contains the signature. The start tag was chosen as ACGCTTCGCA and end tag as GTATCCTATG. Also, when we described error correction code to tolerate mutations within the DNA, we assumed that the start and end tag do not mutate. Otherwise, it will not be possible to locate the signature and consequently, it will not be possible to locate the error correction code itself. Without the ability to locate the error correction code, we cannot invoke error correction in the first place. Let us look at the example provided in chapter 6 - after signature generating is complete the DNA is of the form - **SEQ START ORCID PID SIGN ECC END UENCE**. This form is synthesized and sent to the receiver. The receiver while validating looks for the **start** and **end** tag to extract the information between them. If there is any mutation within the tags itself, for example, **SEQ SMART ORCID PID SIGN ECC END UENCE** (the T in the start has mutated to M) this first step will fail and the user will come up with an alert message that the tags cannot be located.

In order to overcome this limitation, we have used partial matching techniques such that the start and end tag can be located approximately. Since the start and end tags are fixed, we know what we are searching for in the DNA molecule. As per the example, we are looking for the keyword **START** within the mutated string **SEQ SMART ORCID PID SIGN ECC END UENCE**. Although there is no concrete match, we can use the partial matching techniques to output the

closest match to **START**, which in this example is **SMART**. The approximate matching technique breaks the entire string in which we are looking for the result into substring of length input string. Each of the broken substring in the larger string is assigned a score based on how similar it is to the input string. The match is inferred using the highest score. Now in the real DNA, we are looking for sequences of A, C, G, and T. So there might be a case that there are multiple close matches which means there are multiple starts. In those cases, we use the end tags to narrow our results. The following steps describe how the approximate matching technique works. There can be a total of four scenarios -

1. **Case 1: No mutation in both start and end tags.** - In this case, we can locate the extract locations of the tags and hence approximate matching techniques are not needed. There can be mutations in any other place which will be handled by the error correction code.

2. **Case 2: Mutation in START tag only.** - In this case, only the start tag has mutated, the end tag is found directly. The tool looks for the closest match to **START**. If there is a single match with the highest score then we can be pretty certain that the start tag has been located correctly. However, in the substring of A, C, G and T, there can be multiple matches with close scores i.e. there is no single stand out high score. In that case, we can use the end tag for further elimination of choices. We already know that the content within the start tag and the end tag is more than 556 base pairs (Signature is 512, ORCID is 32 and Plasmid ID is 12). Hence we choose only those start tags which are of distance 556 base pairs/characters or more apart from the end tag. The logic is set to 556 or more because the length of the error correction can be 0 if the user chooses no error correction.

3. **Case 3: Mutation in END tag only.** - In this case, only the end tag has mutated, the start tag is found directly. The tool looks for the closest match to **END**. If there is a single match with a highest score then we can be pretty certain that the end tag has been located correctly. For multiple matches with close scores, we use the same logic as described above - the distance between the start and end tag is more than or equal to 556 base pairs.

4. **Case 4: Mutation in both START and END tags.** - In this case, we try to locate the closest match for both tags. If there is a single match with a highest score for both of them then we can be pretty certain that we have located them both correctly. Also, we invoke the criteria of length more than or equal to 556 between them for more certainty. In case of multiple start and end tags, we check the length criteria for each start and end tag pair possible from the obtained results.

Various techniques exist to handle matching of similar strings. Most of the methods measure, the distance between strings, using a distance equation (similar to euclidean distance) to do so. One of the poineering works in this field is the Levenshtein distance. The Levenshtein distance between two words is the minimum number of single-character edits (insertions, deletions or substitutions) required to change one word into the other. Levenshtein distance is generally considered to be very accurate, but can become computationally intensive with very long strings. Similar to Levenshtein, Damerau-Levenshtein distance with transposition (also sometimes calls unrestricted Damerau-Levenshtein distance) is the minimum number of operations needed to transform one string into the other, where an operation is defined as an insertion, deletion, or substitution of a single character, or a transposition of two adjacent characters. The Optimal String Alignment variant of Damerau-Levenshtein (sometimes called the restricted edit distance) computes the number of edit operations needed to make the strings equal under the condition that no substring is edited more than once, whereas the true Damerau-Levenshtein presents no such restriction. The difference from the algorithm for Levenshtein distance is the addition of one recurrence for the transposition operations. Jaro-Winkler is a string edit distance that was developed in the area of record linkage (duplicate detection). The Jaro-Winkler distance metric is designed and best suited for short strings such as person names, and to detect typos. Jaro-Winkler computes the similarity between 2 strings, and the returned value lies in the interval [0.0, 1.0]. It is (roughly) a variation of Damerau-Levenshtein, where the substitution of 2 close characters is considered less important then the substitution of 2 characters that a far from each other. The distance is computed as - $1 - (Jaro - Winkler\ similarity)$. A few algorithms work by converting strings into sets of

n-grams (sequences of n characters, also sometimes called k-shingles). The similarity or distance between the strings is then the similarity or distance between the sets. Jaccard index is one such method which works on n-grams.

We used all of the five algorithms mentioned above for the approximate start and end tag matching. One of the reasons for using all of the above was we wanted to find out which would be most suited to the DNA domain. Also we wanted to measure the accuracy and efficiency of these well known approaches. For testing, the fasta file is taken as input and the start and end tag within the fasta file are manually changed. Then we search for the location of the defined start and end tags within the mutated fasta file. The cases considered were single substitution, single deletion, single insertion, double substitution, double deletion, double insertion, triple substitution, triple deletion, and triple insertion.

The results for each algorithm are summarized on a case by case basis in Figure 8.1. The raw results are presented in table 8.1. As can be seen from the Figures the Jaro algorithm was fairly inaccurate with an average accuracy of only 35.12 %. The Jaccard algorithm fared much better but was still imperfect with an average accuracy of only 95.18 %. All of the three Levenshtein variants were perfectly accurate in their assessment. These results indicate that if accuracy was the chief concern, either of the three Levenshtein variants would be ideal choices.

Table 8.1: Correct matches out of total reviewed strings by algorithm for each case.

	Jaro - Winkler	Jaccard index	Levenshtein	Damerau-Levenshtein	Optimal Alignment
1 Substitution	10/30	30/30	30/30	30/30	30/30
1 Deletion	6/9	9/9	9/9	9/9	9/9
1 Insertion	28/34	34/34	34/34	34/34	34/34
2 Substitution	50/435	408/435	435/435	435/435	435/435
2 Deletions	19/44	41/44	44/44	44/44	44/44
2 Insertions	212/665	665/665	665/665	665/665	665/665
3 Substitution	118/3675	3216/3675	3675/3675	3675/3675	3675/3675
3 Deletions	42/123	101/123	123/123	123/123	123/123
3 Insertions	900/9129	9129/9129	9129/9129	9129/9129	9129/9129
Average Accuracy	35.12%	95.18%	100.00%	100.00%	100.00%

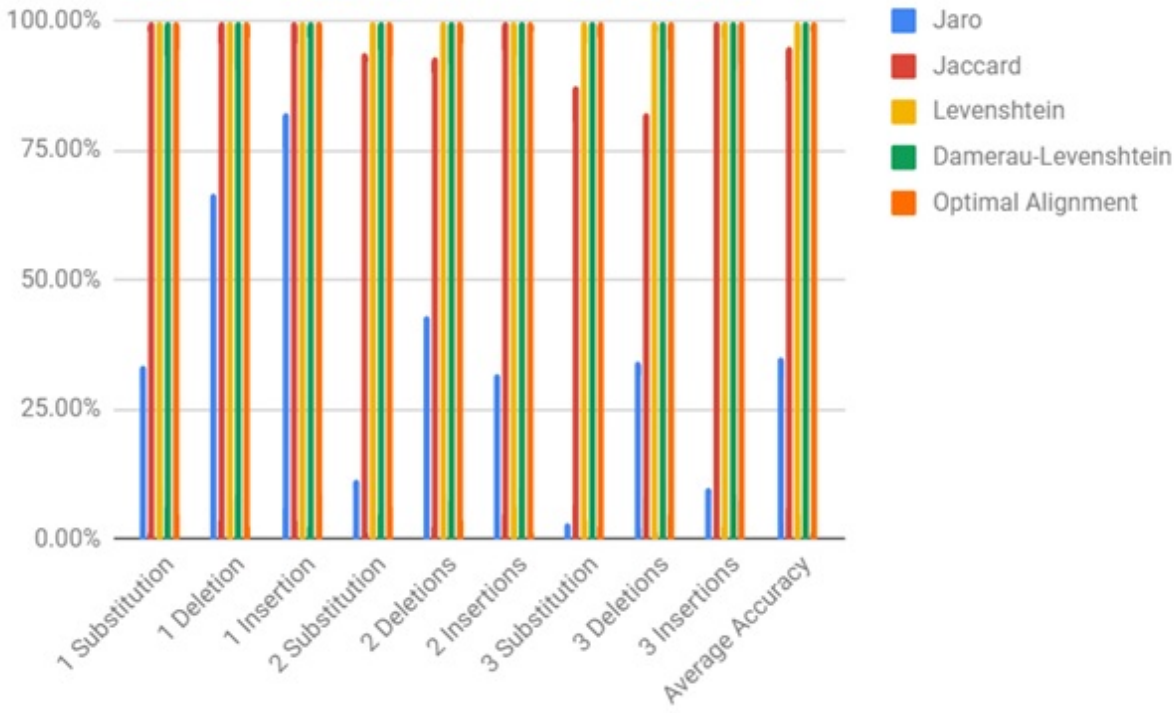

Figure 8.1: Accuracy of algorithms per case as a percentage.

Another important consideration in algorithm selection was speed. While an algorithm may be perfectly accurate in its selection of a closest match to a string this means little in practice if the algorithm has an untenably long run time. To this end the speed of the algorithms were compared. To accomplish this each method was used to compare a series of one million random strings of a set length. The distance metric was calculated between each of these strings against a separate tag string of equal length. The amount of time it took for all of the strings to be compared was recorded. This was done for each algorithm with strings ranging in size from 4 characters to 100 characters. A graph of the time in milliseconds (ms) for each algorithm is given in Figure 8.2.

As can be seen from Figure 8.2 there were noticeably different rates of growth for each algorithm. The Jaro and Optimal String Alignment algorithms were the quickest, each growing at very slow rates with Jaro being slightly faster overall. Jaro never surpassed 9000ms while Optimal String Alignment remained below 13000 ms (milliseconds). There did not appear to be an appreciable difference in the rate of growth between these two. The rates of growth for the Jac-

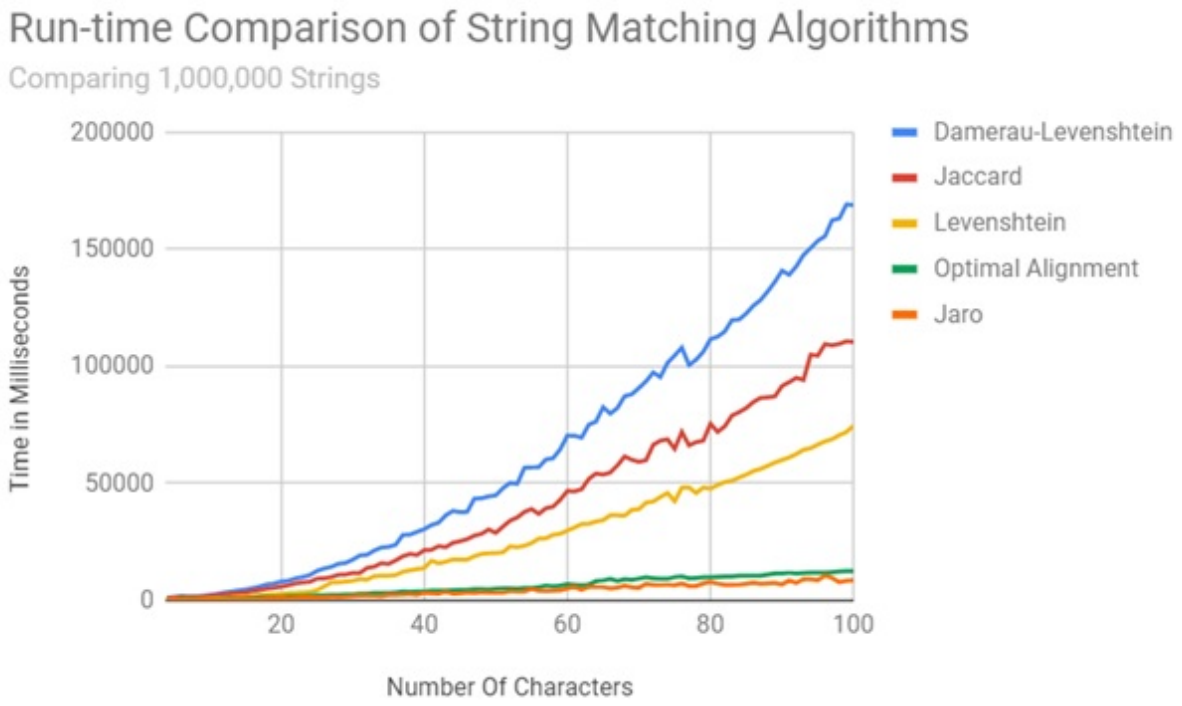

Figure 8.2: Runtime analysis of various algorithms in milliseconds.

card, Levenshtein, and Damerau-Levenshtein algorithms appeared to be exponential. However, their growth seemed to be occurring at different exponential rates. Damerau-Levenshtein was increasing the most rapidly overall. Run times for very long strings exceeded 150,000 ms. Jaccard was not far behind, taking up to 100,000ms for lengthy strings. Levenshtein was somewhat more nimble, never exceeding 75,000 ms. The results indicate that if the speed of the algorithm was the chief concern then Jaro or Optimal String Alignment would be the clear choice. Considering both timing and accuracy a clear winner emerges. The Optimal String Alignment Algorithm was both very fast and extremely accurate. Although any of the three Levenshtein variants were more than accuracy enough Levenshtein and Damerau-Levenshtein suffered from unacceptable growth in runtime. While the Jaro algorithm was marginally faster than the Optimal String Alignment algorithm, Jaro was significantly less accurate and displayed a similar level of growth. Taking both of these factors into consideration Optimal String Alignment was chosen as the preferred method.

8.2 Search for shorter signatures

The length of the signature plays a very important role in this domain. As mentioned earlier, shorter signatures imply less cost of synthesizing the signature and shorter signatures will be less likely to impact the existing functionality and stability of the plasmid during signature embedding. In our work, we simplified the identity-based signature(IBS) scheme proposed by Adi Shamir to gain signature length. The Shamir's IBS scheme and consequently our simplified Shamir's IBS is based on multiplicative groups over integers. There is no other identity-based signature scheme that is based on multiplicative groups over integers. The other IBS schemes are based on elliptic curve cryptography and pairing-based cryptography. We have used both of them to generate shorter signatures than the simplified Shamir's IBS. We will first briefly discuss the working principles of elliptic curve cryptography and then discuss the identity-based signature scheme based on elliptic curves called "Identity-based elliptic curve digital signature algorithm"(IBECDSA). Then we will briefly discuss the basics of pairing-based cryptography and then discuss some of the identity-based signature schemes based on pairing-based cryptography.

The use of elliptic curves in cryptography was suggested independently by Neal Koblitz [21] and Victor S. Miller [22] in 1985. Elliptic curve cryptography algorithms entered wide use from 2004 to 2005. As of now, many standard cryptography algorithms that are widely used for secure communication such as the Diffie-Hellman (DH) key exchange protocol, Digital Signature Algorithm (DSA) have been replaced by their elliptic curve variants namely, Elliptic Curve Diffie-Hellman(ECDH) key exchange protocol and Elliptic Curve Digital Signature Algorithm (ECDSA).

The primary benefit promised by elliptic curve cryptography is a smaller key size, reducing storage and transmission requirements, i.e. that an elliptic curve group could provide the same level of security afforded by an RSA-based system with a large modulus and correspondingly larger key: for example, a 256-bit elliptic curve public key should provide comparable security to a 3072-bit RSA public key. The U.S. National Institute of Standards and Technology (NIST) has endorsed elliptic curve cryptography in its Suite B set of recommended algorithms, specifically elliptic curve Diffie-Hellman (ECDH) for key exchange and Elliptic Curve Digital Signature Al-

gorithm (ECDSA) for digital signature. The U.S. National Security Agency (NSA) allows their use for protecting information classified up to top secret with 384-bit keys. Elliptic Curve cryptography is particularly suited to applications where security is needed but lacks the power, storage and computational power that is necessary for our current cryptosystems. The following figure explains the key size needed in elliptic curves compared to traditional RSA/DSA.

Symmetric scheme (key size in bits)	ECC-based scheme (size of *n* in bits)	RSA/DSA (modulus size in bits)
56	112	512
80	160	1024
112	224	2048
128	256	3072
`92	384	7680
256	512	15360

Figure 8.3: Comparison of key-size required in different cryptography schemes.

8.2.1 Brief background on elliptic curve cryptography

The main difference between RSA/DSA and elliptic curve cryptography is the fact that we are not dealing with numbers or integers anymore. We are dealing with points on the curve. A detailed explanation of elliptic curve cryptosystems and its working principles can be found here [23]. Apart from this, there are several other resources on the internet which an interested reader can find. We will be going through a very short introduction to elliptic curve cryptography.

An *elliptic curve* is a plane curve defined by an equation of the form :

$$y^2 = x^3 + ax + b \ (mod\ p)$$

where p is a prime number, a and $b \in \mathbb{Z}_p$, and $4a^3 + 27b^2 \neq 0$. The following figure shows some examples of elliptic curves.

The modulo p, creates a group of points for a curve. In elliptic curve cryptography, we work with this group of points instead of integers as in RSA/DSA. A group in mathematics is a set for

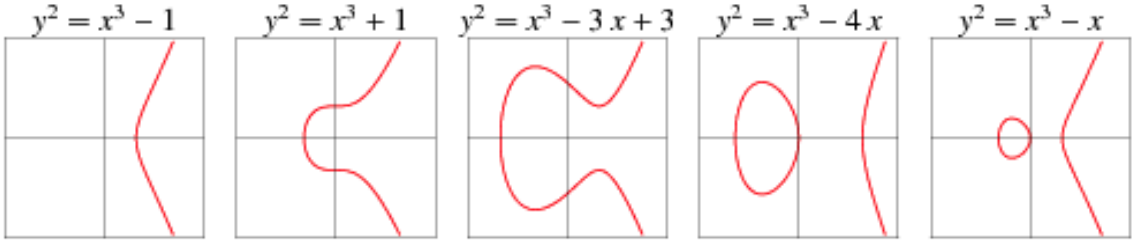

Figure 8.4: Examples of elliptic curves.

which we have defined a binary operation that we call addition and indicate with the symbol $+$. In order for the set $\mathbb{G}$ to be a group, addition must defined so that it respects the following four properties:

1. **closure**: if a and b are members of $\mathbb{G}$, then $a + b$ is a member of $\mathbb{G}$;
2. **associativity**: $(a + b) + c = a + (b + c)$;
3. there exists an **identity element** 0 such that $a + 0 = 0 + a = a$;
4. every element has an **inverse**, that is: for every a there exists b such that $a + b = 0$.

   If we add a fifth requirement:

5. **commutativity**: $a + b = b + a$, then the group is called **abelian** group.

With the usual notion of addition, the set of integer numbers $\mathbb{Z}$ is a group (moreover, it's an abelian group). The set of natural numbers $\mathbb{N}$, however, is not a group, as the fourth property can't be satisfied.

We can define a group law for ellipctic curves as follows:

- the elements of the group are the points of an elliptic curve;
- the **identity element** is the point at infinity 0;
- the **inverse** of a point P is the one symmetric about the x-axis;

- **addition** is given by the following rule: given three aligned, non-zero points $P, Q, \ and \ R$, their sum is $P + Q + R = 0$.

Point addition

With 2 distinct points, P and Q, addition is defined as the negation of the point resulting from the intersection of the curve, E, and the straight line defined by the points P and Q, giving the point,R.

$$P + Q = R$$

$$(x_p, y_p) + (x_q, y_q) = (x_r, y_r)$$

Assuming the elliptic curve, E, is given by $y^2 = x^3 + ax + b$, this can be calculated as:

$$\lambda = \frac{y_q - y_p}{x_q - x_p}$$

$$x_r = \lambda^2 - x_p - x_q$$

$$y_r = \lambda(x_p - x_r) - y_p$$

Point doubling

Where the points P and Q, are coincident (at the same coordinates), addition is similar, except that there is no well-defined straight line through P and Q, so the operation is closed using limiting case, the tangent to the curve, E, at P and Q. This is calculated as above, except with:

$$\lambda = \frac{3x_p^2 + a}{2y_p}$$

where a is from the defining equation of the curve, E, above.

Scalar multiplication

Other than addition, we can define another operation: scalar multiplication, that is:

$$n \cdot P = \underbrace{P + P + \cdots + P}_{n \text{ times}}$$

where n is a natural number. Written in that form, it may seem that computing nP requires n additions. But there are faster algorithms like **double and add** and others that can efficiently do this computation.

Given, n and P, there exists many algorithms that will give us the result $Q = nP$. But if we are given Q and P, and we need to find n, there is no polynomial time algorithm to find n. This is the crux of elliptic curve cryptography and is called the elliptic curve discrete logarithm problem.

Finally, the summary of the differences between working with groups of integers as in RSA/DSA and working with groups of elliptic curve points are described in the figure below:

Group	$\mathbb{Z}_p^*$	$E(\mathbb{Z}_p)$
Group Elements	Integers $\{1, 2, \ldots, p-1\}$	Points (x, y) on E plus $\mathcal{O}$
Group Operation	Multiplication modulo p	Addition of points
Notation	Elements: g, h Multiplication: $g \cdot h$ Inverse: g^{-1} Division: g/h Exponentiation: g^a	Elements P, Q Addition: $P + Q$ Negative: $-P$ Subtraction: $P - Q$ Multiple: aP
Discrete Logarithm Problem	Given $g \in \mathbb{Z}_p^*$ and $h = g^a \bmod p$, find a	Given $P \in E(\mathbb{Z}_p)$ and $Q = aP$, find a

Figure 8.5: Group operations on elliptic curves vs group operation on integers

8.2.2 Elliptic curve parameters

After getting familiar with the working principles of elliptic curves, the question arises how do we get a curve and its parameters. There are many curves out there which can be used for elliptic curve cryptography and some of them have been standardized and recommended by NIST after testing them rigorously. The full list of NIST recommended curves can be found here [24]. Below is an example of a curve from that document. This curve is called NIST P-192 or `secp192r1` because of the prime and the order of the curve has 192 bits.

The verifiably random elliptic curve domain parameters over $\mathbb{F}_p$ `secp192r1` are specified by the sextuple $T = (p; a; b; G; n; h)$ where the finite field $\mathbb{F}_p$ is defined by:

$$\begin{aligned} p &= \text{FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFFFF FFFFFFFF} \\ &= 2^{192} - 2^{64} - 1 \\ &= 6277101735386680763835789423207666416083908700390324961279 \end{aligned}$$

The curve $E : y^2 = x^3 + ax + b$ over $\mathbb{F}_p$ is defined by:

$$\begin{aligned} a &= \text{FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFFFF FFFFFFFC} \\ &= -3 \\ b &= 64210519\ E59C80E7\ 0FA7E9AB\ 72243049\ FEB8DEEC\ C146B9B1 \\ &= 2455155546008943817740293915197451784769108058161191238065 \end{aligned}$$

E was chosen verifiably at random as specified in ANSI X9.62 from the seed:

$$S = 3045AE6F\ C8422F64\ ED579528\ D38120EA\ E12196D5$$

The base point G in compressed form is:

$$G = 03\ 188DA80E\ B03090F6\ 7CBF20EB\ 43A18800\ F4FF0AFD\ 82FF1012$$

and in uncompressed form is:

$$G = 04\ 188DA80E\ B03090F6\ 7CBF20EB\ 43A18800\ F4FF0AFD\ 82FF1012$$
$$07192B95\ FFC8DA78\ 631011ED\ 6B24CDD5\ 73F977A1\ 1E794811$$

Finally the order n of G and the cofactor are:

$$n = FFFFFFFF\ FFFFFFFF\ FFFFFFFF\ 99DEF836\ 146BC9B1\ B4D22831$$
$$= 6277101735386680763835789423176059013767194773182842284081$$
$$h = 01$$

After loading these parameters we can proceed to a signature algorithm like ECDSA. The signature generated by this curve having 192 bits prime using ECDSA will have the same security of using a 1536 bit prime DSA.

8.2.3 ECDSA

Here we will describe how ECDSA [25] works. Note that ECDSA is not an identity-based signature algorithm that is every entity using ECDSA must have its own public-private key pair. But in order to move to identity-based ECDSA, we need to go through the steps in ECDSA as many steps are common. ECDSA is widely used today as a signature algorithm as it is fast, secure and offers a smaller signature size than DSA. One of the notable use of ECDSA is in Bitcoin to ensure that funds can only be spent by their rightful owners.

Suppose Alice wants to send a signed message to Bob. Initially, they must agree on the curve parameters $(CURVE, G, n)$. In addition to the field and equation of the curve, we need G, a base

point of prime order on the curve; n is the multiplicative order of the point G. These parameters can be taken from the NIST P-192 curve described in the above section.

Alice creates a key pair, consisting of a private key integer d_A, randomly selected in the interval $[1, n-1]$; and a public key curve point $Q_A = d_A \times G$. We use $\times$ to denote elliptic curve point multiplication by a scalar.

For Alice to sign a message m, she follows these steps:

1. Calculate $e = HASH(m)$, where $HASH$ is a cryptographic hash function, such as SHA-256.
2. Let z be the L_n leftmost bits of e, where L_n is the bit length of the group order n.
3. Select a cryptographically secure random integer k from $[1, n-1]$.
4. Calculate the curve point $(x_1, y_1) = k \times G$.
5. Calculate $r = x_1 \ mod \ n$. If $r = 0$, go back to step 3.
6. Calculate $s = k^{-1}(z + rd_A) \ mod \ n$. If $s = 0$, go back to step 3.
7. The signature is the pair (r, s).

It can be noted that the signature is a tuple (r, s) and both are modulo n. So the length of each variable is as long as n and consequently, the length of the signature is $2n$. Now if we use NIST P-192, the length of n is 192 bits, hence the signature length is 384 bits. If we use a more secure curve NIST P-256, the signature length will be 512 bits.

For Bob to authenticate Alice's signature, he must have a copy of her public-key curve point Q_A. Bob can verify Q_A is a valid curve point as follows:

1. Check that Q_A is not equal to the identity element O, and its coordinates are otherwise valid.
2. Check that Q_A lies on the curve.
3. Check that $n \times Q_A = O$.

After that, Bob follows these steps for signature verification:

1. Verify that r and s are integers in $[1, n-1]$. If not, the signature is invalid.
2. Calculate $e = HASH(m)$, where $HASH$ is the same function used in the signature generation.
3. Let z be the L_n leftmost bits of e.
4. Calculate $w = s^{-1}\ mod\ n$.
5. Calculate $u_1 = zw\ mod\ n$ and $u_2 = rw\ mod\ n$.
6. Calculate the curve point $(x_1, y_1) = u_1 \times G + u_2 \times Q_A$. If $(x_1, y_1) = O$ then the signature is invalid.
7. The signature is valid if $r \equiv x_1\ (mod\ n)$, invalid otherwise.

It is not immediately obvious why verification even functions correctly. To see why, denote as C the curve point computed in step 6 of verification,

$$C = u_1 \times G + u_2 \times Q_A$$

From the definition of the public key as $Q_A = d_A \times G$,

$$C = u_1 \times G + u_2 d_A \times G$$

Because elliptic curve scalar multiplication distributes over addition,

$$C = (u_1 + u_2 d_A) \times G$$

Expanding the definition of u_1 and u_2 from verification step 5,

$$C = (zs^{-1} + rd_A s^{-1}) \times G$$

Collecting the common term s^{-1},

$$C = (z + rd_A)s^{-1} \times G$$

Expanding the definition of s from signature step 6,

$$C = (z + rd_A)(z + rd_A)^{-1}(k^{-1})^{-1} \times G$$

Since the inverse of an inverse is the original element, and the product of an element's inverse and the element is the identity, we are left with

$$C = k \times G$$

From the definition of r, this is verification step 6.

8.2.4 Identity-based ECDSA

The identity-based variant of ECDSA called IBECDSA was proposed by Hu Jin et. al. in 2010 [26]. The algorithm utilizes many steps in the original ECDSA. The key derivation from the identity is the added step like any other identity-based signature scheme. The scheme consists of four steps: setup, extract, sign, verify.

Setup: The setup generates the elliptic curve parameters. One can utilize any recommended parameters from the NIST document [24] e.g. NIST P-192 or `secp192r1`. The secret key is x is randomly chosen and is called master secret key because the secret-keys of the users will be derived from this and their identity. The public key $P_{pub} = xG$ where G is the base point. The public key is also called master public key.

Extract: Takes as input the elliptic curve parameters, the master secret key x, and a user's identity and returns the users identity-based secret key. This step is perfomed the central authority for each user A with identity ID_A.

1. Choose a random $r \in Z_n^*$, computer $R_A = rG$ and $h = H(ID_A \parallel R_A)$. where H is a secure cryptographic hash function such as SHA-256.

2. Compute $s_A = (r + hx)\ mod\ n$.

User A's secret key is (R_A, s_A) and it is transmitted to A via a secure channel. A can validate her private key by checking whether the following equation holds:

$$s_A G = R_A + H(ID_A \parallel R_A) P_{pub}$$

Sign: To sign a message m, a user A with the curve parameters and the secret key (R_A, s_A) does the following:

1. Generate the signature (r, s) of the message m using the algorithm of ECDSA signature generation, where the secret key is s_A.

2. A's signature for the message m is (R_A, r, s).

Verify: To verify A's signature (R_A, r, s) on m, B needs to obtain the curve parameters and the associated master public key P_{pub}. The verification procedure is as follows:

1. Compute $h = H(ID_A \parallel R_A)$

2. Compute $Q = R_A + hP_{pub}$

3. Verify the signature (r, s) of the message m, using the ECDSA signature verification algorithm, where the public key is Q.

It can be noted that the signature generated by IBECDSA is a triple (R_A, r, s). We already know the r and s from ECDSA and their length. r and s is of length n each where n is the order. For

example using the NIST P-192 curve, where n is 192 bits r and s totals 384 bits. Lets look ar R_A, it is a point on the curve and so it is two parts to it, the x-coordinate and the y-coordinate. It would be better to represent the signature as (R_Ax, R_Ay, r, s) because R_A is internally two components. Each of the coordinates is again n bits in length so the total signature length in IBECDSA is $4n$. Hence using NIST P-192 the signature length would be 768 bits. However, there is a technique called point compression where instead of denoting a point on the curve by (x, y) it is sufficient to just denote it by x. This is because the y coordinate can be solved by plugging the value of x in the curve equation $E : y^2 = x^3 + ax + b$.

But it is to be noted that the y is not linear, it is a square term. So for every x there are two possible values of y. So which value of y is the correct value? For this inference, we need to encode some extra bits of information about y coordinate while compressing. Recall section 8.2.2, the base point has two representations, the compressed form, and the normal form. The normal form begins with 04. This is to indicate the receiver that both the values are passed and there is no need to calculate the y coordinate. In the compressed form we can see that the x coordinate us prefixed by 03. The convention of compressing is that, if the y coordinate has odd value then the x co coordinate is prefixed by 03. If it is even, the x coordinate is prefixed by 02. The receiver can use this information while calculating the corresponding y value from the compressed form. The length of the normal representation (x, y) is of length 2n - (n, n) but the compressed representation is not n rather it is $n + 8$. In this extra 8 bits or 1 byte, the extra information about the y coordinate is enclosed i.e. 02 or 03. Hence, it is possible to make this IBECDSA signature length $3n + 8$ bits instead of $4n$ bits.

We compared the IBECDSA signature length with the original Shamir's IBS scheme and the simplified Shamir's IBS scheme for various security parameters. The following table represents the signature length in each of the schemes and how much signature length gain we can have by switching to IBECDSA.

Table 8.2: Signature size comparison using Shamir's scheme, Simplified Shamir's Scheme and IBECDSA.

Key Size	Shamir's IBS Size	Simplified Shamir's IBS Size	Equivalent Key Size in ECC / Name of curve	IBECDSA Size Without Point Compression	IBECDSA Size With Point Compression
1024 bits	2048 bits	1024 bits	160 bits / NIST P-160	640 bits	488 bits
1536 bits	3072 bits	1536 bits	192 bits / NIST P-192	768 bits	584 bits
2048 bits	4096 bits	2048 bits	224 bits / NIST P-224	896 bits	680 bits
3072 bits	6144 bits	3072 bits	256 bits / NIST P-256	1024 bits	776 bits

8.2.5 Brief background on pairing-based cryptography

Pairing-based cryptography emerged in 2000 with Joux's tripartite Diffie Hellman [27] and Sakai's pairing-based cryptosystem [28]. But it gained a lot of popularity with Dan Boneh's groundbreaking solution to a long time unsolved problem of "Identity-based Encryption" [29]. Identity-based cryptosystems were first proposed by Shamir in 1984 [11]. The signature scheme we have used in our work is a simplified version of his proposed scheme. Although he came up with a working solution for identity-based signatures, he could not describe a fully working scheme for identity-based encryption. The identity-based encryption is possible only through pairing-based cryptography. Not only identity-based encryption but identity-based key exchange, identity-based identification, and many other schemes are possible because of pairing-based cryptography.

Pairing-based cryptography is an extension of elliptic curve cryptography. Pairing-based cryptography uses specially chosen curves that allows us to check more complicated equations over elliptic curve points. Pairing or bilinear pairing (and hence the name) can be thought of as an operation which when applied to elements of two groups, generates an output element which belongs to a third group. A detailed explanation of how pairing-based cryptography works can be found here [30]. There are many resources on the internet apart from the referred paper which can be used since this is one of the hot topics in cryptography. We will not be going through all the details of the working principles of pairing-based cryptography. We will be going through the definition of

bilinear maps, sketch a small example of how they work and then move on to describe the special curves that support pairing.

The following definition of "pairing" is commonly used in most academic papers [30] -

Let G_1, G_2 be two additive cyclic groups of prime order q, and G_T another cyclic group of order q written multiplicatively. A pairing is a map: $e : G_1 \times G_2 \rightarrow G_T$, which satisfies the following properties:

Bilinearity: $\forall a, b \in F_q^*, \forall P \in G_1, Q \in G_2 : e\left(aP, bQ\right) = e\left(P, Q\right)^{ab}$

Non-degeneracy: $e \neq 1$

Computability: There exists an efficient algorithm to compute e.

If the same group is used for the first two groups (i.e. $G_1 = G_2$), the pairing is called *symmetric* and is a mapping from two elements of one group to an element from a second group.

Some researchers classify pairing instantiations into three (or more) basic types:

1. $G_1 = G_2$
2. $G_1 \neq G_2$ but there is an efficiently computable homomorphism $\phi : G_2 \rightarrow G_1$.
3. $G_1 \neq G_2$ and there are no efficiently computable homomorphisms between G_1 and G_2.

The word "bilinear" here means it satisfies the constraints:

$$e(P, Q + R) = e(P, Q) * e(P, R)$$
$$e(P + S, Q) = e(P, Q) * e(S, Q)$$

Note that $+$ and $*$ can be arbitrary operators, when we create fancy new kinds of mathematical objects, abstract algebra doesn't care how $+$ and $*$ are defined, as long as they are consistent in the usual ways, eg. $a + b = b + a$, $(a * b) * c = a * (b * c)$ and $(a * c) + (b * c) = (a + b) * c$.

Lets take a simple example of pairing with numbers. Let's assume : $e(x, y) = 2^{xy}$. Then, we can see:

$$e(3, 4 + 5) = 2^{(3*9)} = 2^{27}$$

$$e(3, 4) * e(3, 5) = 2^{(3*4)} * 2^{(3*5)} = 2^{12} * 2^{15} = 2^{27}$$

We can observe the "bilinear" property. However, such simple pairings are not suitable for cryptography because the objects that they work on are simple integers and are too easy to analyze; integers make it easy to divide, compute logarithms, and make various other computations; simple integers have no concept of a "public key" or a "one-way function".

8.2.6 Pairing supported curve parameters

In elliptic curve parameters, we saw that the format of the parameters of the different curves is the same, the main difference is in the length of the prime used in the curve equation. Apart from that, all NIST defined curves have almost the same parameters like a, b, G, n etc. However, in pairing supported curves, there are a lot of different "types" of curves. Each type of curve is different in its base form or equation that defines it. Also, for each type, the prime can be different to provide better security. In our work, we have used the **jPBC: Java pairing based cryptography** library [31](http://gas.dia.unisa.it/projects/jpbc). The different types of curves and their parameters are provided in the library as "properties" files. The library has the following types of curves supported :

Type A: Type A pairings are constructed on the curve $y^2 = x^3 + x$ over the field F_q for some prime $q = 3 \ mod \ 4$. Both G_1 and G_2 are the group of points $E(F_q)$, so this pairing is symmetric. It turns out $\#E(F_q) = q + 1$ and $\#E(F_{q^2}) = (q + 1)^2$. Thus the embedding degree k is 2, and hence G_T is a subgroup of F_{q^2}. The order r is some prime factor of $q + 1$.

There is only one built-in type A curve in the library. The parameters in the file **a.properties** are -

type a

q 87807107996633125224377819847540498158068831994142082110286533992664756308
80222957078625179422662221423155858769582317459277713367317481324925129998224791

h 12016012264891146079388821366740534204802954401251311822919615131047207289359 70453110284480218390653778677 6

r 730750818665451621361119245571504901405976559617

exp2 159

exp1 107

sign1 1

sign0 1

Here, q is 512 bits or 64 bytes, r is 160 bits or 20 bytes. It is recommended to use at least 512-bit primes for type A curves. Other parameters can be generated using the following line of code -

ParametersGenerator pbcPg = new PBCTypeACurveGenerator(rBits, qBits);

The size of the group elements are as follows - since r is 20 bytes, any element in the group Z_r is 20 bytes. An element in the group G_1 is an element over F_q. Since q is 512 bits or 64 bytes, an element of the group G_1 is 2*64 = 128 bytes (each element is a point whose x cord is 64 bytes and y cord is 64 bytes). Since this curve uses symmetric pairing, the elements of the group G_2 and G_T are also 128 bytes each.

Type A1: Type A1 uses the same equation, but have different fields. It supports fields of composite order. The parameters are very close to type A, hence we are not listing the A1 parameters. The code to generate type A1 parameters is -

ParametersGenerator pbcPg = new PBCTypeA1CurveGenerator();

Type D: These are ordinary curves of with embedding degree 6, whose orders are prime or a prime multiplied by a small constant. A type D curve is defined over some field F_q and has order $h * r$ where r is a prime and h is a small constant. Over the field, F_{q^6} its order is a multiple of r^2. Typically the order of the curve E is around 170 bits, as is F_q, the base field, thus q^k is around the 1024-bit mark which is commonly considered good enough. These type of curves were discovered by Miyaji, Nakabayashi, and Takano and commonly known as MNT curves [32].

There are 3 type D curves built-in the library - d159, d201, and d224. The numbers represent the bits in the prime q. Let us look at the parameters in the **d159.properties** file -

type d

q 625852803282871856053922297323874661378036491717

n 625852803282871856053923088432465995634661283063

h 3

r 208617601094290618684641029477488665211553761021

a 581595782028432961150765424293919699975513269268

b 517921465817243828776542439081147840953753552322

k 6

Here, q is 160 bits or 20 bytes, r is 160 bits or 20 bytes. The parameters a and b are the coefficients in the equation $E : y^2 = x^3 + ax + b$. The size of the group elements are as follows - since r is 20 bytes, any element in the group Z_r is 20 bytes. An element in the group G_1 is an element over F_q. Since q is 160 bits or 20 bytes, an element of group G_1 is 2*20 = 40 bytes. The elements of the group G_2 are over F_{q^6} and hence they are 120 bytes. An element of G_T is also 120 bytes. Apart from the parameters listed above there are some more which we did not show, they are nk - number of points in $E(F_{q^k})$, hk where $nk = hk * r * r$, coeff0, coeff1, coeff2 and nqr - quadratic nonresidue in F_q. The other two files have the same parameters just the prime bits are more. The code to generate type D parameters is -

ParametersGenerator pbcPg = new PBCTypeDParametersGenerator(discriminant);

Type E: The CM (Complex Multiplication) method of constructing elliptic curves starts with the Diophantine equation $DV^2 = 4q - t^3$. If t=2 and $q = Dr^2h^2 + 1$ for some prime r (which we choose to be a Solinas prime) and some integer h , we find that this equation is easily solved with $V = 2rh$.

As q is typically 1024 bits, group elements take a lot of space to represent. Moreover, many optimizations do not apply to this type, resulting in a slower pairing. Hence this type E curve is not generally used in any schemes and so we will not be listing the parameters here. An interested user can look up the file **e.properties** in the jPBC library.

Type F: Using carefully crafted polynomials, k = 12 pairings can be constructed. Only 160 bits are needed to represent elements of one group and 320 bits for the other. Also, embedding degree k = 12 allows higher security short signatures. The equation is of the form $E : y^2 = x^3 + b$. These type of curves were discovered by Barreto and Naehrig and commonly known as BN curves [33].

There is only one type F curve built in the library. The parameters in the file **f.properties** are -

type f

q 205523667896953300194896352429254920972540065223

r 205523667896953300194895899082072403858390252929

b 402181051568677286985736685258831682221195154 13

beta 115334401956802802075595682801335644058796914268

alpha0 191079354656274778837764015557338301375963168470

alpha1 7144531790369634029619955607283694074171750637 5

Here, q is 158 bits or 20 bytes, r is 158 bits or 20 bytes. The parameter b is the coefficient in the equation $E : y^2 = x^3 + b$. The size of the group elements are as follows - since r is 20 bytes, any element in the group Z_r is 20 bytes. An element in the group G_1 is an element over F_q. Since q is 20 bytes, an element of group G_1 is 2*20 = 40 bytes. The elements of the group G_2 are 80 bytes and the elements of group G_T are 240 bytes. The code to generate type F parameters is -

ParametersGenerator pbcPg = new PBCTypeFCurveGenerator(rBits);

Type G: Type G curves are exactly the same structure as type D curves with the only difference being in the embedding degree. The embedding degree k is of 10. These types of curves were discovered by Freeman [34]. The parameters are exactly the same as described in type D curve parameters. There is only one type G curve built in the library called **g149.properties**. The prime q is 149 bits the order r is also 149 bits. The size of the group elements are as follows - since r is 149 bits or 19 bytes, any element in the group Z_r is 19 bytes. An element in the group G_1 is an element over F_q. Since q is 149 bits or 19 bytes, an element of group G_1 is 2*19 = 38 bytes. The elements of the group G_2 and G_T are 190 bytes each. The code to generate type G parameters is -

ParametersGenerator pbcPg = new PBCTypeGParametersGenerator(discriminant);

The following table represents the different types of curves built in the jPBC library and the different sizes of the group elements for each type of curve.

Table 8.3: Sizes of the different group element for the different types of curves built in jPBC.

Curve Name	**Bits in prime (q) /Bits in order (r)**	**Size of element in Z_r**	**Size of element in G_1**	**Size of element in G_2**	**Size of element in G_T**
a.properties	512 / 160	160 bits or 20 bytes	1024 bits or 128 bytes	1024 bits or 128 bytes	1024 bits or 128 bytes
a1.properties	1033 / 1022	1024 bits or 128 bytes	2080 bits or 260 bytes	2080 bits or 260 bytes	2080 bits or 260 bytes
d159.properties	159 / 158	160 bits or 20 bytes	320 bits or 40 bytes	960 bits or 120 bytes	960 bits or 120 bytes
d201.properties	201 / 181	184 bits or 23 bytes	416 bits or 52 bytes	1248 bits or 156 bytes	1248 bits or 156 bytes
d224.properties	224 / 224	224 bits or 28 bytes	448 bits or 56 bytes	1344 bits or 168 bytes	1344 bits or 168 bytes
e.properties	1020 / 160	160 bits or 20 bytes	2048 bits or 256 bytes	2048 bits or 256 bytes	1024 bits or 128 bytes
f.properties	158 / 158	160 bits or 20 bytes	320 bits or 40 bytes	640 bits or 80 bytes	1920 bits or 240 bytes
g149.properties	149 / 149	152 bits or 19 bytes	304 bits or 38 bytes	1520 bits or 190 bytes	1520 bits or 190 bytes

8.2.7 Identity-based signature schemes using pairings

There are several identity-based digital signature schemes using pairings. Some of the notable schemes are - *Sakai-Kasahara*, 2003 [35], *Sakai-Ohgishi-Kasahara*, 2000 [36], *Paterson*, 2002 [37], *Cha-Cheon*, 2003 [38], *Xun Yi*, 2003 [39]. The *Sakai-Kasahara* scheme described two types of identity-based signatures - one is called an El-Gamal type and the other is called the Schnorr type. We have implemented all the above schemes using the jPBC library. Based on the signature sizes that each of the schemes offers and also the cost of signature generation and verification, the best scheme is the Sakai-Kasahara Schnorr type. A detailed study of the schemes described above can be found in [40].

8.2.8 Sakai-Kasahara identity-based signature

We will first describe the Sakai-Kasahara Schnorr type scheme. Then we will discuss the signature length based on the different type of curves that can be used. Also, the time to generate and validate a signature depends on the type of the curve used. We will evaluate both aspects: time to sign and verify, and the size of the signature using this algorithm for all the different types of curves present in the jPBC library.

The Sakai-Kasahara Schnorr type scheme has four steps: setup, extract, sign and verify

Setup: The setup generates the curve parameters. The different curves provided in the jPBC library can be used to load the parameters. Let g_1 be the generator of G_1, g_2 be the generator of G_2. A random $x \in Z_n^*$ is chosen to be the master secret. Two public keys P_1 and P_2 are calculated as - $P_1 = x \cdot g_1$ and $P_2 = x \cdot g_2$. An embedding function H is chosen such that $H(0,1)^* \rightarrow G_1$.

Extract: Takes as input the curve parameters, the master secret key x, and a user's identity and returns the users identity-based secret key. This step is perfomed the central authority for each user A with identity ID_A.

1. For an identity ID_A, calculate $C_A = H(ID_A)$. That is map the identity string to an element of G_1.

2. Calculate $V_A = x \cdot C_A$.

User A's secret key is (C_A, V_A) and is sent to the user via a secure channel.

Sign: To sign a message m, a user A with the curve parameters and the secret key (C_A, V_A) does the following:

1. Choose a random $r \in Z_n^*$. Compute $Z_A = r \cdot g_2$.
2. Compute $e = e_n(C_A, Z_A)$, where e_n is the pairing operation.
3. Compute $h = H_1(m \parallel e)$, where H_1 is a secure cryptographic hash function such as SHA-256 and $\parallel$ is the concatenation operation.
4. Compute $S = hV_A + rC_A$.

A's signature for the message m is - (h, S)

Verify: The verification procedure is as follows:

1. Compute $w = e_n(S, g_2) * e_n(C_A, -hP_2)$
2. Check $H_1(m \parallel w) \stackrel{?}{=} h$

The above equation works because:

$$
\begin{aligned}
e &= e_n(C_A, Z_A) = e_n(C_A, r \cdot g_2) = e_n(C_A, g_2)^r \\
w &= e_n(S, g_2) * e_n(C_A, -hP_2) \\
&= e_n(hV_A + rC_A, g_2) * e_n(C_A, -hx \cdot g_2) \\
&= e_n(hx \cdot C_A + rC_A, g_2) * e_n(C_A, g_2)^{-hx} \\
&= e_n((hx + r) \cdot C_A, g_2) * e_n(C_A, g_2)^{-hx} \\
&= e_n(C_A, g_2)^{hx+r} * e_n(C_A, g_2)^{-hx} \\
&= e_n(C_A, g_2)^r
\end{aligned}
$$

Hence, $h = H_1(m \parallel e) = H_1(m \parallel w)$.

It can be observed that the signature is a tuple (h, S) where h is the result of a hash function. This length is dependent on the choice of the hash function. If we choose SHA-1, then h is 20 bytes and if we choose SHA-256 the length is 32 bytes. The value S is an element of the group G_1. Hence its length will be dependent on the curve type and the length of the prime. The following is the comparison of the signature length using the different curves without using any point compression.

Table 8.4: Signature size using different curves for the Sakai-Kasahara scheme.

Curve Name	**Signature Size using SHA-1 (Bytes)**	**Signature Size using SHA-256 (Bytes)**
a.properties	(20, 128) = 148	(32, 128) = 160
a1.properties	(20, 260) = 280	(32, 260) = 292
d159.properties	(20, 40) = 60	(32, 40) = 72
d201.properties	(20, 52) = 72	(32, 52) = 84
d224.properties	(20, 56) = 76	(32, 56) = 88
e.properties	(20, 256) = 276	(32, 256) = 288
f.properties	(20, 40) = 60	(32, 40) = 72
g149.properties	(20,38) = 58	(32, 38) = 70

A bit of tuning to further shorten signature size:

It can be noted that since S is a point on the curve, we can apply point compression technique to shorten its length. By default, the jPBC library does not have this built in but we have included this technique to shorten the signature size. So if the size of S is $2n$ bytes - the point compression technique will result in $n + 1$ bytes.

Another observation can be made when computing the verification step - $e_n(C_A, -hP_2)$. Here $-h$ is the negative hash value integer and we perform a scalar multiplication with the point P_2. There is only one integer group involved and that is Z_r where r is the order of the curve. So when performing that scalar multiplication $-h \cdot P_2$, jPBC internally converts the hash value to an element

of Z_r by modulo r. Hence, instead of writing the signature as (h, S) we can rewrite that as (R, S) where $R = h \ mod \ r$. This implies our signature is now of the form (R, S), where R is an element of the group Z_r.

Also, we can note the fact that if we use SHA-1, then it will not make that much of a difference in the size, but SHA-1 is not a recommended hash function nowadays, it has been replaced by SHA-256. In this case, we can gain some length since SHA-256 is 32 bytes. However, this tuning will degrade signature length for type A1 curves as the element in Z_r is 128 bytes. So if we use the A1 curve, it is better to keep the signature in the form (h, S). For all other types converting the signature to (R, S) is better. The following table represents the signature sizes using different curves after tuning the signature to (R, S) -

Table 8.5: Signature size using different curves for the Sakai-Kasahara scheme before and after tuning.

Curve Name	**Signature size before tuning and using SHA-256 (Bytes)**	**Signature size after tuning and using SHA-256 (Bytes)**
a.properties	(32, 128) = 160	(20, 65) = 85
a1.properties	(32, 260) = 292	(128, 131) = 259
d159.properties	(32, 40) = 72	(20, 21) = 41
d201.properties	(32, 52) = 84	(23, 27) = 50
d224.properties	(32, 56) = 88	(28, 29) = 57
e.properties	(32, 256) = 288	(20, 129) = 149
f.properties	(32, 40) = 72	(20, 21) = 41
g149.properties	(32, 38) = 70	(19, 20) = 39

Based on the signature size, it can be observed that the best performance is provided by the d159, f, and g149 curves. However, the length of the primes are a bit different and also the embedding degree is different. In the d159 curve, the prime is 159 bits and the embedding degree is 6. In the f curve, the prime is 158 bits and the embedding degree is 12. In the g149 curve, the prime is 149 bits and the embedding degree is 10. In the context of the small difference in signature sizes and the security related to each type, the better choice is the f curve.

The time to generate the signature and verify also depends on the type of the curve because of their properties. The following table denotes the time to sign and verify using the different types of curves. The times are measured after tuning the signature because it involves additional computation of point compression and decompression.

Table 8.6: Average time taken to sign and verify for different types of curves for the Sakai-Kasahara scheme.

Curve Name	**Average Time to sign (milliseconds)**	**Average Time to verify (milliseconds)**
a.properties	56	60
a1.properties	594	448
d159.properties	102	98
d201.properties	121	138
d224.properties	129	131
e.properties	262	214
f.properties	133	251
g149.properties	170	219

From the speed perspective, it can be seen that the a type curve is the fastest while generating and verifying the signature. But the size of the signature is way larger. The short signature size generating curves i.e. d159, f and g149 take a bit more time. The d159 curve takes around 100 ms for both signature generation and verification. The f curve takes around 150 ms to sign and 250 ms to verify, and the g149 curve takes around 200 ms to sign and verify. It is, therefore, a matter of priority - signature size over speed. If we need to sign and verify a lot of messages and we do not care about the signature size then type A curve is a good choice. However, if the size of the signature is more important than speed like in our application, the f type curve is a better option.

The f type curve offers the best of both world - the signature size is short, second only to the g149 curve. With respect to speed it offers almost the same performance as the g149 and the d159 curve. But the main reason to select the f type curve is that it offers the best security among the three as its embedding degree is higher.

We started with 512 base pair signatures using simplified Shamir's scheme. We then moved to elliptic curve based signature schemes and found that if we use the same security level the signature size would be 244 base pairs. For a better security level, the signature size would be 292 base pairs. Then we moved to pairing-based cryptography and found that for a tight security level, the length of the signature can be reduced to just 164 base pairs.

Chapter 9

Conclusion and Future Work

We discussed a novel effort to embed digital signatures into DNA molecules that are synthesized in the laboratory. Does digital security solutions, in particular cryptographic techniques, map well to a physical world? That was the big question that we had in mind for this research.

Synthetic DNA molecules are frequently shared physically. There is a need to bestow origin attribution properties to these molecules which are often licensed intellectual property. However, recent efforts to provide this property using watermarking techniques suffer from the problem that the watermark is independent of the DNA molecule (although it is embedded in the molecule). Thus, the watermark can potentially be removed from a physical DNA and embedded in another sample or replaced with another entity's watermark. We described our efforts to provide more secure origin attribution properties using digital signatures.

As a proof of concept, signatures were generated in the laboratory and inserted into two plasmids. The first, 401734, is a synthetic plasmid composed of two antibiotic resistance genes and an origin of replication. The second, 190691, is the commonly used standard vector pUC19. The physical signature embedding and validation protocol is a complex biological process, a detailed discussion of which involves extensive domain knowledge beyond the scope of this work. Consequently, we omit this discussion from this work. We performed experiments to determine if the addition of the signature would impact the characteristic and the function of the original plasmid. From the experiments, it was inferred that the addition of the signature does not impact the characteristic of the plasmid. The experiments that we did was on the 512 bp signature using the simplified Shamir's scheme. As improvements, we have come up with schemes that offer way smaller signature size - 164 bp. So we can be optimistic about the fact that this signature will also not impact the characteristics and the functionality of the plasmid.

There are several research questions that have not yet been answered and works that we plan to undertake in this area. First, our choices of parameters for the digital signatures as well as error

correction codes are not based on some mathematical models of properties of the DNA vis-a-vis its size. Rather they are based on domain knowledge and experimental analysis. Models of DNA properties can help in making optimal parameter choices to trade off the size of the sequences encoding signatures, security strengths of signatures and degree of error resiliency.

Related to this first research challenge is the question of whether embedding signatures is applicable for naturally occurring DNA and for DNA sequences larger than plasmids such as microbial or even plant or animal genomes. We have assumed that there is some leeway regarding the properties of DNA molecules that allows for the addition of a signature without altering function or stability. Current knowledge indicates that large portions of genomes probably do not play a functional role. So it is possible to embed a signature in these regions. However, from an evolutionary standpoint, sequences which do not play a functional role are less likely to be retained unchanged in the genome. Additionally, the sheer size of a genome means that mutations are more likely to occur and sequencing technologies are not yet robust enough to provide completely error-free whole genome sequences. In that case, what kind of security guarantees can we provide? For instance, could we apply digital signatures only to portions of larger sequences that are significant, such as those parts of a microbial genome which have been re-engineered for a specific purpose? Are there alternate signature schemes that are worth investigating that would be better-suited to whole genomes?

A third research area would involve signing and verifying the same DNA molecule multiple times by different users. Alice signs and sends a DNA sample to Bob and Bob validates Alice's DNA. Then Bob continues to modify it, signs it and sends it to Mallory. Can Mallory only verify Bob's signature, or is there a way for Mallory to track the entire pathway starting from Alice? It would be interesting to see if the concept of aggregate signatures can be applicable in these scenarios. Also, it would be interesting to see if we put a signature on top of an existing signature whether the characteristic of the DNA changes or not. If it does not, how many signatures can be inserted before the characteristics of the original DNA molecule begin to change? Also, if we cannot put multiple signatures within the same DNA molecule, how do we remove the signature that

was present before signing it again. Finally, does removing the signature also alters the property of the DNA?

A fourth problem that comes into focus is the issue of key revocation. What are the semantics of key revocation and how is this going to be achieved? Certainly key revocation in the cyber-bio world is very different from the digital world. If a key is revoked in the digital world, one can dis-validate any message signed with the revoked key. Signing, a duplicate copy of the same message is not expensive. However, it is not trivial to dis-validate a DNA molecule, synthesize a new one with the same properties and sign it with a new key. This causes us to rethink the whole key revocation model in the cyber-bio world.

A fifth research area involves studying the synthetic DNA sharing supply chain and developing techniques to ensure origin attribution. To illustrate the problem, recall that DNA molecules are shared to study their properties and possibly manipulate them to build a different DNA molecule. If a DNA molecule is licensed as an IP and is subsequently used in this manner, the synthesizer of the original molecule would like to get some credit for the root molecule. It will be interesting to study how if at all, this can be achieved.

Bibliography

[1] Peter Ney, Karl Koscher, Lee Organick, Luis Ceze, and Tadayoshi Kohno. Computer Security, Privacy, and DNA Sequencing: Compromising Computers with Synthesized DNA, Privacy Leaks, and More. In *Proc. of the 26th USENIX Security Symposium*, Vancouver, Canada, August 2017.

[2] Jean Peccoud, Megan F. Blauvelt, Yizhi Cai, Kristal L. Cooper, Oswald Crasta, Emily C. DeLalla, Clive Evans, Otto Folkerts, Blair M. Lyons, Shrinivasrao P. Mane, Rebecca Shelton, Matthew A. Sweede, and Sally A. Waldon. Targeted development of registries of biological parts. *PLOS ONE*, 3(7):1–7, 07 2008.

[3] Jean Peccoud, J. Christopher Anderson, Deepak Chandran, Douglas Densmore, Michal Galdzicki, Matthew W. Lux, Cesar A. Rodriguez, Guy-Bart Stan, and Herbert M. Sauro. Essential information for synthetic dna sequences. *Nature Biotechnology*, 29:22 EP –, Jan 2011.

[4] Jean Peccoud, Jenna E. Gallegos, Randall Murch, Wallace G. Buchholz, and Sanjay Raman. Cyberbiosecurity: From naive trust to risk awareness. *Trends in Biotechnology*, 36(1):4–7, 2018/03/29 XXXX.

[5] George M. Church, Yuan Gao, and Sriram Kosuri. Next-generation digital information storage in dna. *Science*, 337(6102):1628–1628, 2012.

[6] Daniel C. Jupiter, Thomas A. Ficht, James Samuel, Qing-Ming Qin, and Paul de Figueiredo. Dna watermarking of infectious agents: Progress and prospects. *PLOS Pathogens*, 6(6):1–3, 06 2010.

[7] Clyde A. Hutchison, Ray-Yuan Chuang, Vladimir N. Noskov, Nacyra Assad-Garcia, Thomas J. Deerinck, Mark H. Ellisman, John Gill, Krishna Kannan, Bogumil J. Karas, Li Ma, James F. Pelletier, Zhi-Qing Qi, R. Alexander Richter, Elizabeth A. Strychalski, Lijie Sun,

Yo Suzuki, Billyana Tsvetanova, Kim S. Wise, Hamilton O. Smith, John I. Glass, Chuck Merryman, Daniel G. Gibson, and J. Craig Venter. Design and synthesis of a minimal bacterial genome. *Science*, 351(6280), 2016.

[8] Daniel G. Gibson, John I. Glass, Carole Lartigue, Vladimir N. Noskov, Ray-Yuan Chuang, Mikkel A. Algire, Gwynedd A. Benders, Michael G. Montague, Li Ma, Monzia M. Moodie, Chuck Merryman, Sanjay Vashee, Radha Krishnakumar, Nacyra Assad-Garcia, Cynthia Andrews-Pfannkoch, Evgeniya A. Denisova, Lei Young, Zhi-Qing Qi, Thomas H. Segall-Shapiro, Christopher H. Calvey, Prashanth P. Parmar, Clyde A. Hutchison, Hamilton O. Smith, and J. Craig Venter. Creation of a bacterial cell controlled by a chemically synthesized genome. *Science*, 329(5987):52–56, 2010.

[9] Sarah M. Richardson, Leslie A. Mitchell, Giovanni Stracquadanio, Kun Yang, Jessica S. Dymond, James E. DiCarlo, Dongwon Lee, Cheng Lai Victor Huang, Srinivasan Chandrasegaran, Yizhi Cai, Jef D. Boeke, and Joel S. Bader. Design of a synthetic yeast genome. *Science*, 355(6329):1040–1044, 2017.

[10] Michael Liss, Daniela Daubert, Kathrin Brunner, Kristina Kliche, Ulrich Hammes, Andreas Leiherer, and Ralf Wagner. Embedding permanent watermarks in synthetic genes. *PLOS ONE*, 7(8):1–10, 08 2012.

[11] Adi Shamir. Identity-based cryptosystems and signature schemes. In George Robert Blakley and David Chaum, editors, *Advances in Cryptology*, pages 47–53, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.

[12] Jean Peccoud, J Christopher Anderson, Deepak Chandran, Douglas Densmore, Michal Galdzicki, Matthew W Lux, Cesar A Rodriguez, Guy-Bart Stan, and Herbert M Sauro. Essential information for synthetic DNA sequences. *Nature Biotechnology*, 29:22, January 2011.

[13] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

[14] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[15] James S Plank et al. A tutorial on reed-solomon coding for fault-tolerance in raid-like systems. *Softw., Pract. Exper.*, 27(9):995–1012, 1997.

[16] CKP Clarke. R&d white paper. *Reed-Solomon error correction," WHP*, 31, 2002.

[17] Yaniv Erlich and Dina Zielinski. Dna fountain enables a robust and efficient storage architecture. *Science*, 355(6328):950–954, 2017.

[18] O. Tornea and M. E. Borda. Dna cryptographic algorithms. In Simona Vlad, Radu V. Ciupa, and Anca I. Nicu, editors, *International Conference on Advancements of Medicine and Health Care through Technology*, pages 223–226, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[19] Dominik Heider and Angelika Barnekow. Dna-based watermarks using the dna-crypt algorithm. *BMC Bioinformatics*, 8(1):176, May 2007.

[20] W. W. Peterson and D. T. Brown. Cyclic codes for error detection. *Proceedings of the IRE*, 49(1):228–235, Jan 1961.

[21] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.

[22] Victor S Miller. Use of elliptic curves in cryptography. In *Conference on the theory and application of cryptographic techniques*, pages 417–426. Springer, 1985.

[23] Darrel Hankerson, Alfred J Menezes, and Scott Vanstone. *Guide to elliptic curve cryptography*. Springer Science & Business Media, 2006.

[24] SECG SEC. 2: Recommended elliptic curve domain parameters. *Standards for Efficient Cryptography Group, Certicom Corp*, 2000.

[25] T. Pornin. Deterministic usage of the digital signature algorithm (dsa) and elliptic curve digital signature algorithm (ecdsa). RFC 6979, RFC Editor, August 2013. http://www.rfc-editor.org/rfc/rfc6979.txt.

[26] Hu Jin, He Debiao, and Chen Jianhua. An identity based digital signature from ecdsa. In *Education Technology and Computer Science (ETCS), 2010 Second International Workshop on*, volume 1, pages 627–630. IEEE, 2010.

[27] Antoine Joux. A one round protocol for tripartite diffie–hellman. In *International algorithmic number theory symposium*, pages 385–393. Springer, 2000.

[28] Ryuichi Sakai. Cryptosystems based on pairing. *Proc. of SCIS2000, Jan.*, 2000.

[29] Dan Boneh and Matt Franklin. Identity-based encryption from the weil pairing. In *Annual international cryptology conference*, pages 213–229. Springer, 2001.

[30] Alfred Menezes. An introduction to pairing-based cryptography. *Recent trends in cryptography*, 477:47–65, 2009.

[31] Angelo De Caro and Vincenzo Iovino. jpbc: Java pairing based cryptography. In *Computers and communications (ISCC), 2011 IEEE Symposium on*, pages 850–855. IEEE, 2011.

[32] Atsuko Miyaji, Masaki Nakabayashi, and Shunzou Takano. New explicit conditions of elliptic curve traces for fr-reduction. *IEICE transactions on fundamentals of electronics, communications and computer sciences*, 84(5):1234–1243, 2001.

[33] Paulo SLM Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In *International Workshop on Selected Areas in Cryptography*, pages 319–331. Springer, 2005.

[34] David Freeman. Constructing pairing-friendly elliptic curves with embedding degree 10. In *International Algorithmic Number Theory Symposium*, pages 452–465. Springer, 2006.

[35] Ryuichi Sakai and Masao Kasahara. Id based cryptosystems with pairing on elliptic curve. *IACR Cryptology ePrint Archive*, 2003:54, 2003.

[36] R Sakai, K Ohgishi, and M Kasahara. Cryptosystems based on pairing, scis 2000-c20, jan. 2000. *Okinawa, Japan*.

[37] Kenneth G Paterson. Id-based signatures from pairings on elliptic curves. *Electronics Letters*, 38(18):1025–1026, 2002.

[38] Jae Cha Choon and Jung Hee Cheon. An identity-based signature from gap diffie-hellman groups. In *International workshop on public key cryptography*, pages 18–30. Springer, 2003.

[39] Xun Yi. An identity-based signature scheme from the weil pairing. *IEEE communications letters*, 7(2):76–78, 2003.

[40] Jerome A Solinas. Id-based digital signature algorithms. In *Slide Show presented at 7th Workshop on Elliptic Curve Cryptography (ECC 2003)*, 2003.