Synthesizing DNA Molecules with Identity-based Digital Signatures to Prevent Malicious Tampering and Enabling Source Attribution

Diptendu Mohan Kar [a] Indrajit Ray [a] [*] Jenna Gallegos [b] Jean Peccoud [b,??] Indrakshi Ray [a]
[a] *Department of Computer Science, Colorado State University, Fort Collins, Colorado, USA*
E-mails: diptendu.kar@colostate.edu, indrakshi.ray@colostate.edu
[b] *Department of Chemical and Biological Engineering, Colorado State University, Fort Collins, Colorado, USA*
E-mails: jenna.gallegos@colostate.edu, jean.peccoud@colostate.edu
[c] *GenoFAB, Inc., Fort Collins, Colorado, USA*
E-mails: jenna.gallegos@colostate.edu, jean.peccoud@colostate.edu

Abstract. The area of synthetic biology has seen rapid progress in recent years. Commercial DNA synthesis is increasingly used to create new biological organisms that do not exist in the natural world. A major concern in this domain is that a malicious actor can potentially tweak with a benevolent synthesized DNA molecule and create a harmful organism [1] or create a DNA molecule with malicious properties. To detect if a synthesized DNA molecule has been modified from the original version created in the laboratory, the authors in [2] had proposed a digital signature protocol for creating a signed DNA molecule. It uses an identity-based signatures and error correction codes to sign a DNA molecule and then physically embed the digital signature in the molecule itself. However there are several challenges that arise in more complex molecules because of various forms of DNA mutations as well as size restrictions of the molecule itself that determine its properties. In this work, we extend the work in several directions to address these problems. A second major concern with synthesized DNA is that it is an intellectual property. In order to allow its use by third parties, an annotated document of the molecule needs also to be distributed. However, since the molecule and document are two different entities, one being a physical product and the other being a digital one, ensuring that both are distributed correctly together without tampering is challenging. Additionally, there may be portions in the documents that the creator of the molecule may not want to share. In this work, we also address this problem by transforming the document into a DNA molecule and embedding it within the original molecule together with the signature.

Keywords: Cyber-Bio Security, Identity-Based Signatures, Reed-Solomon Codes, Compression, Approximate String Matching, Pairing-Based Cryptography, Synthetic DNA

1. Introduction

Synthesizing DNA molecules in the laboratory is quite common these days. Such a synthetic DNA molecule is often a licensed intellectual property. DNA samples are shared between academic laboratories, ordered from DNA synthesis companies and manipulated for a variety of purposes, for example, to create new biochemicals, reduce the burden of diseases, improve agricultural yields or simply to study

the DNA's properties and improve upon them. There have also been instances of new biological organisms that do not exist in the natural world being created using synthesized DNA [1]. While the vast majority of such activities are pursued for beneficial purposes, there are concerns that malicious users can use the technology malevolently, for example, to make harmful biochemicals, or render existing bacteria more dangerous [1]. Recently, a DNA-based security exploit was demonstrated as a proof-of-concept, where a synthesized DNA was used to attack a DNA sequencer that has been deliberately modified with a vulnerability [3]. Preventing such malicious use of synthesized DNA is beyond the scope of this current work. However, attribution of a physical DNA sample and establishing proof of origin can contribute significantly to deter such malicious activities.

Following the anthrax attack of 2001, there is an increased urgency to employ microbial forensic techniques to trace and track agent inventories. For instance, it has been proposed that unique watermarks be inserted in the genome of infectious agents to increase their traceability [4]. The synthetic genomics community has demonstrated the feasibility of this approach by inserting short watermarks into DNA without introducing significant perturbation to genome function [5–8]. The use of watermarks has also been proposed in order to identify genetically modified organisms (GMOs) or proprietary strains. Heider et al. [9], for example describe DNA-based watermarks using DNA-Crypt algorithm. This technique is applicable to provide proof of origin to a DNA molecule. However, there is a major shortcoming with all watermark based approaches. The watermark in all these works is generated from an arbitrary binary data and added to the original sequence, and so is independent of the original sequence and provides no integrity of the actual DNA sequence.

To enable effective trace back and eliminate the limitation of watermark-based approaches, Kar et al. [2] had proposed a scheme to create digital signatures of DNA molecules in living cells. The main idea is as follows: Take a DNA molecule and sequence it. The result is a string over the alphabet A, C, G, and T, representing the four nucleotide building blocks of DNA. The output of the sequencer is stored in what is called a FASTA file. For interpretability reasons, the FASTA file is annotated by the researcher to create another file called the GenBank file. The authors then use Shamir's identity-based signature scheme [10], Reed-Solomon error-correction codes [11, 12] and the 16 digits Open Researcher and Contributor ID (ORCID – https://orcid.org) of the researcher to create a digital signature of the string in the FASTA file. The resulting signature is in the form of a DNA sequence which is now synthesized as a physical molecule. Finally, the signature molecule is inserted into the original DNA molecule using DNA editing tools to obtain a signed 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.

However, there are significant challenges related to the placement of the signature within the molecule and various types of mutations in more complex molecules that Kar et al. do not address (discussed in more details in Section 2). The current work improves the previous scheme to address these problems (Sections 4 and 5). Moreover, we would like to shorten the size of the signature sequence as much as possible without impacting security. While biologists believe that the size of the DNA has a correlation with its properties within certain bounds, they still do not know by how much a DNA molecule can be expanded without changing the properties of interest. The current work explores other cryptographic algorithms towards this end as well as signature compression schemes (Section 6).

In order to verify a signature, the verfier needs to first identify which part of a signed message is the signature. There are three choices - before the actual message, after it or within it. If the signature is placed within the message, identifying the signature becomes problematic and needs to be addressed during the signature placement time. (The other two cases when the signature is placed before or after

the message, are easily addressed by using a counting-length approach.) Unfortunately, when working with DNA molecules, it may not always be possible to place the signature at the end or the beginning of the message. This is because there can be a feature present at these locations. The possible places to place the signature are most likely to be somewhere within the original sequence. For this reason the file that documents the DNA molecule needs to be also accessible to the receiver. Now, these molecule documentation files are created using gene editors that maintain databases of DNA molecule properties. It is conceivable that the databases are updated with location of signature in the molecule. However, these databases may not be consistent across different editors in the sense that receiver's gene editor may not have all the information about the same set of molecules that the sender's gene editor has. In addition, there is no universal file format used by gene editors, unlike the FASTA file format.

The document is needed not only to identify and retrieve the signature but also to address the problem of sequence alignment. Thus, we recommend sending the document created by the creator of molecule be also sent to the receiver. This is described in [2]. The updated algorithm described in section 4 takes into account for any circular or double helix permutations, and the verifiers sequence need not be in perfect alignment with the senders sequence. The purpose of sharing the digital file is not just limited to aligning sequences at the receivers end. The digital file contains descriptions about the physical DNA. There are gene editing software such as Snapgene which can be used to generate documentations or annotations about a sequence. A user can sequence a DNA, provide the FASTA file containing just the raw sequences to the software. The software searches its database for descriptions matching any subsequence within the FASTA file and generates a genbank file which contains the descriptions (from the database) and the original sequence. Refer to Figure 6 and Figure 5, the genbank file is generated from the FASTA file, the keyword "ORIGIN" separates the actual sequence and the descriptions. Although the software like Snapgene can generate documentation for some sequences, it cannot describe what is not present in its database. For this, the software have the flexibility of user addition, deletion and update. The user can add more descriptions, can delete wrong descriptions populated by Snapgene, and update a description to include or exclude information.

When a DNA sample is shared, the receiver can only generate the descriptions that the software can generate using its database. The extra descriptions written by the sender cannot be obtained by the receiver unless the digital file is shared. For this reason the digital file also needs to be shared with the receiver along with the physical sample. However, this file, being a separate entity from the DNA molecule, is not strongly tied to the molecule. Sending it separately from the molecule can result in the receiver using a different document file from the one that creator generated. Thus, we propose embedding the document in the DNA molecule itself and sending the whole signed and documented molecule to the receiver. Only this way would the receiver be obtain all the descriptions which the sender intended to share. However, note that placing the document within the molecule brings up the same issue of locating it as the signature. In this work, we develop the necessary techniques to generate, embed, retrieve and validate signatures as well as document in/from the molecule. The details of these procedures are explained in Sections 4, 5 and 8.

We would like to note here that there can sometimes be reasons why the originator may not want to share the entire document file. While the creator may be willing to divulge the properties of the synthesized DNA as a whole, s/he may not be willing to divulge properties of some sub-sequences because of reasons related to protecting intellectual properties or preventing mal-actors from exploiting some of the properties for nefarious purposes. Sending the entire document file may jeopardize these confidentiality needs. However, we do not address these concerns in the current work but is left as

a future topic. We assume that if and when the creator is willing to share the document (may be by encrypting parts that cannot be revealed to the receiver) it is embedded in the molecule.

2. Limitations of Earlier Work and Current Contributions

2.1. Cyclic shifts and reverse complement

In [2], the signer is *required* to send the GenBank file along with the physical DNA sample to the receiver. This is because the GenBank file is needed to align the FASTA file (which is the output of a DNA sequencer) in the same order as during the signature generation. Plasmid DNA is *cyclic* and *double-stranded*. Following DNA sequencing, any cyclic permutation of the DNA structure is possible. A sequence represented in a FASTA file, and consequently the GenBank file, is thus one of several possible linear representations of a circular structure. For example, in a FASTA file if the sequence was "ACGGTAA", and the same sample is sequenced again, the FASTA file might read as "TAAACGG".

Moreover, since DNA is composed of two complementary, anti-parallel strands, a DNA 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 a polarity with one end represented as 5′ and the other represented as 3′. One strand adheres to its reverse complement in an 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". Thus, by combining these two properties, for a DNA that contains N number of bases, the possible number of correct representations of the same sample is $2N$: N cyclic permutations plus each reverse complement.

Let us now consider the implications of this characteristic of DNA on the signature generation and verification. The sender has a sequence say "ACCGTT". The sender synthesizes the sequence and sends it to the receiver. The receiver after sequencing with an automated DNA sequencer may not have exactly "ACCGTT". It can be "TTACCG" which is a cyclic permutations. The receiver can also get something like "AACGGT" which is the reverse complement of "ACCGTT". Owing to such domain challenges, the signature verification procedure is not as simple as in digital messages.

Let us assume the signature sequence is "TTAA". (The actual signature length is 512 base pairs). In [2], the authors had defined a start and an end tag which served as delimiters for the signature. Let "ACGC" and "GTAT" be the start and end tags. For this discussion, we will use the term message to denote some linear representation of the sequence generated by a DNA sequencer. There can be three cases for including the signature sequence in the DNA sequence:

(1) **Append the signature after the message:** In this case, the sender's message with the signature embedded looks like - "ACCGTT ACGC TTAA GTAT". The receiver, after sequencing the signed DNA sample may get something like – "GTT ACGCTTAA GTAT ACC" or something else depending on which base position the sequencer considers as the beginning of the sequence. In the permutation, the DNA sequencer assumed the 4^{th} base from the left as the start of the sequence. The message is split but the delimiters and signature are intact. The simplest way to extract the message and signature is to append the extracted sequence to itself. With the permutation, this becomes "GTT ACGC TTAA GTAT ACC || GTT ACGC

TTAA GTAT ACC". Now we can extract the message which will be contained between two "ACGC TTAA GTAT" when the string is wrapped around. The receiver reconstructs the message which is "ACCGTT". The receiver can then invoke the verification. Note that this scheme works no matter which position the sequence considers as the start of the sequence.

(2) **Append the signature before the message:** In this case, the sender's message with signature looks like - "ACGC TTAA GTAT ACCGTT". The receiver after sequencing the DNA might get something like - "AA GTAT ACCGTT ACGC TT". We observe that this is the same as the previous case. We can append the extracted sequence to itself – "AA GTAT ACCGTT ACGC TT || AA GTAT ACCGTT ACGC TT. Thus we can extract the message using the same procedure as above and then invoke the verification.

(3) **Append the signature between the message:** In this case, the sender's message with signature might look like - "ACC ACGC TTAA GTAT GTT". The receiver after sequencing the DNA might get something like "ACGC TTAA GTAT GTT ACC". The problem occurs in this scenario. Even if we append the extracted sequence, we will not be able to recover the message. After appending the sequence we get "ACGC TTAA GTAT GTT ACC || ACGC TTAA GTAT GTT ACC". We can observe that the sequence contained by the two "ACGC TTAA GTAT" is "GTTACC". This is not the message the sender signed. The sender signed the message on "ACCGTT". But the receiver has no way of knowing this and hence the verification will fail since the message is not the same even though there is no modification to either the message or the signature.

The problem of recovering the message only occurs when the signature is placed within the message. The other two cases when the signature is placed before or after the message works perfectly fine. However, when working with DNA molecules, it may not always be possible to place the signature at the end or the beginning of the message. This is because there can be a feature present at that location. The possible places to place the signature are most likely to be within the original sequence. For this reason the GenBank file needed to be shared. Only this way would the receiver be able to align the sequence in the same order that the sender had when he signed.

There are several reasons why we may not want to share the GenBank file. The GenBank file is created by the originator of the DNA molecule using a gene editor. Its only purpose is to annotate the DNA sequence. If the DNA is an intellectual property, then the creator of the DNA will be annotating the DNA's GenBank file with different features of different subsequences of the DNA. While the creator may be willing to divulge the property of the synthesized DNA as a whole, s/he may not be willing to divulge properties of various subsequences. Sending the GenBank file jeopardizes the latter. Moreover, gene editors maintain databases of DNA molecule properties. However, these databases may not be consistent across different editors in the sense that receivers gene editor may not have all the information about the same set of molecules that the sender's gene editor has. Finally, the GenBank file format is not the only format used by gene editors, unlike the FASTA file format. In order to not share the GenBank file with the receiver, we have changed the signature generation procedure in this work, such that the verification is not dependent on where the signer placed the signature. The details of the new signature generation procedure are explained in section 4.

2.2. Mutations in identifying tags

In our previous work, we defined two identifying tags to demarcate the signature. The start tag was chosen as "ACGCTTCGCA" and the end tag as "GTATCCTATG". These two delimiters were chosen

not just randomly but for very specific reasons. First biologists typically have some idea about what DNA sequence will not occur in their specific project. Thus they can choose delimiters from these non-occurring sequence. Second, from these possible delimiters, they will choose the ones that are simple to synthesize and assemble since DNA synthesis is expensive. Finally, they will choose a sequence that are easy to identify visually, are unlikely to develop secondary structures and have a balanced number of "`A, C, G and T`"s. Our domain experts selected these delimiters for this project. We also used error correction code to tolerate mutations within the DNA. However, we assumed that the start and end tag do not mutate. If they do, our previous work will fail to locate the signature and consequently, it will not be possible to verify the signature.

To overcome this limitation, in this work we propose using partial matching techniques such that the start and end tag can be located approximately. This is used in conjunction with error correction codes. Note that since the start and end tags are fixed, we know what we are searching for in the DNA molecule. For example, we may want to look for strings similar to "`ACGCTTCGCA`" such as "`GCGCTTCGCG`". The different techniques we use for achieving this are discussed in section 5. It has to be noted that the error-correction code that is used can only tolerate substitution errors, whereas the partial matching technique can work for any type of errors.

2.3. Signature length

The length of the signature plays a very important role in this biology domain. Shorter signatures imply less cost of synthesizing the signature into a physical DNA molecule. Shorter signatures will also be less likely to impact the existing functionality and stability of the plasmid during signature embedding. Previously, we used 1024 bit keys and that resulted in 512 base-pair signature. However, 1024 bit keys are no longer considered very strong and not recommended in practice for digital signatures. Generally, 2048 bit keys are used. In our domain, this would result in a 1024 base pair signatures. This length has a higher probability of affecting the characteristics and stability of the plasmid. Furthermore, when synthesizing the signature, presently with a 512 base pair signature the cost is $46.08 - 512 base pairs at $0.09 per base pair. With a 1024 base pair signature, even if the plasmid remains stable and functional, the cost of synthesizing the signature would be $92.16. The new signature scheme with a shorter signature is described in section 6.

3. Overview of the DNA Signature Workflow

We begin by providing an overview of the DNA signature workflow to give the reader a general idea of the biological ecosystem that is involved. We also discuss the threat model that we assume for the rest of the work.

Our biology-related experiments have used plasmid DNA and in the following we use "plasmid" to refer to physical DNA molecules that are being created, used or modified in the laboratory. We use the term "sequence" to mean the digital representation of the DNA molecule order after it has been sequenced by a DNA sequencer. Note that in the biology domain the term "sequence" is used to imply the order of nucleotides in a DNA molecule. It can mean both in the bilogical sense or the physical sense. Bilogical operations on DNA molecules are conducted on both on the physical DNA molecule sequence as well as the digital sequence (domain of bio-informatics/computational biology). For this work, we limit the term "sequence" to mean the digital representation. The documentation, digital signing and

verification processes in this work are all conducted on the sequence. The document as well as signature are digital sequences that are converted back to physical molecules and embedded in the DNA molecule.

The DNA signature workflow is presented in Figure 1. There are seven major entities that are involved in the workflow: (1) Alice, the Originator, creates a plasmid and its sequence using a sequencer. Alice also creates a plasmid from a given sequence (which, in our case, will be a document of a plasmid and a signature). Alice uses two services (which can be local processes) - Doc creator and Signer. (2) Doc creator service prepares a document of a sequence. The output is also a sequence. (3) Signer creates a identity-based digital signature of a sequence. (4) Ellen, the User, obtains a plasmid for her use. The plasmid is received via a out-of-band physical communication channel. As needed Ellen creates a sequence of the received plasmid. (5) Verifier verifies a signed signature and returns Pass/Fail depending whether the verification is successful or not. If Fail, verifier also provides explanation of failure. (6) Doc user retrieves document from a sequence and converts it to a form understandable by a biologist. (7) An identity-based signature (IBS) key authority is a trusted entity that creates and distributes keys needed for an identity-based signature scheme from an ORCID id. We assume an out-of-band secure channel for such key distribution.

We assume that Alice has also created the plasmid in her lab and in that sense is the originator. When the sequence document and signature is ready, Alice modifies the original plasmid to embed the document and signature. 2) Ellen, the User/Verifier needs to use the plasmid. Ellen obtains the signed plasmid and uses the signature to verify if the DNA sequence offered by Alice has remained unchanged after signing and the sequence did indeed originate from Alice. In addition, Ellen extracts the documentation of the DNA sequence from the plasmid and uses it as needed. 3) A central authority that provides the signer with a token that is associated with the signer's identity. We assume that the central authority is secure and trusted by all participants in the system.

The steps of the DNA sign-share-verify workflow is shown in Figure 1. Alice creates a plasmid in her laboratory, submits her ORCID and the plasmid ID to the IBS Authority and gets a corresponding IBS token that is provided to the Signer. She also creates a sequence for this plasmid (FASTA file) . Next she uses the Doc creator to create a documentation of the sequence (a GENBANK file). She uses the signature generating service, Signer, to create a DNA signature sequence that she will add to her design. This sequence is the digital signature and the documentation about the plasmid. It is generated using the signature algorithm described in section 4 and the documentation algorithm described in section 8. The digital signature and documentation is then inserted in the sequence between two demarcating sequences used to identify the signature from the rest of the original plasmid's sequence. We rely on the biological properties of the DNA molecule to determine what is the best position within the molecule where the signature-documentation sequence can be inserted.

Biologists have observed that there are segments within a DNA molecule that do not contribute to the properties of the molecule that are of interest – the so called "junk DNA". (In fact, one of the roles that the DNA documentation plays is to identify what are the segments that contribute to the biological properties.) Note that this is not to say that the "junk DNA" does not contribute any properties. It is just that those properties are not of interest to the biologist. The signature-documentation sequence is inserted around such segments. We would like to emphasize two points here: (i) No segment is removed from the original molecule. This means that when the signature-documentation sequence is inserted *there is an expansion in the size of the DNA molecule* (see Figure 2). Biologists have experimentally observed that DNA molecules can tolerate some expansion in size without affecting the properties of interest. The tolerable fraction of increase is a function of the size of the original molecule. However, since we do not know the exact fraction of expansion that can be tolerated, *a significant focus of this work is in reducing*

the size of the signature-docmentation sequence. Most of the effort has been spent on experimentally determining what works. We urge the reader to keep this context in mind while reading this paper. (ii) It is possible that some DNA molecule does not allow any expansion in the size. In such cases, no signature-documentation sequence can be inserted and this work is not relevant for such molecules. For the rest of the paper, we assume that we are guaranteed to find a spot within the molecule where the signature-documentation sequence can be inserted. If not, the process exits.

Once the signature-documentation sequence is inserted, the Signer returns another sequence in the form of a GENBANK file which contains the original plasmid sequence combined with the signature and documentation sequence. Alice then creates a new plasmid (probably by getting the service from a gene synthesis company) corresponding to this signature and documentation sequence. Alice communicates about the original plasmid by using the plasmid ID used to identify the plasmid in the signature and claims ownership of it using her ORCID. When requested she sends the signed plasmid to the User through some physical means (In the biology domain, plasmids are shared. for example, as bio-solutions in test tubes).

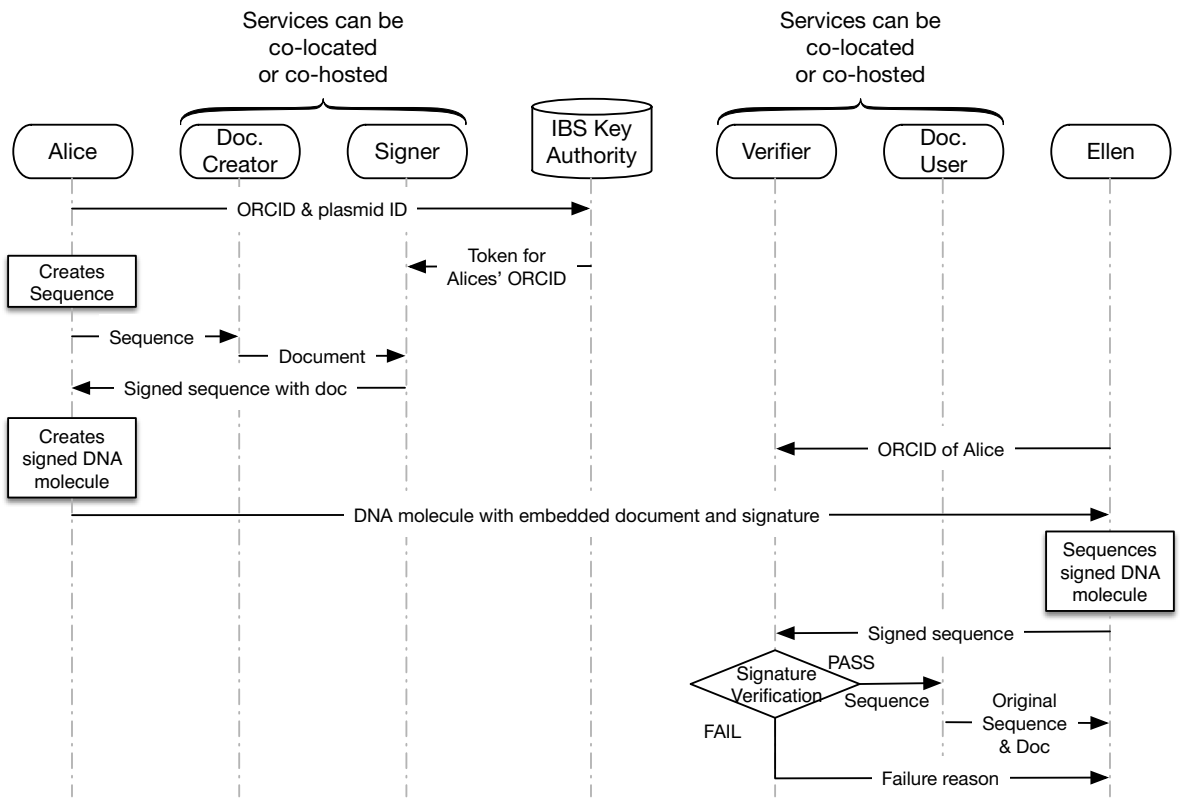


Fig. 1. DNA sign-share-validate workflow

Ellen gets the plasmid from some source (may or may not be directly from Alice). 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 this sequence of the signed plasmid to the Verifier to verify the plasmid signature. This uploaded file is a FASTA file. The signature validation service in

the Verifier identifies the signature inserted between the two signature tags. The Verifier will proceed with the signature validation as discussed in Section 4. If the signature is correct, indicated by a PASS response from the Verifier, Ellen will know that the plasmid was signed by Alice and that the physical sequence of the received plasmid corresponds exactly to Alice's design. Also after successful validation, the Verifier sends the genbank file containing the descriptions and signature that Alice embedded within the plasmid to the Doc user, which reveals the original sequence and the corresponding sequence (as allowed by Alice).

Alternatively, the validation service might have determined that the signature was invalid. 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 may have maliciously changed. One could also not rule out the possibility of spontaneous mutations or a 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 as discussed in Section 5.

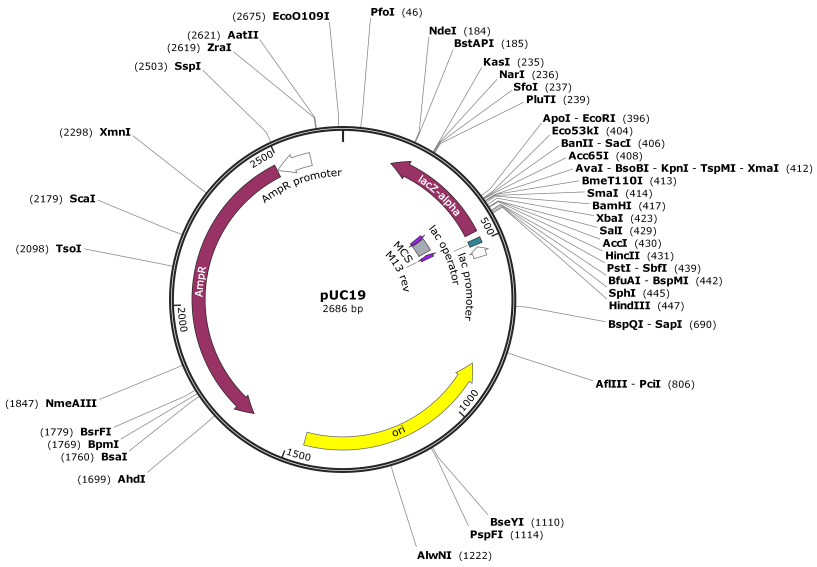
For the curious reader we show in Figure 2 the two sequenced versions of the pUC19 plasmid that we used in our work, both before the plasmid being signed (figure 2(a)) and after (figure 2(b)). The figures were created via the SnapGene editor that used as input the sequenced versions of both the un-signed and signed molecules. Note in Figure 2(b), the increased size of the DNA molecule because of the embedded signature.

4. DNA Signature Generation and Verification Procedure

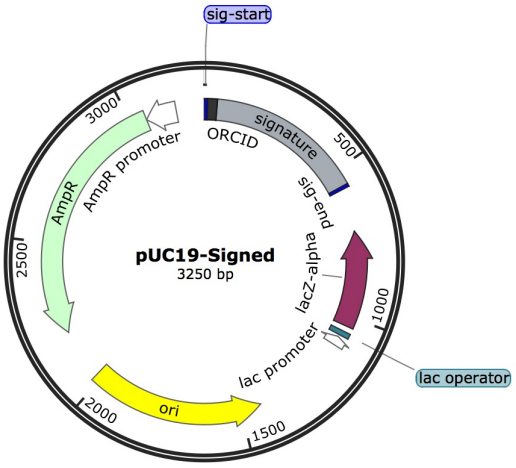
In our DNA sign-share-validate workflow, there are three players: (i) The DNA signer will create the DNA signature and sign a DNA sequence. (ii) The verifier will use the signature to verify whether the received DNA sequence was sent by the appropriate sender and was unchanged after signing. (iii) A central authority, which is trusted, provide the signer with an encrypted token that is associated with the signer's identity. The token contains the signer's private key.

Trust 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.

Use of error correction in DNA signature: In the digital domain, the digital signature on a message can be used to detect integrity violations. If a violation is detected, the sender can always re-transmit the signed message without incurring much extra cost. However, in the DNA world, we are primarily shipping physical DNA samples. This implies that if a DNA signature identifies that there is an error in the signature validation, then the sample needs to be physically transported and/or synthesized again. This incurs significant cost. DNA mutation is a very natural and common phenomenon. Thus, there is a good likelihood that signature validation will fail. Moreover, associated with the problem of mutation lies the problem of sequencing. When the DNA is processed by an automated DNA sequencer, the output is not always one hundred percent correct. It is dependent on the depth of sequencing, and increased sequencing depth means higher costs. Sequencing a small plasmid to sufficient depth is relatively inexpensive, but for larger sequences, sequencing errors can be an issue. In order to overcome these limitations, we



(a) Unsigned pUC19 (annotated)



(b) Signed pUC19 showing embedded signature

Fig. 2. View of sequenced pUC19 plasmid in SnapGene editor before and after signing

use block-based error correction codes, such as a Reed-Solomon code [11], together with signatures. The presence of error correction codes helps the receiver to locate a limited number of errors (as set by the signer) in the sequenced DNA as well as correct them. The position of the errors and the corrected values are conveyed to the verifier. The verifier can then decide if the errors are in any valuable feature of the DNA 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 DNA, the verifier can disregard the error and continue to work with it. Generally, there are three types of mutations that occur within a DNA - substitution, insertion and deletion. The error-correction code that is used here only deals with substitution errors.

Algorithm 1: DNA Signature Algorithm Accommodating Cyclic Shifts, Reverse Complement and Mutating Tags

Input: The GenBank (.gb) file: file, ORCID: a 16 digit number in xxxx-xxxx-xxxx-xxxx format, Plasmid ID: a 6 digit number, Location of signature placement: number, Error tolerance limit: number (can be 0 meaning no error tolerance)

Output: Signed GenBank (.gb) and FASTA (.fa) file: file

1 Input checks e.g. correct file extension, ORCID format, integers etc.
2 Parse GenBank file. Split content and sequence based on keyword ORIGIN. Parse content to get the list of feature locations.
3 **if** *Location of signature placement NOT within a feature* **then**
4 Make the position as start of the sequence and wrap everything before the location to the end. If position is 0 or length of sequence - no wrap is needed.
5 Generate hash (SHA-256) of this sequence.
6 Generate signature on the hash.
7 Convert the signature bytes,ORCID and Plasmid_ID to ACGT sequence. Create the following string by concatenating parts :
8 BESN+ORCID+Plasmid_ID+SIN+EDSN
9 **if** *error tolerance NOT 0* **then**
10 Append MSG (shifted sequence) before BESN+ORCID+PLASMID_ID+SIN+EDSN.
11 Pass SEQUENCE+BESN+ORCID+PLASMID_ID+SIN+EDSN to Reed-Solomon Encoder. (Here SEQUENCE is the shifted msg. Can be any shifts e.g. QUENCESE)
12 Convert the parity bytes to ACGT. (call this ECC)
13 Signature_Sequence = BESN+ORCID+PLASMID_ID+SIN+ECC+EDSN.
14 **else**
15 Signature_Sequence = BESN+ORCID+PLASMID_ID+SIN+EDSN.
16 **if** *signature placement location is start of the original sequence* **then**
17 Final_Sequence = SEQUENCE+Signature_Sequence
18 **else if** *signature placement location is end of the original sequence* **then**
19 Final_Sequence = Signature_Sequence+SEQUENCE
20 **else**
21 part1 = prefix of SEQUENCE of length $n-1$ (where signature is to be placed at location n
22 part2 = suffix of SEQUENCE of length $len(SEQUENCE)-n+1$
23 Final_Sequence = part1+Signature_Sequence+part2
24 Write the Final_Sequence to a new GenBank file and FASTA file.
25 **else**
26 Alert user about collision. Allow user to input new location. Go to step 3 with new location.

We now describe our new DNA signature scheme. The steps are shown in Algorithm 1 (for signing) and Algorithm 2 (for verification). To avoid confusion we use the following conventions. The term *sample* is used to indicate the physical DNA molecule. The term *sequence* is used to signify the digital counterpart of a DNA molecule. This is generated by sequencing a sample in a DNA sequencer. The raw sequence (output of sequencing) is stored in a FASTA file. The annotated sequence is stored in a GenBank file. The signer creates a physical DNA sample from the signed sequence and sends the sample (only) to the verifier. The verifier sequences this sample to get another sequence that is then verified.

For ease of understanding, we denote the sequence to be signed by the string SEQUENCE, the signature by SIN, the begin and end tags as BESN and EDSN and the error correction code as ECC. Each of these strings is really a sequence of bases that can be synthesized into a physical DNA molecule and embedded in the sample. Any location reference in SEQUENCE for subsequence discussion is specific to the location within the sequence. For instance, location 3 in the string contains character Q. However, in the real sequence, the subsequence denoted by Q may occur in position 350 (for example) depending on how many bases constitute S and E.

Signature generation: The signature generation procedure begins by scanning the GenBank file for the keyword ORIGIN and locating the actual DNA sequence. Let there exists a feature from location 1 to 3 in the sequence, which corresponds to SEQ. Next, the location of the signature placement specified by the signer is checked. If the location collides with a feature, the user is alerted to change the location. In our example, if the user had provided 2, the algorithm 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 Q, it will be allowed. Next, the ORCID and Plasmid ID (which are integers) are converted to the corresponding A C G T 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. if ORCID is 0000-0001-4578-9987, the converted sequence will not have a long run of a single base. Long runs of a single nucleotide can result in errors during sequencing. Let the converted ORCID and Plasmid ID sequences be ORCID and PID respectively.

To account for the problem of placing the signature within the sequence mentioned earlier in section 2, the signature is generated on the hash of a tweaked version of the sequence. We left rotate a copy of the sequence by $n - 1$ where n is the location within the sequence where the signature needs to be placed. For this example, the sender wants to place the signature after Q. The sequence will be shifted as – UENCESEQ. The signature is generated on the hash of the left rotated sequence UENCESEQ. The signature bits are then converted to A C G T sequence. Let this signature sequence be SIN. Let the start tag be BESN and end tag be EDSN. The signature sequence is concatenated with ORCID and PID and then placed between the start and end tags as BESN ORCID PID SIN EDSN. This entire string is then placed at the position specified by the user. We chose 4 in our example. Hence, the signed sequence looks like - SEQ BESN ORCID PID SIN EDSN UENCE.

Next, this sequence is passed into the error correction encoder. According to the number of tolerable errors specified by the user, the error correcting parity bits are generated. These parity bits are then converted to some A C G T sequence. Let this sequence be ECC. When the encoder output is generated, the sequence would look like – SEQ BESN ORCID PID SIN EDSN UENCE ECC. Next, the ECC is separated and is placed before the signature and end tag. So the final output sequence is - SEQ BESN ORCID PID SIN ECC EDSN 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 instance, if we put 2 as our error tolerance

limit, then any 2 errors within the string SEQ BESN ORCID PID SIN ECC EDSN UENCE can be tolerated. If there is 1 error in SEQ and 1 error in SIN, or 2 errors in SIN, or 1 error in SIN and 1 error in ECC, these can be corrected. But if there are more than two errors it cannot be corrected. The final output sequence - SEQ BESN ORCID PID SIN ECC EDSN 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 were features after location 4 in the original DNA, the locations of these features are also updated. This GenBank file is for reference of the sender. It is not required for signature verification and there is no need to share it with the receiver unless there are other reasons. The output sequence is now synthesized into the signed DNA sample.

Signature verification: The signature verification procedure is described below in Algorithm 2.

The receiver sequences the shared DNA using an automated DNA sequencer. The sequence in the FASTA file might not be the in the same order when the sender signed it. That is, after sequencing the shared DNA, the FASTA file may look like - ORCID PID SIN ECC EDSN UENCE SEQ BESN which is a cyclic permutation of the sender's sequence.

The first step in the verification procedure is to extract the BESN and EDSN tags. If they are not mutated they are retrieved directly. If the tags cannot be located directly, we use Algorithm 3 to retrieve their closest matches and use them as BESN and EDSN tags. We defer the discussion on Algorithm 3 to section 2.2. The verification step now will concatenate the FASTA sequence - ORCID PID SIN ECC EDSN UENCE SEQ BESN + ORCID PID SIN ECC EDSN UENCE SEQ BESN.

Now, it looks for 2 BESN tags and extracts the content between them. 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 EDSN tag is retrieved, this is the error correction sequence. Finally, the substring between EDSN and BESN is the message for signature verification.

Until this point, we have retrieved UENCESEQ, ORCID, PID, SIN, and ECC. The UENCESEQ, ORCID and SIN is used for signature verification. With our previous signature generation method, since the message signed by the sender was SEQUENCE and the message retrieved by the verifier is ENCESEQ the hashes will be different and the validation would fail. With the new procedure, we can see that the although the sender's file contained the sequence SEQUENCE, the signature was actually generated on the shifted UENCESEQ. Due to this shift, the retrieved sequence and the sender's sequence will always be the same under any rotations. We have shifted the message of the sender to make the signature placement at the start of the message. We call this new generation scheme as force shift 0.

If the FASTA file contains the reverse complement of the sender's DNA sequence, the entire FASTA file is reverse complemented and then we look for the BESN and EDSN tags. If there is a match, we arrive at the conclusion that the FASTA file contains the reverse complement. Then we start the same verification steps on the reverse complemented FASTA sequence.

5. Allowing Mutations in Start and End Tags

The approximate matching technique, shown in Algorithm 3, breaks the entire string in which we are looking for the result into substrings of the length of the 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. A 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

Algorithm 2: New signature verification procedure

Input: A FASTA file generated from sequencing the DNA sample received
Output: Prompt - Signature Valid or Invalid.

```
1  Input checks: file extension and only ACGT content.
2  Parse FASTA file and create reverse complement of the file
3  Use Algorithm 3 to get the BESN and EDSN tags.
4  if (file contains BESN or EDSN) OR (reverse contains BESN or EDSN) then
5      if file contains BESN or EDSN then
6          Create content string by appending FASTA file content thrice.
7          Get the sequence between two BESN tags. Create the following parts by counting:
             ORCID = first 32 chars; PLASMID_ID = next 12 chars; SIN = next 512 chars; ECC =
             chars between SIN and END (may be empty); MSG = chars from END to end of string.
8      else
           /* When input FASTA file is in reverse complement form.   */
9          Create content string by appending reverse complement of FASTA file content thrice.
10         Same as Step 6. i.e. get the parts from reverse complement.
11     Generate hash (SHA-256) of MSG
12     Invoke signature verification
13     if signature is valid then
14         Alert user about success.
15     else
16         Alert user about failure and start error correction procedure.
17         if ECC length is 0 then
18             Alert user there is no ECC and correction not possible.
19         else
20             Create the following string from the parts:
                 SEQUENCE+BESN+ORCID+PID+EDSN+ECC and send to Reed-Solomon
                 decoder.
21             if decoder outputs null or same as input then
22                 Alert user errors are more than tolerable limit.
23             else
24                 Get the corrected parts and re-invoke verification.
25                 if re-verification is success then
26                     Alert user that verification succeeded after error correction. Compare the parts
                         before and after error correction and display the errors.
27                 else
28                     Alert user that verification failed even after successful correction.
29 else
30     Alert user that BESN and EDSN tags are not present.
```

be a case that there are multiple close matches which means that there are multiple starts (or end) tags. In those cases, we use the end tags (or start tags respectively) 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 either start or end tags.** - In this case, we can find the exact 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** `BESN` **tag only.** - In this case, the `EDSN` tag is found directly. The algorithm looks for the closest match to `BESN`. If there is a single match with the highest score, then we can be quite certain that the `BESN` tag has been located correctly. However, there can be multiple matches with close scores, i.e., there is no single stand out high score. In that case, we use the `EDSN` 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. Hence we choose only those potential `BESN` tags which are at distance of 556 base pairs/characters or more away from the `EDSN` 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** `EDSN` **tag only.** - In this case, the `BESN` tag is found directly. The tool looks for the closest match to `EDSN`. As in case 2, if there is a single match with the highest score then we can be quite certain that the `EDSN` tag has been located correctly. For multiple matches with close scores, we use the same logic as described in case 2 above, using the distance between the `BESN` and `EDSN` tags to be more than or equal to 556 base pairs.
(4) **Case 4: Mutation in both** `BESN` **and** `EDSN` **tags.** - In this case, we try to locate the closest matches for both tags. If there is a single match with the 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 potential `BESN` and `EDSN` tags, we employ the length counting criteria for each `BESN` and `EDSN` tag pair possible from the obtained results and narrow down the results.

5.1. Experimentally Determining Most Suitable Distance Measures for String Matching

Various techniques exist to handle matching of similar strings. These methods measure the distance between strings using a distance equation. One of the most important works in this field is the Levenshtein distance [13]. Other notable algorithms are Damerau-Levenshtein[13–15], Optimal String Alignment variant of Damerau-Levenshtein (sometimes called the restricted edit distance) [15], Jaro-Winkler edit distance [16], and Jaccard index [17, 18].

We used all these five algorithms for the approximate start and end tag matching. One of the reasons for using all of the above was that we wanted to find out which would be most suited to the DNA domain. For testing, the FASTA file is taken as input and the start and end tag within the FASTA file are manually changed. Next, we search for the location of the defined start and end tags within the mutated FASTA file. The results for each algorithm are summarized on a case by case basis in Figure 3. As can be seen from the figure, 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.

Another important consideration in algorithm selection was speed. While an algorithm may be perfectly accurate in its selection of the closest match to a string it would not help much in practice if the

Algorithm 3: Approximate matching of tags

Input: Content of FASTA file: String
Output: BESN and EDSN tags: 2 Strings

1 `begin` = ACGCTTCGCA; `end` = GTATCCTATG `/* hardcoded */`
2 `revcomp` = reverse complement of input string
3 **if** *input contains (*`begin` *and* `end`*)* **then**
4 BESN = `begin`; EDSN = `end`
5 **else if** *input contains* `end` *and NOT* `begin` **then**
6 EDSN = `end`; Split input into substrings of length 10
7 **foreach** *substring* **do**
8 Calculate score with `begin`; Store each substring and score. Sort by score.
9 **if** *single highest score* **then**
10 BESN = highest score substring
11 **else if** *multiple high scores* **then**
12 Calculate distance between each substring to `end`.
13 BESN = substring where distance > 556
14 **if** *multiple pairs with distance > 556.* **then**
15 Alert user about failure to extract tags. Exit
16 **else if** *input contains* `begin` *and NOT* `end` **then**
17 BESN = `begin`; Split input into substrings of length 10
18 Same as step 7 and 8. Replace `begin` with `end`
19 Same as step 9. Set EDSN = highest score substring as in step 10.
20 Same as step 11. Replace `end` with `begin` in step 12. Set EDSN as in step 13.
21 Same as step 14 and 15.
22 **else if** *input does NOT contain* `begin` *and* `end` **then**
23 Split input into substrings of length 10
24 **foreach** *substring* **do**
25 Calculate score with both `begin and end`;
26 Store each substring and score for both. Sort by score.
27 **if** *single highest score in both* **then**
28 BESN = highest score substring;EDSN = highest score substring;
29 **else if** *multiple high scores in both* **then**
30 Calculate distance between each pair of substrings. Set BESN and EDSN where distance > 556.
31 **if** *multiple pairs with distance > 556.* **then**
32 Alert user about failure to extract tags. Exit
33 Repeat the same four conditions as in step 3, 5, 16 and 22 with `revcomp` instead of input. e.g. `revcomp` contains (`begin` and `end`)
34 return BESN and EDSN

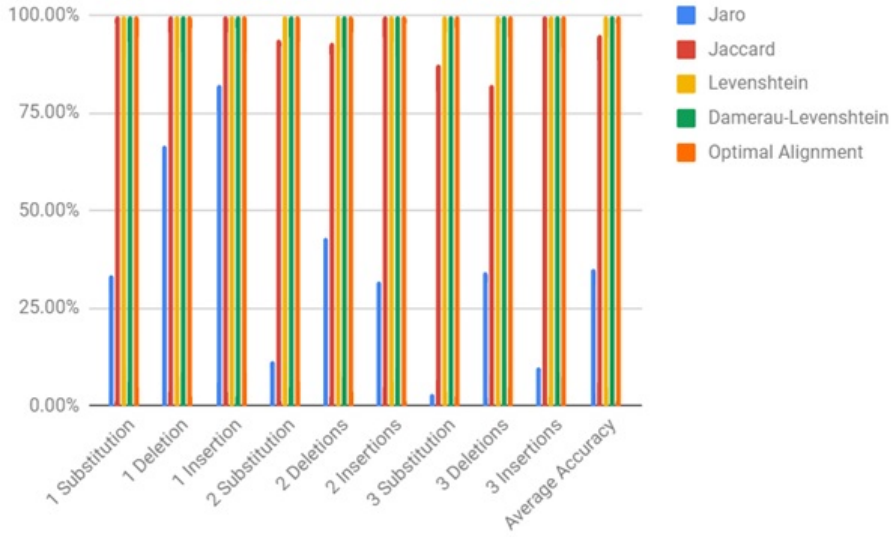


Fig. 3. Accuracy of algorithms per case as a percentage.

algorithm has an unacceptably long run time. Towards this end, the execution time of the algorithms were also compared. To accomplish this each method was used to compare a series of one million random strings of a set length. A graph of the time in milliseconds (ms) for each algorithm is given in Figure 4.

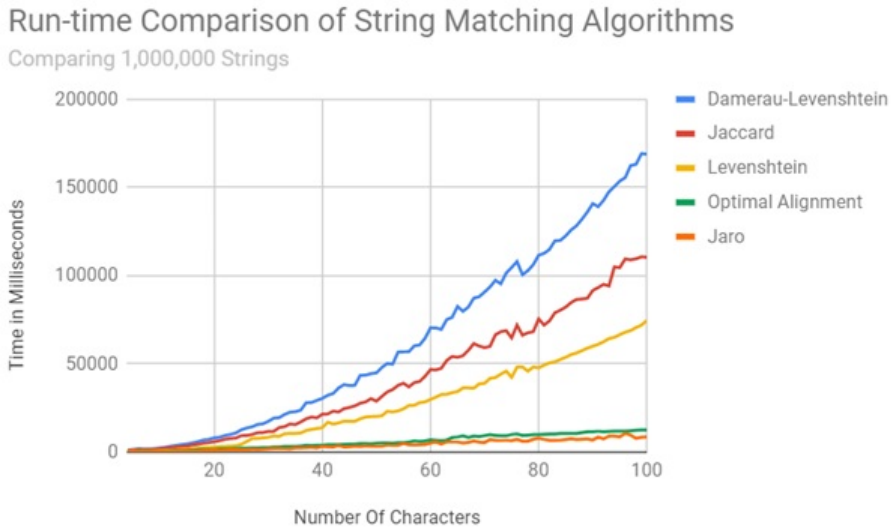


Fig. 4. Runtime analysis of various algorithms in milliseconds

As can be seen from Figure 4, the Jaro-Winkler and Optimal String Alignment algorithms were the quickest, each growing at very slow rates with Jaro-Winkler being slightly faster overall. Taking both

of these factors into consideration, we chose the Optimal String Alignment variant of the Damerau-Levenshtein algorithm [15] as our preferred method for string matching.

6. New Identity-based Signature Scheme with Shorter Signature Size

There are several identity-based digital signature schemes using pairings. Some of the notable schemes are: *Sakai-Kasahara* [19], *Sakai-Ohgishi-Kasahara* [20], *Paterson* [21], *Cha-Cheon* [22], and *Xun Yi* [23]. The *Sakai-Kasahara* scheme described two types of identity-based signatures. One is El-Gamal type and the other is Schnorr type. To identify the most appropriate scheme we first implemented all the above schemes using the Java Pairing Based Cryptography library (jPBC) [24]. We then investigated the signature lengths based on different types of curves that can be used. The time to generate and validate a signature depends on the type of the curve used. We evaluated 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.

Based on the signature size and the computation cost of signature generation and verification, we identified the best scheme to be the Sakai-Kasahara Schnorr type. We now describe the Sakai-Kasahara Schnorr type identity-based signature scheme. It 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 performed by 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:

$$e = e_n(C_A, Z_A) = e_n(C_A, r \cdot g_2) = e_n(C_A, g_2)^r$$

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

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

The signature is a tuple (h, S) where h is the result of a hash function and is dependent on the choice of the hash function. If h is SHA-1, then length is 20 bytes, if h is 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. There are six types of curves in the jPBC library namely – a, a1, d, e, f, and g. The different types of curves and their parameters are provided in the library as "properties" files. Table 1 summarizes the comparison of the signature length using the different curves.

Table 1
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

Based on the signature size, 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. Keeping in view 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. Table 2 summarizes the time to sign and verify using the different types of curves.

From the speed perspective, the a type curve is the fastest for 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. It is, therefore, a matter of priority - signature size over speed. If we need to sign and verify a lot of messages and 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. Also, the f type curve offers the best security among the three as its embedding degree is higher. Using this Sakai-Kasahara scheme we have reduced the signature size from 512 base pairs to

Table 2
Average time taken to sign and verify for different types of curves for the Sakai-Kasahara scheme.

Curve Name	Average time to sign (ms)	Average time to verify (ms)
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

288 base pairs. The only thing it affects in our earlier algorithms is determination of `BESN` and `EDSN` in Section 5 when these tags mutate and we need to rely on counting base pairs to locate those tags.

Security of scheme. : Since we use well-known signature schemes that assume that no polynomial-time adversary can forge a genuine signature without knowing the secret used to sign, it trivially follows that our scheme is also secure.

7. Compressing the Signature to Reduce Molecule Size

The Sakai-Kasahara scheme mentioned above generates the shortest signature size among the known identity-based signature schemes. The signature is a tuple (h, S), where h is a hash function and S is an element in the group G_1 of the curve.

Now, the elements in the group G_1 are points on the elliptic curve. They have two subcomponents the X-coordinate and the Y-coordinate which are packed together. In the JPBC library, the elements are assigned and computed using the class “Element". The fields are generated from the pairing curves i.e. the properties files. When we print the “Element" from the group G_1, we can observe that it contains the two coordinates. The following syntax is used to generate a random element in G_1 -

Pairing pairing = PairingFactory.getPairing(“f.properties");
PairingFactory.getInstance().setUsePBCWhenPossible(true);
Field G1 = pairing.getG1();
Element g1 = G1.newRandomElement().getImmutable();
System.out.println("g1 - "+g1);
System.out.println("G1 bytes = "+g1.toBytes().length);
The code produces the following output -
g1 - 185005020540567871876248888433515090492299777 4,
559997702586520753280126012454714158056438703 92,0
G1 bytes = 40

In order to extract the X and Y coordinate separately, the Element class needs to be converted to a byte array. In that array, half of the array contains the X coordinate the other half contains the Y coordinate. We can then separate the two arrays and convert them to the BigInteger class.

Now every elliptic curve and consequently every pairing curve is defined by an equation of the form $y^2 = x^3 + ax + b$. Hence, the Y coordinate can be calculated from the X coordinate. We do not need to

store the Y coordinate in the signature. The Y coordinate can be calculated from the X coordinate during verification. But it has to be noted that when we plug in the value of X, there will be two solutions for the Y coordinate and the square root is modulo prime. For this reason, the way to distinguish between which value of Y to keep, there needs to be some more data about the Y coordinate. The convention to do this is to add one extra byte that will denote if the Y coordinate is odd or even. When we discard the Y coordinate we can do a modulo 2 and if Y is odd, we append the byte 02 before X value. If Y is odd we append 03. Using this point compression technique, it can be observed that a signature which contains an element of G_1 and has a size of $2n$ bytes(assuming each X and Y are n bytes), can be compressed to a size of $(n + 1)$ bytes.

This technique can only be applied to signature schemes where the signature contains an element in G_1. We are not sure if the elements of G_2 and G_T can be compressed. So for all of the schemes that are implemented already and the schemes that we described above can utilize this compression. Also, the Sakai-Kasahara Schnorr analog scheme contains a hash value in one of the tuple. Recall that the Sakai-Kasahara signature was (h, S), where h is a hash function like SHA-256. Along with the point compression, the hash value can also be shortened using the techniques that are used to generate Ethereum[25] or Bitcoin [26] addresses. In Ethereum, the public key is hashed using keccak 256 hash algorithm. Then instead of taking the entire 32 bytes (64 bytes in hex representation), the Ethereum blockchain takes only the last 20 bytes (40 bytes in hex) and generates the wallet address. Each wallet address is 40 hex bytes but the hash generates 64 hex bytes. There is also another way of doing the same hash compression. 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. But this method will not be efficient for curves which have large orders like the elements in the type a1 curve are 128 bytes.

In order to recover the Y coordinate during verification we need to do point decompression i.e. retrieve Y using X and the extra byte. The X value is plugged in the equation and the square root modulo prime is calculated. The two values of Y are obtained. Then using the extra byte we know if we should keep the odd Y or the even Y. Note that every curve has a different equation so the decompression method needs to consider the curve and the coefficients a and b. Also, there exists a very easy way to compute the square root modulo prime when the prime modulo 4 equals 3. If prime $\equiv 3 \bmod 4$ -

$Ycoordinate = (Y^2)^{\frac{prime+1}{4}} \ mod \ prime.$

However if prime $\equiv 1 \bmod 4$, then there is no one line way to calculate the square root. The f type curve will be faster in this approach as the prime in that curve parameters is 3 mod 4. The code for point decompression is provided in the Appendix 3. This point compression method is not present in the JPBC library by default.

Using these compression methods the signature size which was (32, 40) bytes totalling 72 bytes can be reduced to (20, 21) totalling 41 bytes. Hence the size can be reduced from 288 base pairs (72 * 4) to 164 base pairs (41 * 4).

8. Self Documenting Plasmids

In the context of DNA sharing, the receiver gets a physical DNA molecule, which he/she can pass through an automated DNA sequencer and obtain the sequences present within the molecule. The se-

quence is in the form of a FASTA file which can be used to verify the signature within the molecule. The FASTA file contains just the raw sequences refer to fig 5.

```
>pUC19.gb(2686bp)
tcgcgcgtttcggtgatgacggtgaaaacctctgacacatgcagctcccggagacggtcacagcttgtctgtaagcggatgccgggagcagacaagcccgtcagggcgcgtcag
cgggtgttggcgggtgtcggggctggcttaactatgcggcatcagagcagattgtactgagagtgcaccatatgcggtgtgaaataccgcacagatgcgtaaggagaaaataacc
gcatcaggcgccattcgccattcaggctgcgcaactgttgggaagggcgatcggtgcgggcctcttcgctattacgccagctggcgaaaggggatgtgctgcaaggcgattaa
gttgggtaacgccagggttttcccagtcacgacgttgtaaaacgacggccagtgaattcgagctcggtacccggggatcctctagagtcgacctgcaggcatgcaagcttggcg
taatcatggtcatagctgtttcctgtgtgaaattgttatccgctcacaattccacacaacatacgagccggaagcataaagtgtaaagcctggggtgcctaatgagtgagctaa
ctcacattaattgcgttgcgctcactgcccgctttccagtcgggaaacctgtcgtgccagctgcattaatgaatcggccaacgcgcggggagaggcggtttgcgtattgggcgc
tcttccgcttcctcgctcactgactcgctgcgctcggtcgttcggctgcggcgagcggtatcagctcactcaaaggcggtaatacggttatccacagaatcaggggataacgca
ggaaagaacatgtgagcaaaaggccagcaaaaggccaggaaccgtaaaaggccgcgttgctggcgtttttccataggctccgcccccctgacgagcatcacaaaaatcgacgc
tcaagtcagaggtggcgaaacccgacaggactataaagataccaggcgtttcccccctggaagctccctcgtgcgctctcctgttccgaccctgccgcttaccggatacctgtcc
gcctttctcccttcgggaagcgtggcgctttctcatagctcacgctgtaggtatctcagttcggtgtaggtcgttcgctccaagctgggctgtgtgcacgaaccccccgttcag
cccgaccgctgcgccttatccggtaactatcgtcttgagtccaacccggtaagacacgacttatcgccactggcagcagccactggtaacaggattagcagagcgaggtatgta
ggcggtgctacagagttcttgaagtggtggcctaactacggctacactagaagaacagtatttggtatctgcgctctgctgaagccagttaccttcggaaaaagagttggtagc
tcttgatccggcaaacaaaccaccgctggtagcggtggtttttttgtttgcaagcagcagattacgcgcagaaaaaaaggatctcaagaagatcctttgatcttttctacgggg
tctgacgctcagtggaacgaaaactcacgttaagggattttggtcatgagattatcaaaaggatcttcacctagatccttttaaattaaaaatgaagttttaaatcaatctaa
agtatatatgagtaaacttggtctgacagttaccaatgcttaatcagtgaggcacctatctcagcgatctgtctatttcgttcatccatagttgcctgactccccgtcgtgtag
ataactacgatacgggagggcttaccatctggccccagtgctgcaatgataccgcgagacccacgctcaccggctccagatttatcagcaataaaccagccagccggaagggcc
gagcgcagaagtggtcctgcaactttatccgcctccatccagtctattaattgttgccgggaagctagagtaagtagttcgccagttaatagtttgcgcaacgttgttgccatt
gctacaggcatcgtggtgtcacgctcgtcgtttggtatggcttcattcagctccggttcccaacgatcaaggcgagttacatgatcccccatgttgtgcaaaaaagcggttagc
tccttcggtcctccgatcgttgtcagaagtaagttggccgcagtgttatcactcatggttatggcagcactgcataattctcttactgtcatgccatccgtaagatgcttttct
gtgactggtgagtactcaaccaagtcattctgagaatagtgtatgcggcgaccgagttgctcttgcccggcgtcaatacgggataataccgcgccacatagcagaactttaaaa
gtgctcatcattggaaaacgttcttcggggcgaaaactctcaaggatcttaccgctgttgagatccagttcgatgtaacccactcgtgcacccaactgatcttcagcatctttt
actttcaccagcgtttctgggtgagcaaaaacaggaaggcaaaatgccgcaaaaaagggaataagggcgacacggaaatgttgaatactcatactcttcctttttcaatattat
tgaagcatttatcagggttattgtctcatgagcggatacatatttgaatgtatttagaaaaataaacaaatagggggttccgcgcacatttccccgaaaagtgccacctgacgtc
taagaaaccattattatcatgacattaacctataaaaataggcgtatcacgaggccctttcgtc
```

Fig. 5. Sample fasta (.fasta) file

However, the receiver has no description about the molecule. Sequence manipulation software such as SnapGene can be used to convert a GENBANK file to a FASTA 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 sample, the creator would need to share the GENBANK file containing the manually added annotations.

Hence in order to establish a strong tie between the shared GENBANK file and the shared physical sample, the use of dual signature was proposed in [2]. Using a dual signature, it can be verified that the shared GENBANK file containing the descriptions is indeed meant to describe the shared sample and not for any other sample. But having a large set of GENBANK files and a large set of physical samples, it will be difficult to match a sample with its related GENBANK description, i.e. all the samples needs to be tested for the dual signature match.

A sample GENBANK file is shown below in two parts Fig 6 and Fig 7. This file corresponds to the FASTA file shown above. The GENBANK file is displayed in two parts due to its length.

The keyword “ORIGIN" demarcates the annotations / descriptions about the molecule and the actual sequence. The keyword “FEATURES" describes the location of the different features that this sample contains and their location within the sequence. Also it denotes the total number of base pairs in the sample under “source".

As we are already embedding a digital signature within a molecule, it is also possible to embed the descriptions within it as well. If the descriptions about the molecule can be embedded within itself then

```
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"
     primer_bind     complement(465..481)
                     /label=M13 rev
```

Fig. 6. Sample GENBANK (.gb) file - part 1

there is no need to share the GENBANK file and the physical sample is self documented. The descriptions are text and it can be easily converted to bytes and from bytes to ACGT. But there is a limit to how much information can be put inside a molecule such that it is stable and retains its original functionality. For this criteria, we explored lossless text compression techniques and encode the compressed bytes as ACGT instead of the original text bytes. While unpacking the original text can be recovered

```
     CDS             complement(1626..2486)
                     /codon_start=1
                     /gene="bla"
                     /product="beta-lactamase"
                     /label=AmpR
                     /note="confers resistance to ampicillin, carbenicillin, and
                     related antibiotics"
                     /translation="MSIQHFRVALIPFFAAFCLPVFAHPETLVKVKDAEDQLGARVGYI
                     ELDLNSGKILESFRPEERFPMMSTFKVLLCGAVLSRIDAGQEQLGRRIHYSQNDLVEYS
                     PVTEKHLTDGMTVRELCSAAITMSDNTAANLLLTTIGGPKELTAFLHNMGDHVTRLDRW
                     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 acccggggat
      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 gccccctga cgagcatcac
      901 aaaaatcgac gctcaagtca gaggtggcga aacccgacag gactataaag ataccaggcg
      961 tttccccctg 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 ctagaagaac agtatttggt
     1321 atctgcgctc tgctgaagcc agttaccttc ggaaaaagag ttggtagctc ttgatccggc
```

Fig. 7. Sample GENBANK (.gb) file - part 2

and the exact same GENBANK file can be reproduced. Some of the compression algorithms we explored were zip, bzip, lzma, lz4, snappy, and deflate. Our main focus was compression size rather than time. We observed that the best algorithm in terms of compression size was deflate with the parameter BEST_COMPRESSION. (Deflater compressedtext = new Deflater(BEST_COMPRESSION);)

For the sample GENBANK file, the length of the descriptions (i.e. all text before the keyword “ORIGIN" in Figures 6 and 7) is 3675 bytes. After compression it shrinks to 1493 bytes. Consequently it is a better choice to encode the compressed annotations instead of the actual text. At the time of inserting the

annotations, the same challenges we faced for signature insertion and verification because of the circular and double stranded nature of DNA, also appears.

To explain the process, we consider a small example. Let the sample have the following sequence - AAA CCC GGG TTT. Also, let us consider that this sample has 2 features - AAA is feature X and GGG is feature Y. The GENBANK file for this sample will look somewhat the following:

```
LOCUS...
....
FEATURES
     source          1 .. 12
     featureX        1 .. 3
     featureY        7 .. 9
ORIGIN
1 aaacccgggt tt
```

Let us assume that when the annotations are compressed and encoded into sequence, it is ACGCTC. As discussed before we need two identifiers. Let GCG and ATA be the identifiers for this example. This will be inserted into the original content. The location is chosen by the user and it cannot collide with an existing feature. Let us assume that the chosen location is 4 i.e. after AAA. The sequence now becomes AAA GCG ACGCTC ATA CCC GGG TTT.

When this is shared with the receiver and sequenced it might come out as ATA CCC GGG TTT AAA GCG ACGCTC. The identifiers will aid in locating the annotations. We can identify ACGCTC convert to bytes and decompress to get the actual text. Recall the annotations had featureX 1..3 and featureY 7..9. But as per this sequence AAA is not at 1..3 as also GGG is not at 7..9. The source can always be overwritten with the new count. But the problem is with locating the features and generating the GENBANK file with the actual location of the features. We can permute the sequence such that it always starts with the identifier after locating i.e. ATA CCC GGG TTT AAA GCG ACGCTC can be rewritten as GCG ACGCTC ATA CCC GGG TTT AAA. In this way, the original content which was shared can be located. But still there is a mismatch with the feature location. In the original AAA was in position 1..3, here it is CCC. As we mentioned, before this would not happen if the sender selects the first position or the last position of the sequence as the annotation insertion point. But there is no guarantee that there will be no feature at the start or end. Hence in order to generalize this, we changed the procedure to annotation placement similar to the signature placement strategy discussed in section 4. To explain it briefly, before inserting the signature at the chosen location, the original sequence is shifted such that it becomes the dummy start. In the example, AAA CCC GGG TTT was the original sequence and 4 was the chosen location. The original sequence is shifted to CCC GGG TTT AAA and the new feature location are calculated based on this i.e the features are now updated as featureX 10..12 and featureY 4..6. Then this annotation is compressed and encoded to ACGT and then put into the location. Let this annotation sequence be CAGATA. Note that this is not the same as the annotations in the original file. Since we shifted the sequence internally, the compression sequence will not be the same as the previous example. The final sequence is therefore AAA GCG CAGATA ATA CCC GGG TTT, At the receiver's end, the result might come up as GGG TTT AAA GCG CAGATA ATA CCC. Like before we can look for the delimiters and then permute the sequence as GCG CAGATA ATA CCC GGG TTT AAA. But the decompressed text now will be expanded as featureX 10..12 and featureY 4..6. This matches with the annotations precisely if we consider the start of the sequence as after the end delimiter ATA. The final generated GENBANK will contain an offset of the length of start delimiter compressed annotation

length and the end delimiter length. So for this example, it will be updated as featureX - 10+12..12+12 and featureY - 4+12..6+12. The length of offset - *len*(GCGCAGATAATA) = 12. It can be verified that GCG CAGATA ATA CCC GGG TTT AAA contains the featureX which was AAA at position 22..24 and featureY which was GGG at position 16..18. The algorithms for generating a self documented plasmid and reading the plasmid to get back the annotations are described in Algorithm 4 and 5.

Algorithm 4: A self documenting plasmid generation algorithm

Input: The GenBank (.gb) file: file, Plasmid ID: a 6 digit number, Location of annotation placement: number, Error tolerance limit: number (hard coded as 2 errors)

Output: Annotated GenBank (.gb) and FASTA (.fa) file: file

1 Input checks e.g. correct file extension, Plasmid ID format, integers etc.
2 Parse GenBank file. Split content and sequence based on keyword ORIGIN. Parse content to get the list of feature locations.
3 **if** *Location of annotation placement NOT within a feature* **then**
4 Make the position as start of the sequence and wrap everything before the location to the end. If position is 0 or length of sequence - no wrap is needed.
5 Update the annotations by considering the placement location as 0 .
6 Compress the annotations.
7 Convert the compressed annotation bytes, Plasmid_ID to ACGT sequence. Create the following string by concatenating parts :
8 BESN+Plasmid_ID+ANNOTATION+EDSN
9 Append MSG (shifted sequence) after BESN+PLASMID_ID+ANNOTATION+EDSN.
10 Generate a checksum e.g. crc32 on the appended sequence BESN+PLASMID_ID+ANNOTATION+EDSN+MSG. Let this be CRC
11 Put the CRC sequence between ANNOTATION AND EDSN. BESN+PLASMID_ID+ANNOTATION+CRC+EDSN
12 . Pass BESN+PLASMID_ID+ANNOTATION+CRC+EDSN+MSG to Reed-Solomon Encoder.
13 Convert the parity bytes to ACGT. (call this ECC)
14 Annotated_Sequence = BESN+PLASMID_ID+ANNOTATION+CRC+ECC+EDSN.
15 **if** *annotation placement location is start of the original sequence* **then**
16 Final_Sequence = SEQUENCE+Annotated_Sequence
17 **else if** *annotation placement location is end of the original sequence* **then**
18 Final_Sequence = Annotated_Sequence+SEQUENCE
19 **else**
20 part1 = prefix of SEQUENCE of length $n-1$ (where annotation is to be placed at location n
21 part2 = suffix of SEQUENCE of length $len(SEQUENCE) - n + 1$
22 Final_Sequence = part1+Annotated_Sequence+part2
23 Write the Final_Sequence to a new GenBank file and FASTA file.
24 **else**
25 Alert user about collision. Allow user to input new location. Go to step 3 with new location.

8.1. Encrypting parts of the annotations for controlled dissemination

As mentioned earlier, the creator of the DNA molecule may not always be willing to share all information in the DNA document. One simple approach to do this is to encrypt some features of the annotations such that the receiver can only read parts of the annotations. Only the annotations that needs to be read by everyone will be kept in plaintext and encoded as mentioned above. For this approach, there can be several different ways each with its own advantages and disadvantages. Since we are already using identity-based signature, the obvious technique is to use its counterpart identity-based encryption. There are already existing schemes such as Boneh-Franklin IBE scheme [27], Sakai-Kasahara IBE scheme [19] etc., that can be used to accomplish this. However, note that all of the IBE schemes generate the encrypted message as a tuple. The size of the message is doubled. Unfortunately, in this domain we need to keep the message or annotations as short as possible. To what extent we can allow the expansion of a molecule without affecting properties is largely unknown. However, biologists believe that it depends on the size of the original DNA molecule where we would include the annotations. If that sample is large then it is very likely that IBE schemes can be used. One issue with IBE schemes is that it is meant to be decrypted by exactly one particular user. The user's ORCID which is used to encrypt the annotation will only be able to read the annotations. In cases where we would like to have a selected group of users read an annotation IBE schemes cannot be used.

Another possible method is by using symmetric key encryption like AES. One advantage is that the message size will not expand. But the sender and receiver need to agree on a shared key. This key can be sent to other selected users who will be allowed to read the annotations. But this also implies that a different key is needed for each DNA. Attribute-based encryption techniques is worth exploring in this regard. As of now we have not yet implemented these schemes and is left as a future work.

9. Conclusion and Future Work

In this work, we improve the previous DNA signing scheme [2] in several directions. First, we remove the need to share the genbank file by eliminating the requirement of alignment at the sample receiver's end. The new signature generation procedure is independent of where the signer wants to place the signature. Notwithstanding any cyclic shifts or reverse complements that the receiver may get during sequencing, the signature can still be verified. To account for DNA mutations, we use error correction codes in the signature protocol to correct errors within pre-specified tolerable limits. Our second improvement is a way to locate mutated tags using approximate string matching techniques. This allows us to overcome mutation in the identifying tags and hence we can correctly recover the error correction code. This was a major problem in previous scheme.

Our third improvement is the reduction of signature size. We used pairing based cryptography to improve the previous signature scheme which generated 512 base pair signature to the Sakai-Kasahara scheme which generates 288 base pair signature. Then we further compressed that to 164 base pairs. It will be worth exploring the lattice-based schemes to check if those can be used to generate a shorter signature size than 164 base pairs.

We also explored the possibility of encoding the documentation or annotation about the sample within the sample itself. For just the signature scheme we had removed the need to share any digital file along with the physical sample. But still the receiver after validating the correctness of the sample would have limited knowledge about the description of the sample and the features it has. Previously, we had to share

Algorithm 5: A self documented plasmid reading algorithm

Input: A FASTA file generated from sequencing the DNA sample received
Output: A GENBANK file with annotations and the sequence

```
1  Input checks: file extension and only ACGT content.
2  Parse FASTA file and create reverse complement of the file
3  Use Algorithm 3 to get the BESN and EDSN tags.
4  if (file contains BESN or EDSN) OR (reverse contains BESN or EDSN) then
5      if file contains BESN or EDSN then
6          Create content string by appending FASTA file content thrice.
7          Get the sequence between two BESN tags. Create the following parts by counting:
             PLASMID_ID = first 12 chars; ECC = 32 chars before EDSN; CRC = 32 chars before
             ECC; ANNOTATION = chars between PLASMID_ID and CRC; MSG = chars from
             EDSN to end of string.
8      else
           /* When input FASTA file is in reverse complement form.   */
9          Create content string by appending reverse complement of FASTA file content thrice.
10         Same as Step 6. i.e. get the parts from reverse complement.
11     Generate the following string BESN+PLASMID_ID+ANNOTATION+EDSN+MSG
12     Invoke checksum validation
13     if checksum is valid then
14         Alert user about success. Convert ANNOTATION to bytes and decompress.
15         Generate the original text and offset the original feature locations by the length of
             len(BESN+PLASMID_ID+ANNOTATION+CRC+ECC+EDSN).
16         Write the sequence as
             BESN+PLASMID_ID+ANNOTATION+CRC+ECC+EDSN+MSG.
17     else
18         Alert user about failure and start error correction procedure.
19         Send BESN+PLASMID_ID+ANNOTATION+CRC+EDSN+MSG to Reed-Solomon
             decoder.
20         if decoder outputs null or same as input then
21             Alert user that errors are more than tolerable limit. Cannot generate Genbank and exit.
22         else
23             Get the corrected parts and re-invoke checksum verification.
24             if re-verification is success then
25                 Alert user that verification succeeded after error correction. Follow Step 14 to 16.
26             else
27                 Alert user that verification failed even after successful correction.
28 else
29     Alert user that BESN and EDSN tags are not present.
```

the digital file but included a dual signature such that the receiver can strictly tie one digital file with exactly one physical sample. If we can encode the documentation about the sample also within it, there will be no need to share any extra information and the physical sample will the self validating and self documenting. As of now, the signature and the self documenting parts are implemented as two different modules. In future they will be combined to generate signatures and annotations together.

One of the future directions in this work 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 validate Alice's DNA. Then Bob continues to modify it, then signs it and sends it to Eve. Can Eve only verify Bob's signature, or is there a way for Eve to track the entire pathway starting from Alice? It would be interesting to see if the concept of aggregate signatures can be applied in these scenarios.

Another direction to be explored is signing a part of the DNA. In this work, we have only applied signatures on plasmids and the entire plasmid sequence. Plasmids are relatively smaller in size than genomes ranging from 2.5 - 25 kilobases, and hence sequencing a plasmid without any errors is feasible. But as we move from plasmids to genomes, the sheer size of the DNA makes it almost impossible to produce and error free sequencing. The error correction code can be used to tolerate some errors but it might not be the optimal solution. It can be a better solution to sign a part of the DNA rather than the entire DNA sequence for example a particular protein sequence. In that way, we can ensure that the protein sequence is unchanged although there might be some errors in rest of the DNA sequence.

Also, it would be interesting to see if we put a signature on top of an existing signature whether the characteristic of the DNA molecule changes or not. If it does not, how many signatures can be inserted before the characteristics of the original DNA molecule begin to change? If we cannot put multiple signatures within the same DNA molecule, how do we remove the signature that was present before signing it again? Does removing the signature also alter the property of the DNA molecule? These are some future directions that we plan to explore further.

Acknowledgment

This work was partly supported by the U.S. National Science Foundation's award #1934573 "EAGER: Development of a tool-chain to write and read self-documenting plasmids", and award #1832320 "EAGER: Modeling DNA Manufacturing Processes Using Extensible Attribute Grammars", and by the Colorado State University's Office of the Vice President for Research Catalyst for Innovative Partnerships Program. The work of Indrajit Ray was performed while serving as Program Director at the U.S. National Science Foundation (NSF) and supported by the foundation's Independent Research and Development program for staff. Research findings presented here and opinions expressed are solely that of the authors, and in no way reflect the opinion of the NSF, other federal agencies or the Office of the Vice President for Research of Colorado State University.

References

[1] *Biodefense in the Age of Synthetic Biology*, National Academies of Sciences, Engineering and Medicine, 2018.

[2] D.M. Kar, I. Ray, J. Gallegos and J. Peccoud, Digital Signatures to Ensure the Authenticity and Integrity of Synthetic DNA Molecules, in: *Proceedings of the New Security Paradigms Workshop*, NSPW '18, ACM, Windsor, UK, 2018, pp. 110–122.

[3] P. Ney, K. Koscher, L. Organick, L. Ceze and T. 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, 2017.

[4] D.C. Jupiter, T.A. Ficht, J. Samuel, Q.-M. Qin and P. de Figueiredo, DNA Watermarking of Infectious Agents: Progress and Prospects, *PLOS Pathogens* **6**(6) (2010), 1–3.

[5] C.A. Hutchison, R.-Y. Chuang, V.N. Noskov, N. Assad-Garcia, T.J. Deerinck, M.H. Ellisman, J. Gill, K. Kannan, B.J. Karas, L. Ma, J.F. Pelletier, Z.-Q. Qi, R.A. Richter, E.A. Strychalski, L. Sun, Y. Suzuki, B. Tsvetanova, K.S. Wise, H.O. Smith, J.I. Glass, C. Merryman, D.G. Gibson and J.C. Venter, Design and Synthesis of a Minimal Bacterial Genome, *Science* **351**(6280) (2016).

[6] D.G. Gibson, J.I. Glass, C. Lartigue, V.N. Noskov, R.-Y. Chuang, M.A. Algire, G.A. Benders, M.G. Montague, L. Ma, M.M. Moodie, C. Merryman, S. Vashee, R. Krishnakumar, N. Assad-Garcia, C. Andrews-Pfannkoch, E.A. Denisova, L. Young, Z.-Q. Qi, T.H. Segall-Shapiro, C.H. Calvey, P.P. Parmar, C.A. Hutchison, H.O. Smith and J.C. Venter, Creation of a Bacterial Cell Controlled by a Chemically Synthesized Genome, *Science* **329**(5987) (2010), 52–56.

[7] S.M. Richardson, L.A. Mitchell, G. Stracquadanio, K. Yang, J.S. Dymond, J.E. DiCarlo, D. Lee, C.L.V. Huang, S. Chandrasegaran, Y. Cai, J.D. Boeke and J.S. Bader, Design of a Synthetic Yeast Genome, *Science* **355**(6329) (2017), 1040–1044.

[8] M. Liss, D. Daubert, K. Brunner, K. Kliche, U. Hammes, A. Leiherer and R. Wagner, Embedding Permanent Watermarks in Synthetic Genes, *PLOS ONE* **7**(8) (2012), 1–10.

[9] D. Heider and A. Barnekow, DNA-based Watermarks Using the DNA-Crypt Algorithm, *BMC Bioinformatics* **8**(1) (2007).

[10] A. Shamir, Identity-Based Cryptosystems and Signature Schemes, in: *Advances in Cryptology*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 1984, pp. 47–53.

[11] I.S. Reed and G. Solomon, Polynomial Codes Over Certain Finite Fields, *Journal of the Society for Industrial and Applied Mathematics* **8**(2) (1960), 300–304.

[12] J.S. Plank et al., A Tutorial on Reed-Solomon Coding for Fault-tolerance in RAID-like Systems, *Software Practice and Experience* **27**(9) (1997), 995–1012.

[13] V.I. Levenshtein, Binary Codes Capable of Correcting Deletions, Insertions, and Reversals, *Soviet physics doklady* **10**(8) (1966), 707–710.

[14] F.J. Damerau, A Technique for Computer Detection and Correction of Spelling Errors, *Communications of ACM* **7**(3) (1964), 171–176.

[15] Damerau – Levenshtein Distance, 2019.

[16] M.A. Jaro, Advances in Record-Linkage Methodology as Applied to Matching the 1985 Census of Tampa, Florida, *Journal of the American Statistical Association* **84**(406) (1989), 414–420.

[17] P. Jaccard, Etude De La Distribution Florale Dans Une Portion Des Alpes Et Du Jura, *Bulletin de la Societe Vaudoise des Sciences Naturelles* **37**(142) (1901), 547–579.

[18] P. Jaccard, Distribution De La Flore Alpine Dans Le Bassin Des Dranses Et Dans Quelques Régions Voisines., *Bulletin de la Societe Vaudoise des Sciences Naturelles* **37**(140) (1901), 241–72.

[19] R. Sakai and M. Kasahara, ID Based Cryptosystems with Pairing on Elliptic Curve, *IACR Cryptology ePrint Archive* (2003).

[20] R. Sakai, K. Ohgishi and M. Kasahara, Cryptosystems Based on Pairing, in: *Proceedings of the 2000 Symposium on Cryptography and Information Security*, Okinawa, Japan, 2000.

[21] K.G. Paterson, ID-based Signatures from Pairings on Elliptic Curves, *Electronics Letters* **38**(18) (2002), 1025–1026.

[22] J.C. Choon and J. Hee Cheon, An Identity-Based Signature from Gap Diffie-Hellman Groups, in: *Public Key Cryptography — PKC 2003*, Y.G. Desmedt, ed., Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2002, pp. 18–30.

[23] X. Yi, An Identity-based Signature Scheme from the Weil Pairing, *IEEE Communications Letters* **7**(2) (2003), 76–78.

[24] A. De Caro and V. Iovino, jPBC: Java Pairing Based Cryptography, in: *Proceedings of the 16th IEEE Symposium on Computers and Communications, ISCC 2011*, IEEE, Kerkyra, Corfu, Greece, June 28 - July 1, 2011, pp. 850–855.

[25] G. Wood et al., Ethereum: A secure decentralised generalised transaction ledger, *Ethereum project yellow paper* **151** (2014), 1–32.

[26] S. Nakamoto et al., Bitcoin: A peer-to-peer electronic cash system (2008).

[27] D. Boneh and M. Franklin, Identity-based encryption from the Weil pairing, in: *Annual international cryptology conference*, Springer, 2001, pp. 213–229.