That's My DNA: Detecting Malicious Tampering of Synthesized DNA

Diptendu Mohan Kar, Indrajit Ray

That's My DNA: Detecting Malicious Tampering of Synthesized DNA

Diptendu Mohan Kar and Indrajit Ray

Colorado State University, Fort Collins CO 80523, USA
{diptendu.kar, indrajit.ray}@colostate.edu

Abstract. The area of synthetic genomics has seen rapid progress in recent years. DNA molecules are increasingly being synthesized in the laboratory. New biological organisms that do not exist in the natural world are being created using synthesized DNA. 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 [13] 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, the earlier work is limited in scope. In this work, we extend the work in several directions to address these problems.

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

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 making 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 [16]. 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 [12]. The synthetic genomics community has demonstrated the feasibility of this approach by inserting short watermarks into DNA without introducing significant perturbation to genome function [8, 6, 20, 15]. The use of watermarks has also been proposed in order to identify genetically modified organisms (GMOs) or proprietary strains. Heider et al. [7] 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. [13] 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 [23], Reed-Solomon error-correction codes [19, 18] 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 (discussed in more details in Section 2) that Kar et al. do not address. The current work improves the previous scheme to address these problems (Sections 3 and 4). 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 (Section 5).

2 Limitations of Earlier Work and Current Contributions

2.1 Cyclic shifts and reverse complement

In [13], 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 [13], 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 3.

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 4.

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 5.

3 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 use block-based error correction codes, such as a Reed-Solomon code [19], 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.

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

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

Input checks e.g. correct file extension, ORCID format, integers etc.
Parse GenBank file. Split content and sequence based on keyword ORIGIN. Parse content to get the list of feature locations.
if *Location of signature placement NOT within a feature* **then**
    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.
    Generate hash (SHA-256) of this sequence.
    Generate signature on the hash.
    Convert the signature bytes,ORCID and Plasmid_ID to ACGT sequence. Create the following string by concatenating parts :
    BESN+ORCID+Plasmid_ID+SIN+EDSN
    if *error tolerance NOT 0* **then**
        Append MSG (shifted sequence) before BESN+ORCID+PLASMID_ID+SIN+ESN.
        Pass SEQUENCE+BESN+ORCID+PLASMID_ID+SIN+ESN to Reed-Solomon Encoder.
        Convert the parity bytes to ACGT. (call this ECC)
        Signature_Sequence = BESN+ORCID+PLASMID_ID+SIN+ECC+EDSN.
    else
        Signature_Sequence = BESN+ORCID+PLASMID_ID+SIN+EDSN.
    if *signature placement location is start of the original sequence* **then**
        Final_Sequence = SEQUENCE+Signature_Sequence
    else if *signature placement location is end of the original sequence* **then**
        Final_Sequence = Signature_Sequence+SEQUENCE
    else
        part1 = prefix of SEQUENCE of length $n-1$ (where signature is to be placed at location n
        part2 = suffix of SEQUENCE of length $len(SEQUENCE) - n + 1$
        Final_Sequence = part1+Signature_Sequence+part2
    Write the Final_Sequence to a new GenBank file and FASTA file.
else
    Alert user about collision. Allow user to input new location. Go to step 3 with new location.

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.

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.
```

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.

4 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 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.

We used the Optimal String Alignment variant of the Damerau-Levenshtein algorithm [2] as our preferred method for string matching. For a discussion on the experiments we performed to arrive at this decision please refer to Appendix 1.

5 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* [22], *Sakai-Ohgishi-Kasahara* [21], *Paterson* [17], *Cha-Cheon* [3], and *Xun Yi* [24]. 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) [5]. 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.

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

1. For an identity ID_A, calculate $C_A = H(ID_A)$. That is map the identity string to an element of G_1.
2. Calculate $V_A = x \cdot C_A$.

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

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

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

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

Verify: The verification procedure is as follows:

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

The above equation works because:

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

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

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.

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.

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

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.

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

Curve Name	**Signature Size using SHA-1 (Bytes)**	**Signature Size using SHA-256 (Bytes)**
a.properties	56	60
a1.properties	594	448
d159.properties	102	98
d201.properties	121	138
d224.properties	129	131
e.properties	262	214
f.properties	133	251
g149.properties	170	219

From the speed perspective, 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 288 base pairs. The only thing it affects in our earlier algorithms is determination of `BESN` and `EDSN` in Section 4 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.

6 Conclusion and Future Work

In this work, we improve the previous DNA signing scheme [13] 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. That is almost 43% gain in signature length.

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, signs it and sends it to Mallory. Can Mallory only verify Bob's signature, or is there a way for Mallory to track the entire pathway starting from Alice? It would be interesting to see if the concept of aggregate signatures can be applied in these scenarios. Also, it would be interesting to see if we put a signature on top of an existing signature whether the characteristic of the DNA changes or not. If it does not, how many signatures can be inserted before the characteristics of the original DNA molecule begin to change? Also, if we cannot put multiple signatures within the same DNA molecule, how do we remove the signature that was present before signing it again. Finally, does removing the signature also alters the property of the DNA?

Acknowledgment

This work is partly based on research supported by the Office of the Vice President of Research, Colorado State University. This material is also based upon work performed by Indrajit Ray while serving at the National Science Foundation. Research findings presented here and opinions expressed are solely those of the authors and in no way reflect the opinions of Colorado State University, the U.S. NSF or any other federal agencies. The authors would like to thank Jenna Gallegos and Jean Peccoud for their comments and suggestions.

References

1. Biodefense in the Age of Synthetic Biology. National Academies of Sciences, Engineering and Medicine (Jun 2018)
2. Damerau – Levenshtein Distance. Wikipedia (Feb 2019)
3. Choon, J.C., Hee Cheon, J.: An Identity-Based Signature from Gap Diffie-Hellman Groups. In: Desmedt, Y.G. (ed.) Public Key Cryptography — PKC 2003. pp. 18–30. Lecture Notes in Computer Science, Springer Berlin Heidelberg (2002)
4. Damerau, F.J.: A Technique for Computer Detection and Correction of Spelling Errors. Communications of ACM **7**(3), 171–176 (Mar 1964)
5. De Caro, A., Iovino, V.: jPBC: Java Pairing Based Cryptography. In: Proceedings of the 16th IEEE Symposium on Computers and Communications, ISCC 2011. pp. 850–855. IEEE, Kerkyra, Corfu, Greece, June 28 - July 1 (2011)
6. Gibson, D.G., Glass, J.I., Lartigue, C., Noskov, V.N., Chuang, R.Y., Algire, M.A., Benders, G.A., Montague, M.G., Ma, L., Moodie, M.M., Merryman, C., Vashee, S., Krishnakumar, R., Assad-Garcia, N., Andrews-Pfannkoch, C., Denisova, E.A., Young, L., Qi, Z.Q., Segall-Shapiro, T.H., Calvey, C.H., Parmar, P.P., Hutchison, C.A., Smith, H.O., Venter, J.C.: Creation of a Bacterial Cell Controlled by a Chemically Synthesized Genome. Science **329**(5987), 52–56 (2010)
7. Heider, D., Barnekow, A.: DNA-based Watermarks Using the DNA-Crypt Algorithm. BMC Bioinformatics **8**(1) (May 2007)
8. Hutchison, C.A., Chuang, R.Y., Noskov, V.N., Assad-Garcia, N., Deerinck, T.J., Ellisman, M.H., Gill, J., Kannan, K., Karas, B.J., Ma, L., Pelletier, J.F., Qi, Z.Q., Richter, R.A., Strychalski, E.A., Sun, L., Suzuki, Y., Tsvetanova, B., Wise, K.S., Smith, H.O., Glass, J.I., Merryman, C., Gibson, D.G., Venter, J.C.: Design and Synthesis of a Minimal Bacterial Genome. Science **351**(6280) (2016)
9. Jaccard, P.: 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), 241–72 (1901)
10. Jaccard, P.: Etude De La Distribution Florale Dans Une Portion Des Alpes Et Du Jura. Bulletin de la Societe Vaudoise des Sciences Naturelles **37**(142), 547–579 (1901)
11. Jaro, M.A.: Advances in Record-Linkage Methodology as Applied to Matching the 1985 Census of Tampa, Florida. Journal of the American Statistical Association **84**(406), 414–420 (Jun 1989)
12. Jupiter, D.C., Ficht, T.A., Samuel, J., Qin, Q.M., de Figueiredo, P.: DNA Watermarking of Infectious Agents: progress and Prospects. PLOS Pathogens **6**(6), 1–3 (06 2010)
13. Kar, D.M., Ray, I., Gallegos, J., Peccoud, J.: Digital Signatures to Ensure the Authenticity and Integrity of Synthetic DNA Molecules. In: Proceedings of the New Security Paradigms Workshop. pp. 110–122. NSPW '18, ACM, Windsor, UK (2018)
14. Levenshtein, V.I.: Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. Soviet physics doklady **10**(8), 707–710 (1966)
15. Liss, M., Daubert, D., Brunner, K., Kliche, K., Hammes, U., Leiherer, A., Wagner, R.: Embedding Permanent Watermarks in Synthetic Genes. PLOS ONE **7**(8), 1–10 (08 2012)
16. Ney, P., Koscher, K., Organick, L., Ceze, L., Kohno, T.: 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 (Aug 2017)

17. Paterson, K.G.: ID-based Signatures from Pairings on Elliptic Curves. Electronics Letters **38**(18), 1025–1026 (Aug 2002)
18. Plank, J.S., et al.: A Tutorial on Reed-Solomon Coding for Fault-tolerance in RAID-like Systems. Software Practice and Experience **27**(9), 995–1012 (1997)
19. Reed, I.S., Solomon, G.: Polynomial Codes Over Certain Finite Fields. Journal of the Society for Industrial and Applied Mathematics **8**(2), 300–304 (1960)
20. Richardson, S.M., Mitchell, L.A., Stracquadanio, G., Yang, K., Dymond, J.S., DiCarlo, J.E., Lee, D., Huang, C.L.V., Chandrasegaran, S., Cai, Y., Boeke, J.D., Bader, J.S.: Design of a Synthetic Yeast Genome. Science **355**(6329), 1040–1044 (2017)
21. Sakai, R., Ohgishi, K., Kasahara, M.: Cryptosystems Based on Pairing. In: Proceedings of the 2000 Symposium on Cryptography and Information Security. Okinawa, Japan (January 2000)
22. Sakai, R., Kasahara, M.: ID based Cryptosystems with Pairing on Elliptic Curve. IACR Cryptology ePrint Archive (2003)
23. Shamir, A.: Identity-Based Cryptosystems and Signature Schemes. In: Advances in Cryptology. pp. 47–53. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (Aug 1984)
24. Yi, X.: An Identity-based Signature Scheme from the Weil Pairing. IEEE Communications Letters **7**(2), 76–78 (Feb 2003)

Appendix 1 - Analysis of 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 [14]. Other notable algorithms are Damerau-Levenshtein[4, 14, 2], Optimal String Alignment variant of Damerau-Levenshtein (sometimes called the restricted edit distance) [2], Jaro-Winkler edit distance [11], and Jaccard index [10, 9].

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 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. Then 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 1. As can be seen from the Figures the Jaro algorithm was fairly inaccurate with an average accuracy of only 35.12 %. The Jaccard algorithm fared much better but was still imperfect with an average accuracy of only 95.18 %. All of the three Levenshtein variants were perfectly accurate in their assessment. These results indicate that if accuracy was the chief concern, either of the three Levenshtein variants would be ideal choices.

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 this means little in practice if the algorithm has an untenable long run time. To this end, the speed of the algorithms was compared. To accomplish this each method was used to compare a series of one million random strings of a set

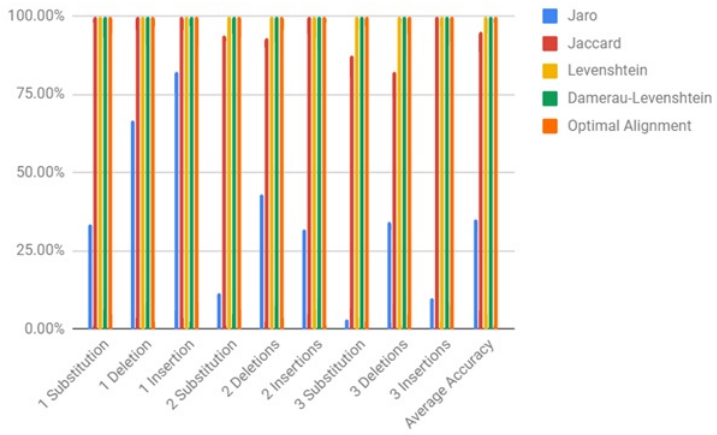

Fig. 1. Accuracy of algorithms per case as a percentage.

length. A graph of the time in milliseconds (ms) for each algorithm is given in Figure 2.

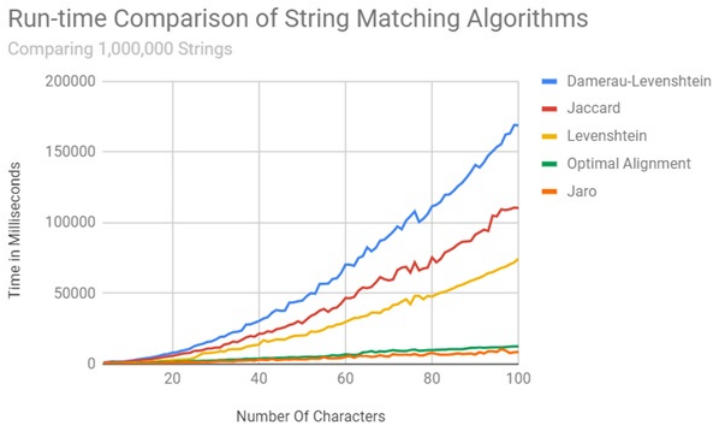

Fig. 2. Runtime analysis of various algorithms in milliseconds.

As can be seen from Figure 2, 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 Optimal String Alignment was chosen as the preferred method.

Appendix 2 - pUC19 DNA before and after signing

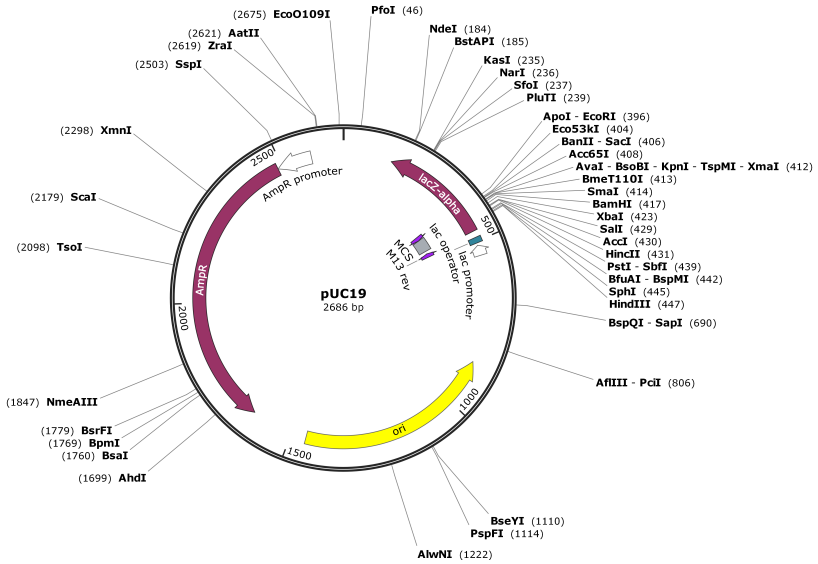

Fig. 3. View of sequenced but unsigned pUC19 in SnapGene editor

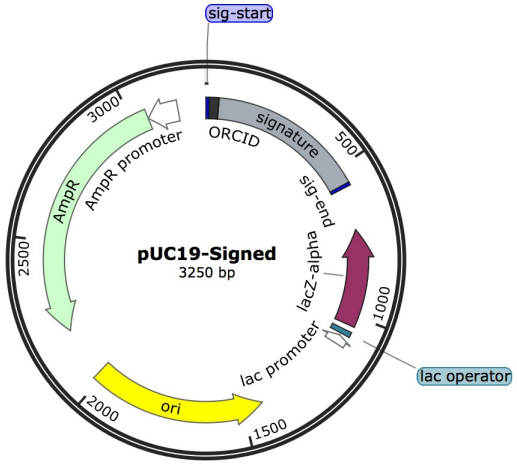

Fig. 4. View of sequenced signed pUC19 in SnapGene showing embedded signature. Note increased size of DNA