Computer Security, Privacy, and DNA Sequencing: Compromising Computers with Synthesized DNA, Privacy Leaks, and More

Peter Ney, Karl Koscher, Lee Organick, Luis Ceze, and Tadayoshi Kohno,
University of Washington

https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ney

This paper is included in the Proceedings of the 26th USENIX Security Symposium

August 16–18, 2017 • Vancouver, BC, Canada

ISBN 978-1-931971-40-9

Open access to the Proceedings of the 26th USENIX Security Symposium is sponsored by USENIX

Computer Security, Privacy, and DNA Sequencing: Compromising Computers with Synthesized DNA, Privacy Leaks, and More

Peter Ney, Karl Koscher, Lee Organick, Luis Ceze, Tadayoshi Kohno
University of Washington
{neyp,supersat,leeorg,luisceze,yoshi}@cs.washington.edu

Abstract

The rapid improvement in DNA sequencing has sparked a big data revolution in genomic sciences, which has in turn led to a proliferation of bioinformatics tools. To date, these tools have encountered little adversarial pressure. This paper evaluates the robustness of such tools *if* (or *when*) adversarial attacks manifest. We demonstrate, for the first time, the synthesis of DNA which — when sequenced and processed — gives an attacker arbitrary remote code execution. To study the feasibility of creating and synthesizing a DNA-based exploit, we performed our attack on a modified downstream sequencing utility with a deliberately introduced vulnerability. After sequencing, we observed information leakage in our data due to sample bleeding. While this phenomena is known to the sequencing community, we provide the first discussion of how this leakage channel could be used adversarially to inject data or reveal sensitive information. We then evaluate the general security hygiene of common DNA processing programs, and unfortunately, find concrete evidence of poor security practices used throughout the field. Informed by our experiments and results, we develop a broad framework and guidelines to safeguard security and privacy in DNA synthesis, sequencing, and processing.

1 Introduction

DNA sequencing costs have dropped exponentially, outstripping Moore's Law since 2008, primarily driven by advances in next-generation sequencing (NGS) technologies. For example, Illumina's cost to sequence the human genome dropped from around $100,000 in 2009 to just $1,000 in 2014 [39]. These advances have revolutionized genomic sciences, accelerating the pace of new discoveries in areas such as cancer biology and epidemiology.

Our research suggests that DNA sequencing and analysis have not to date received significant — if any — adversarial pressure. The key question that motivates our research then, is the following: How robust will the DNA sequencing and processing pipeline be *if* or *when* adversarial pressures manifest? This line of inquiry raises related questions, such as: Are DNA-based attacks possible? What potential consequences could occur if an adversary compromises a component of the DNA processing pipeline? How serious might those consequences be? Since DNA sequencing is rapidly progressing into new domains, such as forensics and DNA data storage [2, 9, 10, 15, 17], we believe it is prudent to understand current security challenges in the DNA sequencing pipeline before mass adoption.

The modern DNA sequencing and analysis pipeline is large, complicated, and computationally-intensive. DNA is pre-processed in a wet lab and analyzed with a high-throughput sequencer (itself a computer) that performs image analysis. It is then common to conduct a wide range of computational tasks with the raw output from the sequencer using many software utilities. We seek to assess the overall state of this pipeline in general, and to experimentally explore key aspects that are *not* represented in traditional computing systems: DNA samples.

Exploiting Computer Programs with DNA. The DNA processing pipeline begins with DNA strands in a test tube. Hence, we start our security explorations from this point. Namely, we first experimentally evaluate whether it is possible to compromise a computer program using physical DNA.

Our exploration of this question lead us to synthesize DNA strands that, after sequencing and post-processing, generated a file; when used as input into a vulnerable program, this file yielded an open socket for remote control. We elaborate on specifics in Section 3.

To the best of our knowledge, ours is the first example of compromising a computer system using biological or synthetic DNA samples. Our exploit did not target a program used by biologists in the field; rather it targeted one that we modified to contain a known vulnerability.

Our use of such a trojaned program was consistent with the primary focus of the first research phase to understand — and overcome — challenges posed by creating an exploit at a physical level. For example, our initial exploit contained too few C and G nucleotides (we review DNA background in Section 2) to synthesize the DNA strand; therefore, we modified our exploit to overcome this challenge. Our key finding is that it *is* possible to encode a computer exploit into synthesized DNA strands.

Side-Effect — Information Leakage. Although not a goal, our efforts to experimentally evaluate the ability to synthesize adversarial DNA resulted in our observing an information leakage channel. Standard practice multiplexes different samples on the same sequencing machine. The methods to multiplex (and later demultiplex) DNA samples can leak information between samples during sequencing. Our exploit sample was sequenced and multiplexed in this manner alongside samples from another research team. We noticed that our sequencing results contained DNA sequences derived from their samples.

Other biologists have observed these effects [16, 19, 25, 27, 33], but their concerns focused on experimental accuracy, not on security or information leakage. From our perspective we use these unanticipated results to guide a security discussion of information leakage inherent in the DNA sequencing pipeline.

Software Security Awareness Throughout the Pipeline. Having demonstrated the ability to exploit a computer program with synthesized DNA, we next evaluated the computer security properties of downstream DNA analysis tools. We analyzed the security of 13 commonly used, open source programs. We selected these programs methodically, choosing ones written in C/C++. We then evaluated the programs' software security practices and compared them to a baseline of programs known to receive adversarial pressure (e.g., web servers and remote shells).

We found that existing biological analysis programs have a much higher frequency of insecure C runtime library function calls (e.g., `strcpy`). This suggests that DNA processing software has not incorporated modern software security best practices. However, rather than rely solely on heuristics, we took the next step and determined whether we could target static buffers to cause program crashes. We readily found three buffer overflow vulnerabilities. Given the prevalence of poor software security practices and the well-known fact that program crashes can often be converted to exploits, we chose not to convert each program crash into a working exploit.

Threat Model and Guidelines. When exploring a technology domain new to computer security, any individual study lacks the breadth to address the entire domain. For example, early work on the attack surface of modern automobiles considered only one vehicle and a few example attacks [7, 20]. However, as the first work to explore a domain, an important contribution can involve drawing inferences from concrete results and domain knowledge to define broader lessons and extrapolate threat models for the entire domain, as others did for the modern automobile [7]. Leveraging our technical results and multidisciplinary backgrounds (computer security, synthetic biology, and the design and use of the DNA processing pipeline), we drew inferences to present a threat model and recommendations for the DNA sequencing and processing pipeline and the associated community.

Summary. To our knowledge, our research is the first to consider computer security implications of the modern DNA sequencing pipeline. Our four key contributions include:

- We demonstrate, for the first time, the ability to compromise a computer program with sequenced DNA. In so doing, we encountered challenges when synthesizing DNA strands containing exploits and developed methods to overcome those challenges.
- We observe a side channel resulting from fundamental properties of DNA sequencing technologies, and we pioneer the exploration of how one might exploit this side channel for adversarial purposes.
- We evaluate the software security in a wide set of DNA processing programs and find that they do not adhere to modern security best practices (e.g., they frequently use insecure function calls and contain buffer overflow vulnerabilities).
- We derive a threat model for the DNA sequencing pipeline and present recommendations to offset potential attacks.

2 Biology and DNA Sequencing: Background

Our work strives to apply computer security principles and perspectives to a new field: genomic sciences, and specifically, DNA synthesis, sequencing, and analysis. To do so, we offer a basic review of the biological, chemical, and computational processes in this field.

2.1 DNA

Deoxyribonucleic acid (DNA) is the carrier of genetic information for all known living organisms. It is composed of an alternating sugar-phosphate backbone to which a sequence of four possible *nucleotides* (also called *bases*) are linearly attached. These nucleotides — adenine, thymine, cytosine, and guanine — are commonly abbreviated as A, T, C, and G, respectively. Each nucleotide bonds with its complement — A with T, and C

with G. *Sequencing* is the process of reconstructing the original order of nucleotides in a DNA sample.

While DNA can form many structures, the most common is double-stranded DNA (dsDNA), where two strands with complementary base sequences bond to form the well-known double helix structure. DNA's sugar-phosphate backbone causes its strand ends to be asymmetric: The phosphate end, called the 5′ end, and the sugar end, called the 3′ end. By convention, nucleotide sequences are read from the 5′ to the 3′ end.

Many traditional lab protocols require DNA strands to be replicated (also called *amplification*). Amplification uses a technique called *polymerase chain reaction*, or PCR. dsDNA is first *melted* at high temperatures to separate its two strands. The temperature is then lowered, and *primers* (synthesized strands typically 20 nucleotides long) anneal (reattach) to the complimentary ends of the DNA strands. At slightly higher temperatures, DNA polymerase (an enzyme that synthesizes DNA), attaches to these end regions where the primer has annealed and produces a complimentary copy of the original strand. This process is repeated as needed to exponentially amplify DNA.

2.2 Next-Generation DNA Sequencing

Next-generation sequencing (NGS) systems differ from prior sequencing methods in that they read relatively short sequences, called *reads*, but in a massively parallel fashion. Longer DNA strands are sequenced by randomly cleaving DNA into shorter sequences, reading these sequences in parallel, and reconstructing the original, longer sequence. Several different types of NGS systems do this work; among the most popular are the various Illumina sequencers, which are based on a technique known as *sequencing by synthesis*.

Before sequencing a typical genomic DNA sample with an Illumina sequencer, the DNA sample must be manually processed in the lab. It is cleaved into short sequences of a few hundred bases and amplified using PCR. Special DNA *adapter* sequences are then attached to both ends of the amplified DNA. This double-stranded DNA sample is separated into single-stranded DNA and applied to a glass flow cell. The adapter sequences attached to the sample fragments bind to complementary fragments on the flow cell surface. The bound sequences locally replicate to produce clusters of identical DNA, called *clonal clusters*.

The DNA in each clonal cluster is sequenced in rounds (called *cycles*) by appending a complementary fluorescently labeled nucleotide to the single-stranded DNA in each clonal cluster. Each time a new fluorescent base is added to the strand, it emits a particular color specific to each base (e.g., A, C, G, and T). The cluster sequence is obtained by imaging the flow cell in each cycle and noting the fluorescent color each cluster emits. The number of cycles determines the length of resulting reads (often between 150-300 bases). These identified bases added in each cycle, called *base calls*, are written out to per-cycle base call files. A separate utility then takes these files and converts the reads into a standard text-based format called FASTQ.

FASTQ files are the de facto standard for exchanging next-generation sequencing results. Their structure is simple: each read has an ASCII header identifying the read source, followed by a line with the sequence written as an ASCII A, C, G, or T. Reads additionally contain a separator line, followed by a line with ASCII characters encoding the quality or confidence of each base call.

2.3 Downstream Processing

The raw FASTQ files that come directly from the sequencer are rarely useful by themselves, and extensive downstream processing and analysis is usually performed after sequencing. This processing is typically done in phases by dedicated programs; the output from a program in one stage is sent to a program in a later processing stage. This section describes some commonly used downstream processing steps, which we explore for security vulnerabilities in Section 6.

Before analyzing the sequence reads, an initial preprocessing phase occurs where by the reads (stored in a FASTQ files) are cleaned up to remove undesired ones. The last base calls in a read often have lower quality scores, so it is common to truncate the reads to a fixed length when the score drops below a defined threshold. DNA sequences from unintended sources — like the adapters used to bind sample DNA to the flow cell or control sequences used to verify sequencing accuracy — need to be removed from the sequence file. Other preprocessing steps merge paired-end reads if there is overlap, convert different quality score file formats, or compress FASTQ files for archival purposes.

Direct output from a sequencer contains only short chunks of reads derived from the full sequence, and in no particular order. These unordered reads can be merged by aligning them to a reference sequence (e.g., the human genome) if one exists, or they can be merged from scratch, using overlaps in the reads to stitch them together in a method called *de novo* assembly. When using a reference sequence, the alignment of each read in relation to the reference is stored in a text based format (SAM) or a compressed representation (BAM). Both methods, especially *de novo* assembly, are computationally and memory intensive and may be run on computer clusters if the size of the sample to reconstruct is sufficiently large (e.g., a mammalian genome).

After the sequence has been aligned or assembled more work may remain, and the following are but a

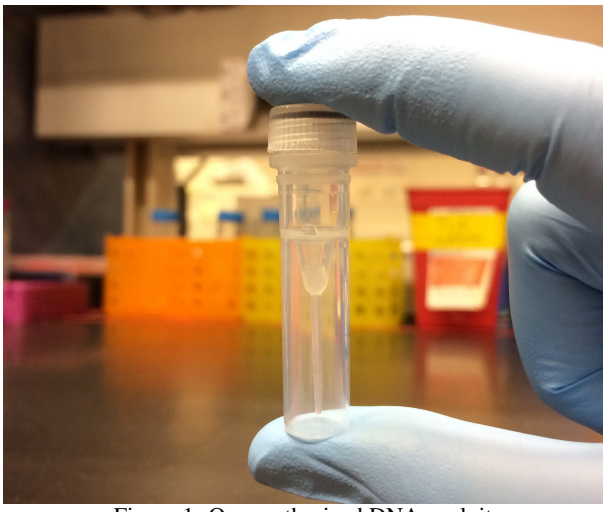

Figure 1: Our synthesized DNA exploit

few examples of the widely varied analysis methods commonly used. It is customary to look for variations between the sample and some reference for biologically meaningful differences (e.g., genetic variations that cause disease). Specific variations in the sequenced sample are usually stored in a plain text file (VCF) so redundant sequencing information can be discarded. NGS techniques are also used in more complicated biological assays to analyze RNA (RNA-seq) or protein-DNA interactions (ChIP-seq). In these cases, the samples' sequence are not only valuable, but the number and precise location of its reads in relation to a reference sequence are also meaningful.

2.4 DNA Synthesis

Synthetic DNA, commercially produced via phosphoramidite chemistry, is characterized by nucleotides attached to one another with specific reagents to form specified sequences. The resulting length, quality, and cost varies greatly depending on the method of reagent delivery, the substrate on which DNA is synthesized, and consumer specifications. For example, Integrated DNA Technologies (IDT) synthesis of a custom gene utilizes their "gBlock" service, which differs in capabilities and constraints from their "custom oligo" service designed for shorter strands (oligos or oligonucleotides are short DNA sequences commonly used in genetics). The cost for these two services varies significantly depending on the length of the strand ordered, the degree to which DNA must be washed, or whether there are DNA modifications (e.g., fluorescent tags).

3 Compromising a Computer with DNA

DNA, in its most basic form, stores data. Conceptually, if DNA were used as input to a computer system, an open issue is the possibility that it could be used to compromise that system. As one might predict, significant unknowns exist. Can DNA itself compromise a computer system, or does something in the DNA sequencing pipeline make such attacks impossible? Prior to our work, to the best of our knowledge, there has never been a demonstrated DNA-based exploit of a computer system. Indeed, without concrete, experimental evidence, it is impossible to know whether DNA-based computer compromises are purely hypothetical or a real possibility. We therefore seek to experimentally answer the previously unexplored question:

> **Can DNA be used to compromise a computer?**

To answer this question, we seek an end-to-end experimental evaluation of an exploit. Namely, we seek to mimic an adversary and (1) synthesize a real, biological DNA sequence with a malicious, embedded exploit. We then seek to experimentally evaluate the impact of that exploit DNA on a victim by having the victim (2) sequence that DNA using standard sequencing methods and (3) post-process the DNA sequence with a realistic program — a program that a scientist might use to analyze the resulting DNA sequence. If the exploit is successful, step (3) should result in arbitrary code execution on the victim computer.

This section explores the biological nature of this attack pipeline — how to encode an exploit into DNA such that, when sequenced, will hijack execution when processed by the victim program. We therefore intentionally chose to create our own vulnerable program for step (3), i.e., a program inspired by actual bioinformatics tools but with an obvious vulnerability. In Section 6, we consider the security of the sequencing pipeline in general. Our results suggest that while our exploited program in this section is vulnerable to a basic buffer overflow exploit, the security hygiene of the overall DNA sequencing pipeline is not much better.

Despite challenges, this section demonstrates that it *is* possible to create DNA that, when sequenced and processed, compromises a victim system. See Figure 1 for a photo of our DNA exploit. In conducting this work, we identified and overcame multiple challenges, which we describe — along with methods for overcoming them and the resulting lessons — below.

3.1 Target Program

The FASTQ compression utility, `fqzcomp`, is designed to compress DNA sequences. For experimental purposes, we inserted a vulnerability into this utility. To do so, we first copied `fqzcomp` from `https://sourceforge.net/projects/fqzcomp/` and inserted a vulnerability into version 4.6 of its source code; a function that processes and compresses DNA reads individually, using a fixed-size buffer to store the compressed data. This

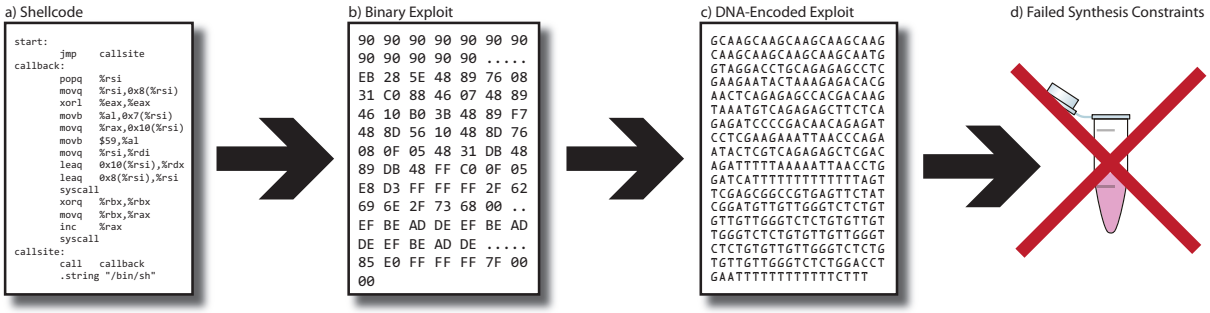

Figure 2: Our initial, unsuccessful exploit attempt

modification lets us perform a buffer overflow with a longer than expected DNA read in order to hijack control flow. While the use of such a fixed-size buffer is an obvious vulnerability, we note that `fqzcomp` already contains over two dozen static buffers. Our modifications added 54 lines of C++ code and deleted 127 lines from `fqzcomp`.

Our modified `fqzcomp` version used a simple 2-bit DNA encoding scheme. The four nucleotides were encoded as two bits — A as 00, C as 01, G as 10, and T as 11 — packing bits into bytes starting with the most significant bits.

We ran the target program in a simplified computing environment and disabled common security features. Specifically, we disabled stack canaries and ASLR, and we marked the stack as executable.

We stress that our target modified program has a known, and in some sense trivial, vulnerability. We also stress that its environment is in many ways the "best possible" environment for an adversary. For experimental purposes, however, we believe that these conditions are acceptable for the following reasons. First, our primary goal is to understand the issues unique to DNA-encoded exploits. Second, as we relate in Section 6, we find that the general security hygiene of bioinformatics programs is very low, with prevalent usage of fixed-size buffers, `strcpy`, and so on. Finally, we note that genome sequencing processes are rapidly improving: since early NGS machines read sequences on the order of 50-100 bases, a fixed-size buffer in that range may have been acceptable years ago. Today, any fixed-size buffer would likely be vulnerable, as new longer read sequencing technologies can produce reads that are upwards of 60,000 bases [30]. These newer sequencers lack the throughput of short-read counterparts and are not at present commonly used; Illumina short-read sequencers now have over 90% market share [18]. Future technological improvements will likely make long-read sequencers more viable in the future.

3.2 Creating and Synthesizing an Exploit

We now turn to our design of a DNA strand that, when sequenced, exploits the vulnerable target program. Our key goal was to identify potential challenges. Our efforts here were successful in two regards. First, we identified several challenges, including limitations on the exploit's size and type and problems inherent in the DNA synthesis process that constrained the sequences we could generate. Second, by overcoming these challenges, we found that it *was* possible to create a DNA sequence that could in fact compromise a program.

Our process was iterative. We created exploits that we thought would work, surfaced challenges, and then iterated on improved exploits.

We initially encoded one of the most straight-forward exploits, i.e., overwriting the return instruction pointer on the stack to point back into shellcode from Aleph One's "Smashing the Stack for Fun and Profit" [26]. We made minor modifications to port the shellcode to the 64-bit Linux syscall interface. To simplify exploit testing, we used a stripped-down version of the vulnerable program that simply compressed a single DNA read into a fixed-size buffer. Our shellcode was 55 bytes long, with another 39 bytes of padding needed for cache line alignment and saved registers. We filled this space with NOPs and bogus saved register values (`0xdeadbeef`). The resulting exploit, 94 bytes long, was encoded as 376 nucleotides. Figure 2 shows this process.

We submitted this sequence to the IDT gBlocks synthesis service, which creates synthetic gene fragments up to 3,000 bases long. Unfortunately, at this step we faced our first challenges. Our sequence contained many issues that prevented IDT from being able to synthesize our order:

- The NOP sled produced a repetitive sequence (`GCAA`) near the start of our sequence, which contributed to more than 69% of the sequence. Repetitive sequences can cause difficulties in sequencing and may cause the physical strand to fold in on itself or form other secondary structures because of DNA's complementary nature.

Figure 3: Our working exploit pipeline

- The negative offset JMP created a run of 13 consecutive Ts. Long runs of the same base, called *homopolymers*, can be difficult to accurately synthesize. The gBlocks service limits homopolymers to no more than nine As or Ts and five Gs or Cs.
- The repeated `0xdeadbeef` bytes produced a long (40+ base pair) repetitive sequence.
- The NOP sled resulted in low GC-content near the beginning of the sequence. Cs and Gs physically bind together more tightly than As and Ts and thus add stability to the DNA strand. Typically, each 20-base window must have 25 to 75 percent GC-content. The first and last 20 bases of a sequence are even more constrained since they must have 40 to 60 percent GC-content to be synthesized.
- A 20 base pair window containing the 13 base pair homopolymer did not meet the minimum GC-content threshold.

Another challenge we faced was the length of our exploit. Our Illumina NextSeq sequencer is rated for a maximum of 300 base pair reads, while the Illumina MiSeq is rated for a maximum of 600 base pair reads.

We addressed these challenges by making our target program and exploit designs more sophisticated. To minimize the number of homopolymers introduced by large pointers and offsets, we switched to targeting the 32-bit x86 instruction set architecture (ISA). We also reduced the buffer size in our target program to minimize the required size of our sequence. Since our ultimate goal was arbitrary remote code execution, we investigated swapping out Aleph One's simple shellcode, which simply spawns a local shell, with one that provided a reverse shell over TCP. We explored the `shell-storm.org` archive for a suitable example; however, even the most compact shellcode was too long to fit inside a sequence that could be reasonably sequenced by the NextSeq sequencer.

Our second exploit attempt uses an obscure feature of `bash`, which exposes virtual `/dev/tcp` devices that create TCP/IP connections. We use this feature to redirect `stdin` and `stdout` of `/bin/sh` to a TCP/IP socket, which connects back to our server. We combined this tactic with a return-to-libc attack that calls `system()`, resulting in a 43-byte exploit, shown in Figure 3. We used a short, fully qualified domain name we controlled as well as a single digit port number to keep exploit length as short as possible. While we considered obtaining a smaller FQDN (e.g., `r.sh`) to keep our exploit size as small as possible, we hypothesized that we could successfully sequence our 176-base[1] DNA strand with our Illumina NextSeq despite exceeding its recommended single-ended read size.

Since the bulk of this exploit consists of lowercase letters, whose two most significant bits were 01 in ASCII — or encoded as a nucleotide, C — we got an acceptable level of GC-content throughout the exploit. The one exception was near the original port number — 3 (encoded as `ATAT`) — which we changed to 9 (encoded as `ATGC`) to maintain a minimum level of GC-content. This sequence was accepted by the IDT gBlocks service with no errors or warnings. IDT's retail cost to synthesize of up to 500 base pairs was $89 USD.

As is standard for NGS runs, our sample was tagged and extended with a unique index (GCCAAT, in our case) and co-sequenced with other experiments. The sequencer was configured to perform 177 non-index read cycles; this is the typical configuration used by another research group that manages the sequencing machine and was sufficiently long to contain the 176 base pair exploit sequence within a single read.

The sample was sequenced on all four lanes (physically separate portions) of the flow cell. After demultiplexing by indices, there were four separate FASTQ files (one for each lane) together containing 811,118 reads.

We processed the four FASTQ files separately, which is done to account for lane-specific errors. We filtered out low-quality reads that did not identify one or more bases; these bases appear as Ns (representing an unknown base) in the FASTQ file. We provided the filtered FASTQ file from the first lane to our modified `fqzcomp` program, which immediately called back to our server, giving us

[1] A bug in our DNA encoding program repeated the final byte, which unnecessarily extended our exploit by four bases, but otherwise did not affect our results.

arbitrary remote code execution via a bash shell.

3.3 Exploit Reliability

The exploit was not robust to errors in sequencing; a single miscalled base would break the exploit. In this experiment, 76.2% of the reads were sequenced with no error. Another issue arose because DNA strands are randomly sequenced in the forward or reverse direction. Reverse sequenced reads will have the reverse complement sequence of the exploit, which is not functional code (see Section 4.2 for a possible solution to this problem). Of the remaining, error free reads, 49.1% were sequenced in the forward direction. Therefore, 37.4% of all reads contained working exploit code (i.e., in the forward direction with no sequencing errors).

The modified `fqzcomp` program contained a buffer too small for the 177 base pair read length, so it would overflow after processing the first read. Therefore, the first read in the file must be the exploit sequence for the exploit to work. With reads randomly appearing in a FASTQ file, we would expect the modified program to be exploited 37.4% of the time. Assuming all four lane files were processed, an attacker would be successful at least once 84.5% of the time. In our case, only the file from the first lane was a successful exploit.

4 Challenges in Encoding Malicious DNA

Informed by our evaluation of the feasibility of manufacturing synthetic DNA capable of exploiting computer systems, we next consider some challenges in crafting arbitrary exploits against other programs and identify directions for future research. In particular, while it is convenient to think of DNA as a simple storage mechanism, our results in Section 3 show that in practice there are several physical and computational constraints that limit the design space of DNA-based exploits.

4.1 Physical Constraints

Any DNA-based exploit must be physically instantiable in DNA. Therefore, any difficulties in the synthesis or amplification of DNA will constrain the sequences attacker can easily synthesize.

Primers. As previously mentioned, it is necessary to amplify the exploit sequence to increase its yield before sequencing. A simple way to do so is to use PCR, which requires a pair of primers to initiate replication. These primers, single stranded DNA sequences usually 18-22 bases long, are complementary to the ends of the target sequence being amplified. PCR primers used together must have similar melting point temperatures to maintain high amplification efficiency. They must also have a high enough annealing temperature to bind only to their complementary locations without mis-pairing to similar sequences. Other parameters also influence primer design such as the amplification region specificity desired, and the GC-content of the primer regions to be amplified.

Primer designing utilities, like Primer3, take these parameters into account to design optimal primer sequences [37]. Since the primers must be complementary to the ends of the exploit sequence, any restrictions in their design will necessarily constrain the ends of the exploit sequence.

Synthesis. DNA synthesis has its own physical constraints that vary across synthesis companies. In Section 3.2 we described constraints imposed by IDT's gBlock gene fragment service, a relatively low cost synthesis method. They required 25 to 75 percent GC-content per 20 base window, A/T and G/C runs no greater than 9 and 6 base pairs, respectively, and sequences that avoided secondary structures (created when different portions of the same strand are complementary to one another).

These synthesis constraints are common but not universal. Different synthesis methods and services can vary in their precise requirements — for example, IDT's custom gene service can tolerate longer homopolymers than gBlock, which may make it easier to synthesize 64-bit addresses. In cases where the exploit cannot be synthesized by any *de novo* synthesis service, it may be possible to synthesize sub-sequences and recombine them manually in a wet lab.

DNA synthesis services also follow strict guidelines to ensure that biologically malicious sequences are not synthesized and spliced into organisms that potentially create pathogens, toxins, or various other harmful products. The shipping, receiving, or purchase of all synthesized sequences must follow guidelines including, but not limited to, those described in the current U.S. Department of Health and Human Services (HHS) and U.S. Department of Agriculture (USDA) Select Agents and Toxins regulations [4–6].

4.2 Sequencing Randomness

Being a biochemical process, DNA sequencing is inherently noisy and random; long DNA strands are randomly cleaved into smaller ones and strands are sequenced in no particular order. This randomness makes DNA-based exploits probabilistic in nature, as discussed in Section 4.2. Robustness against random variations depends on factors like the vulnerability type and what stage in the pipeline is attacked. In general, analysis further along the sequencing pipeline works with more structured data, which will reduce the initial randomness from the sequencer. For example, variant calling programs return processed data in the same order as the reference sequence regardless of the initial read order.

Another source of randomness is that reads will be sequenced in both the forward and reverse direction, which

causes problems because most exploit sequences will be functional only if read in one direction. One solution is to synthesize strands that generate the same reads when sequenced from either end. These can be created by concatenating the forward exploit sequence to its reverse complement (e.g., ACCTG becomes ACCTGCAGGT). Since DNA is always read from 5′ to 3′, the same read will appear, regardless of whether the DNA was sequenced in the forward or reverse direction.

These palindrome like sequences are difficult to synthesize directly because the two halves will bind to each other and create secondary structures. Instead, the two halves could be synthesized separately and conjoined manually in a wet lab.

4.3 Encoding Exploits

Exploits typically contain up to three components: pointers, either to functions or data, instructions in the target instruction set architecture (ISA), and an encoded and/or obfuscated payload. DNA-based exploits introduce unique constraints on each of these components.

Pointers. Bioinformatics programs vary in how they encode DNA data. Some perform a straightforward mapping, encoding each base as two bits and packing these bits together, like our target program in Section 3. However, sequences often have non-standard bases, such as Ns to encode unknown nucelotides or Rs to indicate either an A or G. To support these non-standard bases, some tools use four-bit encodings, or even 8-bit ASCII. Since we can synthesize only standard bases, these alternative encodings will constrain the pointers that we can encode.

Another issue concerns sequencing accuracy and how that will affect the resulting sequence of pointers. Some pointers, such as those to libc or ROP gadgets, are intolerant of any errors. Others, such as pointers to attacker-controlled buffers, can be made somewhat tolerant to errors in the least-significant bits — for example, it could point to a large NOP sled.

Pointers often contain long runs of identical bits and therefore generate homopolymers. For example, without ASLR enabled, 64-bit Linux places user stacks at `0x00007fffffffffff`, which contains a run of 47 consecutive 1s. Using two-bit encoding, this results in a homopolymer of 23 bases. As previously described, a solution is to use a synthesis service more tolerant to homopolymers.

Code. Executable sequences of target ISA instructions can encode malicious operations more compactly than equivalent ROP chains and are easier to develop, which makes them desirable to attackers. However, encoding ISA instructions in DNA presents a number of challenges.

As with pointers, the target program's DNA encoding may restrict the bytes that can be represented. Depending on the encoding and ISA, this could also limit the set of instructions that are available.

The regular structure of most ISAs produces repeated base sequences when encoded into DNA, which again, are difficult to synthesize. Semantically-equivalent instructions and semantic NOPs can be used to break up repetitive sequences to make exploits easier to synthesize.

Another issue to consider is read length. All but the most trivial exploits exceed the read length of most high-throughput sequencers, and thus, the exploit will be randomly cleaved. Depending on which part of the pipeline is being exploited (i.e., whether the target program processes raw reads or fully aligned sequences), this could decode in the middle of a multi-byte instruction, or even in the middle of a byte. Therefore, for robustness, an exploit should encode instructions that are tolerant to such shifts. Prior work demonstrates techniques to generate these types of resynchronizing instruction sequences [22]. Long read sequencers may mitigate these challenges in the future but are currently less accurate than high-throughput sequencers.

Finally, we must consider the effects of sequencing errors. One way to address these errors is to encode redundant instructions that become semantic NOPs with random bit flips.

Payloads. To make payloads more robust to errors introduced by synthesis and sequencing, one may fortify payloads with error-correcting codes. Compression may be used to offset the increase in payload size and cause the sequence to be more equally distributed across the four nucleotides, avoiding issues of too much or too little GC-content.

5 Side Channel: Sample Bleeding

It is common to multiplex samples in NGS runs on modern Illumina sequencers to make better use of sequencing resources and increase throughput. This is accomplished by adding a 6-8 nucleotide index to each sample before sequencing, which is later used to demultiplex the samples. However, the demultiplexing process is not perfect. The sequence of each read is derived by sequencing a cluster of DNA on a flow cell. If clusters overlap, are seeded from multiple distinct strands, or if errors exist in sequencing the index, then the sequence of a cluster may be misassigned to an incorrect index [16]. A read assigned incorrectly will be associated with either an unused index and discarded or assigned to the index of a different sample. In the latter case, it is called *sample bleeding* or *index cross-talk*.

Illumina reports that sample bleeding occurs at a rate of 0.1%-0.2% with the type of flow cell used in this

study [24], though this continues to a topic of discussion in the sequencing community. The amount of sample bleeding depends on many factors, like index design, cluster density, sample diversity, and the underlying sequencing technology [25, 27, 33]. This situation is known to create a problem with the detection of rare genetic variants, like genetic markers for cancer [19].

The rise in outsourced sequencing at external facilities, which multiplex samples from different, untrusted sources creates opportunities for side channel attacks that are — to date — previously unconsidered by the genomic sciences. Since sample bleeding is bidirectional, an attacker could gather reads from other indices to reveal sensitive information or send data to other indices to corrupt or modify their results.

Evaluation of Data Leakage. We can leverage our sequencing results from Section 3 to better understand the security impact and amount of data leakage caused by sample bleeding. When the exploit was sequenced, it was multiplexed with seven other samples. One of these samples contained 1.5 million unique sequences, each 150 base pairs long; this sample is denoted as the target sample. With permission, we obtained the FASTQ file associated with the target sample's index after the sequences were demultiplexed. Using the two FASTQ files, one from the target sample and the other from the exploit, we sought a rough estimate of side channel effects. We note that all samples were sequenced using 6 nucleotide indices, so the sample bleeding rate may be higher than other configurations, like 8 nucleotide indices.

We assume that only the exploit sequence is attacker controlled and that attackers receive only demultiplexed results from the index of the exploit sample. To analyze their ability to pull information from other indices, we examined misassigned reads associated with the target sample in the exploit FASTQ file. There were 112 reads that aligned to sequences that came from the target sample. Two of them originated from the same sequence, so a total of 111 unique, 150 base pair sequences were leaked into the exploit FASTQ file. The quality of these reads was high; 68 of them were a perfect match (60.7%), and 103 had an edit distance of less than 2 (92.0%). Of the 235 million bases represented in the target sample, 16,521 were recoverable in the exploit FASTQ file — for context, the human genome contains around 3.2 billion bases — and, in total, 0.007% of the data was recoverable from the target sample.

If we now consider the sample bleeding side channel in the reverse direction, an attacker could modify the results that appear in other demultiplexed samples. The exploit sample contains many copies of the same short sequence. Thus, any sample bleeding from the exploit sample into the target sample resembles an attacker trying to inject a single sequence into the target FASTQ file. The exploit sequence was found 37 times in the target FASTQ file (30 times with no errors).

Hypothetical Attacks. Now that we have established sample bleeding as a source of information leakage, we propose attacks that leverage this side channel.

An attacker could use sample bleeding to inject specific DNA sequence reads into concurrently sequenced samples. These reads could contain malicious code or be used to confuse subsequent downstream analysis (e.g., variant calling).

Any reads which bleed from other samples into the attacker's sample could reveal sensitive information, like the identity of those samples. Even low levels of only a few reads could identify the species of a sample, which could be commercially sensitive in domains like drug discovery.

Another risk of multiplexing, similar to sample bleeding, is that an attacker may be able to sabotage an entire sequencing run. Most next-generation sequencers are calibrated to sequence biological DNA; they expect to see close to a 1:1:1:1 ratio of A:C:G:T. If one of the samples has low-diversity (a homogenous DNA sample), the read quality will suffer for all samples, and in extreme cases, the run could fail altogether. This could be induced with a high-concentration of the same sequence. Previous experiments by this group showed that if identical sequences compose more than roughly 25% of the total DNA, run quality deteriorated.

Summary. The read errors we encountered while developing the exploit in Section 3 caused us to reflect upon their origin, meaning, and implications. While the genome sciences community has measured rough estimates of sample bleeding, ours may be the first research to consider bleedover from an adversarial perspective and ask, for example, how *much* information is leaked and whether it is possible to push specific data into another party's sequencing files.

6 Software Security Analysis

Having evaluated the potential security threats for maliciously crafted synthetic DNA in Sections 3-4, as well as information leakage channels in Section 5, we now evaluate the software security practices of the larger bioinformatics pipeline. Specifically, we evaluate the security practices of common NGS programs to better understand the risks of DNA-based or other exploits in the real analysis pipeline. Although used broadly by biology researchers, many of these programs are written by small research groups and thus have likely not been subjected to serious adversarial pressure. We therefore hypothesize that the rate of serious vulnerabilities will be higher here than in more mature software (e.g., Internet services).

Category	Program	Version	Lines of Code	Normalized Count (Total Count)					
				strcat	strcpy	sprintf	vsprintf	gets	static buffers
NGS Analysis									
Preprocessing	fastx-toolkit	0.0.14	3,189	0.314 (1)	0.314 (1)	0 (0)	0 (0)	0 (0)	14.425 (46)
	fqzcomp	4.6	2,066	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	23.233 (48)
Alignment	bowtie2	2.2.9	58,377	0 (0)	0 (0)	0 (0)	0 (0)	0.017 (1)	3.272 (191)
	bwa	0.7.15	13,496	1.926 (26)	2.223 (30)	0.222 (3)	0 (0)	0 (0)	10.966 (148)
	hisat2	2.0.5	80,930	0 (0)	0 (0)	0 (0)	0 (0)	0.012 (1)	2.508 (203)
	STAR	2.5.2b	14,760	0 (0)	0.136 (2)	0.271 (4)	0 (0)	0 (0)	3.388 (50)
De novo assembly	MIRA	4.0.2	69,853	0.014 (1)	0.115 (8)	0.115 (8)	0 (0)	0 (0)	1.904 (133)
	velvet	1.2.10	22,794	1.228 (28)	2.106 (48)	1.185 (27)	0 (0)	0 (0)	2.588 (59)
	SOAPdenovo2	2.04-r240	37,010	0 (0)	0.351 (13)	3.161 (117)	0 (0)	0 (0)	4.945 (183)
Alignment processing	samtools	1.5	56,979	0.351 (20)	0.228 (13)	0.509 (29)	0 (0)	0 (0)	3.247 (185)
	bcftools	1.5	77,707	0.090 (7)	0.283 (22)	0.360 (28)	0 (0)	0 (0)	4.375 (340)
RNA-seq	cufflinks	2.2.1	68,539	0.058 (4)	0.817 (56)	1.984 (136)	0.029 (2)	0 (0)	4.844 (332)
ChIP-seq	PeakSeq	1.3	6,806	0.147 (1)	3.967 (27)	3.526 (24)	0 (0)	0 (0)	7.787 (53)
Control Programs									
Web server	nginx	1.11.19	80,905	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	3.411 (276)
	httpd	2.4.25	173,376	0.04 (7)	0.19 (33)	0.052 (9)	0 (0)	0 (0)	3.611 (626)
	php	7.1.1	637,921	0.003 (2)	0.022 (14)	0.011 (7)	0.002 (1)	0 (0)	5.632 (3593)
DNS server	bind	9.9.10b1	255,708	0.055 (14)	0.223 (57)	0.395 (101)	0.004 (1)	0 (0)	7.426 (1899)
Remote shell	openssh-portable	7.4p1	89,403	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	6.264 (560)
	mosh	1.2.6	12,228	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	7.933 (97)
File copying	rsync	3.1.2	39,446	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	6.718 (265)
FTP	vsftpd	3.0.3	16,414	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	2.437 (40)
Database	postgres	9.6.1	784,516	0.088 (69)	0.312 (245)	0.454 (356)	0 (0)	0 (0)	9.964 (7817)
Packet processing	tcpdump	4.9.0	73,711	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	19.726 (1454)

Table 1: Insecure buffer overflow signatures for NGS analysis (top half) and control programs (bottom half). The counts reported are the number of lines containing the corresponding insecure function call or static buffer declaration. Each count is normalized by the number of appearances per 1000 lines of code. `scanf` is not included because it was not present in any program.

Program Selection. Many commonly used, open source analysis programs are written in unsafe languages, like C and C++, known to be vulnerable to buffer overflow attacks. To quantify the risk of buffer overflows in NGS analysis programs, we evaluated 13 programs that operate at different stages of the analysis pipeline (see Table 1). To generate the list of programs in a systematic manner, we choose 6 analysis categories: (1) preprocessing, (2) alignment, (3) *de novo* assembly, (4) alignment processing, (5) RNA-seq, and (6) ChiP-seq. We required at least one program from each category. We searched for programs that were open source and written in either C or C++. To ensure that all of these programs were actively used by biologists, we required that they be available as packages in the Galaxy bioinformatics workflow system (a popular web-based analysis platform) or be part of a major effort, like the ENCODE project or the assembly of the great panda genome [14, 21, 36]. Many of them, including `bwa`, `bowtie2`, and `samtools`, come installed on current Illumina sequencers. The one exception was the `fqzcomp` program, which we included because we used it earlier in Section 3. We shared our findings about these programs with their maintainers in the hope of raising their security mindfulness. Our discussions with them confirmed that many had not considered the security of their software.

Analysis Approach. We evaluated the risk of buffer overflow attacks in these programs by using the recommendations of the OWSAP buffer overflow review guide [29]. It suggests removing insecure C library function calls and checking static buffers and print format strings. To quantify this, we counted the number of lines containing commonly misused, insecure function calls (`strcat`, `strcpy`, `sprintf`, `vsprintf`, `gets`, and `scanf`) and static buffer declarations. We derived these counts using the clang-query tool, which searches the abstract syntax tree generated by the clang C and C++ compiler. We analyzed only those files compiled using the default build. Function calls and buffer declarations in headers were also counted if they were included in code files, but they were ignored if they were in standard library headers (like the C standard lib or Boost library). For comparison, we also computed these same metrics for 10 control programs. For these, we chose programs that were Internet connected and likely to have already received adversarial pressure. Again, we included programs from 7 different categories and only considered open source programs written in C or C++.

Analysis Results. The most common insecure functions in both the NGS and control programs were `strcat`, `strcpy`, and `sprintf`. The others were used infrequently, and `scanf` was not present in any program. The `gets` function appeared once in two NGS programs;

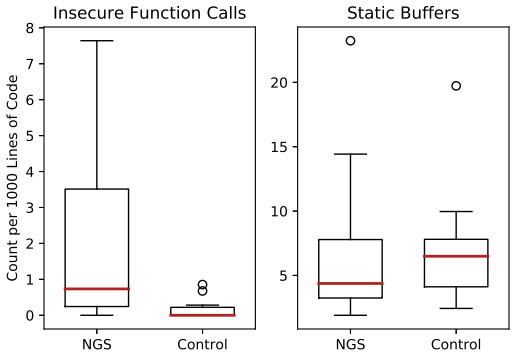

Figure 4: A box plot with the average number of insecure function calls (left) and number of static buffer declarations (right) in each program. Programs are separated into their corresponding type (NGS or control) and all counts are normalized (count / 1000 lines of code).

this is notable because `gets` is an especially insecure function that cannot do bounds checking, which is why it was removed from the 2011 C standard [1]. Overall, there was more insecure function usage in the NGS programs (Figure 4), with an average of 2.005 insecure function calls present per 1000 lines of code (sd=2.299) but only 0.185 in the control programs (sd=0.304) — an 11-fold difference. Using a two-tailed t-test, this difference was found to be statistically significant (p=0.027).

We hypothesized that there may be more static buffer declarations in the NGS programs due to poor coding practices, but there did not appear to be a difference. The NGS programs had an average of 6.729 buffer declarations per 1000 lines of code (sd=5.925), and the control programs had a similar average of 7.312 (sd=4.674). This difference was not statistically significant (p=0.809). These results are only heuristics for buggy code, but the high prevalence of insecure function calls in NGS programs provides evidence that the NGS analysis pipeline does not adhere to security best practices.

A Deeper Dive. To delve deeper into the security of the NGS pipeline, we next looked for vulnerabilities in the 13 programs. To identify them, we compiled each NGS program with the HP Fortify static code analyzer, which generates reports that include possible vulnerabilities [11]. We also manually inspected code for the insecure C library calls we noted previously. We quickly identified buffer overflow vulnerabilities in three of the NGS programs (`fastx-toolkit`, `samtools`, and `SOAPdenovo2`) and designed inputs that targeted these vulnerabilities to overflow buffers and crash programs (Figure 5). These vulnerabilities are described below:

- `fastx-toolkit`. This utility generates aggregate statistics on FASTQ files. It places aggregate results in a static array that is 2,000 bases long, and any reads longer than this will overflow the buffer. A check ensures that the read length does not exceed a limit; however, an incorrect limit of 25,000 was used by mistake. Fittingly, a comment next to the overflowable, static buffer says, "that's pretty arbitrary... should be enough for now."
- `samtools`. This program post-processes DNA read alignment files. In code that parses the header string of an alignment file (SAM file), it places the parsed header into the same buffer as the original unparsed header, which normally shrinks the result. However, if the header is malformed, then the parsed header grows larger than the original and will overflow the buffer.
- `SOAPdenovo2`. This large, *de novo* genome assembler parses reads in a FASTQ file and writes them into a static buffer that is 5,000 characters long. Any reads longer than 5,000 bases will overflow the buffer.

Given that the security risks of buffer overflow vulnerabilities are well known, we did not consider it within the scope of this paper to convert any of these vulnerabilities into working exploits. The aim here, to identify these three vulnerabilities and the construction of the crashing inputs, was straightforward. Thus, we suspect that these types of vulnerabilities are common.

These results have implications beyond direct DNA-based exploits, which we return to in Section 7. Foreshadowing that discussion, NGS data is commonly shared in large biological data repositories, making them a possible vector for spreading malicious files. There are also publicly available, remote servers, controlled and managed by 3rd parties, where users can upload and process data using these or similar programs.

Ethics and Disclosure. Numerous software developers and users are involved in the bioinformatics pipeline at large. Our findings are not specific to any single entity in this space, but rather apply broadly, across the industry as a whole. We have notified the authors of potential issues to the specific software packages that we analyzed, but we stress that many other software packages likely share similar types of vulnerabilities.

7 Discussion

Our results, and particularly our discovery that bioinformatics software packages do not seem to be written with adversaries in mind, suggest that the bioinformatics pipeline has to date not received significant adversarial pressure. We thus consider it critical — both as a research contribution and as a contribution to the broader community — to reflect upon a threat model for the next-gen sequencing pipeline. A concrete threat model can

```
#define MAX_SEQ_LINE_LENGTH (25000)
...
#define MAX_SEQUENCE_LENGTH (2000) //that's pretty arbitrary... should be enough for now
...
struct cycle_data cycles[MAX_SEQUENCE_LENGTH];
...
while ( fastx_read_next_record(&fastx) ) {
    if (strlen(fastx.nucleotides) >= MAX_SEQ_LINE_LENGTH)
        errx(1, "Internal error: sequence too long (on line %llu). Hard-coded max. length is %d",
             fastx.input_line_number, MAX_SEQ_LINE_LENGTH ) ;
    //for each base in the sequence...
    for (index=0; index<strlen(fastx.nucleotides); index++) {
        ....
        cycles[index].nucleotide[ALL].count += reads_count; // total counts
        cycles[index].nucleotide[nuc_index].count += reads_count ; //per-nucleotide counts
        ....
    }
```

```
// header->text is a string with the entire header
char * newtext = header->text;
...
// This is parsed incorrectly if the header
// included multiple LN:<num> in the same line
sprintf(len_buf, "LN:%d", header->target_len[tid]);
strcat(newtext, len_buf);
```

```
int gLineLen = 5000;
...
int lineLen = gLineLen;
char tmpStr[lineLen];
char * str; // = tempStr
...
memcpy ( str, &buf[p + 1], m - p - 1 );
```

Figure 5: Code fragments with buffer overflow vulnerabilities in three different NGS programs: `fastx-toolkit` (top), `samtools` (bottom left), and `SOAPdenovo2` (bottom right). Text in red highlights buggy code, and text in green denotes comments we included for clarification.

serve as a guideline for the community, encouraging the development of defenses and mitigation strategies as well as the investigation of future exploit vectors. We begin with a discussion on the future technological and market trends relevant for DNA sequencing, followed by a taxonomy of threats and directions for future defenses.

7.1 Future Trends

DNA Sequencing. The decreasing cost, the increasing throughput, and the broader deployments of DNA sequencing capabilities will expand the opportunities and motivations for attackers to target this pipeline, including important domains like forensics, medicine, and agriculture. Fundamental aspects of sequencing technology itself, such as the improving accuracy and ongoing development of long read sequencers, e.g., Oxford Nanopore Technologies [8], will radically change the structure of sequencing data.

DNA Synthesis. Another quickly improving technology is *de novo* DNA synthesis, which continues to get faster and cheaper. With novel uses of synthetically produced DNA, like DNA for data storage [2, 9, 15, 28], these improvements are expected to continue.

Wet Lab as a Service. There is increasing access to wet lab techniques and services by non-experts. New companies exist to provide customers with remote control of a wet lab through a computer (even offering wet lab "APIs") [35]. As these grow more prevalent, they will enable more actors, even those with scant laboratory experience, to attack the DNA sequencing pipeline.

Storage and Analysis. As DNA sequencing gets cheaper, the business focus will likely shift to keeping, analyzing and making use of genomic information in cloud services (e.g., Illumina's BaseSpace, Microsoft Genomics). Tools already exist to help scientists who have little programming or data science experience analyze DNA sequencing data. Notable examples include the Galaxy web analysis platform and the Broad Institute's cloud based variant calling workflow [3, 13].

7.2 Attack Surfaces

This section covers the attack surfaces that are present in the end-to-end DNA sequencing pipeline. Our exploration of this threat model focuses on exploits and is complementary to existing efforts that protect privacy in genetic computations [12, 32, 38].

Physical DNA Exploits. Sections 3-4 discussed how DNA strands themselves could be used as a vector for injecting code and data into the sequencing pipeline. To execute such an attack, an attacker could target any facility that accepts samples for sequencing and processing.

Outsourced sequencing facilities are common because next-gen sequencing machines are expensive and require expertise to operate. Many facilities even provide bioinformatics services, which means that it is not just the sequencing machine but downstream analysis utilities that

could be targeted by a DNA-based attack vector.

Another method of DNA injection is to contaminate a biological tissue sample (e.g., blood, hair, and saliva) with malicious DNA that the attacker knows will be sequenced. For example, they could send a contaminated saliva sample to a personalized genomics testing company, like Sure Genomics [34]. This method creates additional challenges because the malicious DNA sample would have to survive genomic DNA extraction and sample preparation, including DNA purification, quality controls, and library preparation.

DNA data storage services are an indirect means of DNA-based code injection; the attacker would provide digital data to be written that would be encoded and synthesized into DNA and later sequenced when read.

Multiplex Sequencing. To achieve high throughput, sequencers will continue to support high levels of sample multiplexing. However, as discussed in Section 5, sample bleeding gives a side channel to attackers that can be used to influence any concurrently sequenced samples. Therefore, it is important to consider the sources of all DNA samples when sequencing.

Analysis Services. Third party analysis service could be targeted if they process attacker controlled data with vulnerable software. Attackers could upload malicious files directly for processing (e.g., Galaxy) or send malicious data from biological instruments, like a DNA sequencer that is integrated with a cloud service (e.g., Illumina's Basespace Hub). Afterwards, the attacker would direct the analysis service to process the malicious files using a vulnerable workflow.

Shared Databases. Biological data generally, and NGS results specifically, are commonly shared and analyzed by different research teams. To facilitate this sharing, public repositories of NGS data are available for download. The NIH, the European Bioinformatics Institute, and the DNA Database of Japan maintain a large combined repository, called the Sequence Read Archive (SRA), which contains nearly 10 quadrillion bases of DNA [31]. Anyone who creates an account can submit sequencing files, which makes this an easy attack vector.

Direct sharing of biological data, including DNA sequences, could also occur directly between collaborators, e.g., via email. An adversary could also explore direct sharing as a potential attack vector.

7.3 Defenses

In this section we categorize possible defenses to help mitigate the attacks described above.

Follow Best Practices for Secure Software. Our analysis suggests that the bioinformatics software community has not received significant adversarial pressure. Hence, its software is in general not hardened against attack. Our first recommendation is therefore to encourage the widespread adoption of standard software security best practices like input sanitization, the use of memory safe languages or bounds checking at buffers, and regular security audits.

Patching is challenging because the analysis software is quite decentralized (packages are often located in individually managed repositories) and not regularly updated. One solution is to use a centralized repository to manage updates and deliver patches, similar to the APT package manager. Packages could also be signed to ensure their authenticity. In the case of file sharing, the sequencing files themselves could be signed by verified research groups before uploading them to centralized databases.

Secure Samples. In some domains, like forensics, attackers could be highly motivated to disrupt sequencing or cause mis-identification. In these cases, the biological sample should be tightly monitored from collection through sequencing. However, physical control of individual samples may not be sufficient to stop contamination because of sample bleeding, which we discuss below.

Minimize Sample Bleeding. Sample bleeding may make concurrently sequencing samples from untrusted sources risky. A simple solution is to enforce, by policy, that the sources of all samples are verified before they are sequenced together or else they are sequenced separately. A better solution is to reduce or detect sample bleeding with technical means.

The overall rate of bleeding can be reduced by preparing samples with two multiplex indices instead of one [19, 24] and by modifying the default cluster identification algorithm [25]. Another approach is to detect misassigned reads by cross-aligning samples against one another, and any found could be removed by the sequencer before returning the demultiplexed files. We encourage future research to minimize this side channel.

Detect Shellcode before Synthesis. Regulations already exist to prevent the synthesis of a known, dangerous DNA sequence. For example, DNA synthesizers are required to verify that it is not synthesizing biological viruses, like chicken pox [4–6]. While this approach works well when detecting known dangerous sequences, it could prove difficult to detect arbitrary DNA shellcode because general shellcode detection has proved difficult in other domains. For example, shellcode can be converted into syntactically correct English [23]. However, we still encourage researchers to find creative strategies that detect executable code in DNA.

8 Conclusions

Significant advances in DNA synthesis, DNA sequencing, and genomic sciences derive from tools and techniques not previously scrutinized for security robustness. We conducted a broad security analysis of the DNA processing pipeline, including a study of the feasibility of synthesizing DNA capable of compromising a computer program (Sections 3-4), a study of information leakage and information injection side-channels during the sequencing process (Section 5), and a study of the general software security practices in DNA processing software (Section 6). To our knowledge, ours is the first effort to broadly consider this pipeline, and the first to demonstrate a DNA-based exploit. Informed by our results, we presented lessons for this field, which has yet to receive adversarial pressure. We strongly encourage additional research before such adversarial pressure manifests.

Acknowledgements

This research was supported in part by the University of Washington Tech Policy Lab, the Short-Dooley Professorship, and the Torode Family Professorship. We thank Sandy Kaplan, R.B. Kohno, Paul Ney, Jay Shendure, Anna Kornfeld Simpson, Lok Yan, and the members of the Molecular Information Systems Lab and the Security and Privacy Research Lab for helpful discussions and comments on earlier drafts of this paper.

References

[1] *GETS(3) Linux Programmer's Manual. Janurary 2012*.

[2] BORNHOLT, J., LOPEZ, R., CARMEAN, D. M., CEZE, L., SEELIG, G., AND STRAUSS, K. A DNA-based archival storage system. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2016), ASPLOS '16, ACM, pp. 637–649.

[3] Broad Institute and Google Genomics. `https://www.broadinstitute.org/google`. Accessed: 2017-06-28.

[4] Code of Federal Reguations, "POSSESSION, USE, AND TRANSFER OF SELECT AGENTS AND TOXINS", title 7, section 331. `http://www.ecfr.gov/cgi-bin/text-idx?c=ecfr&tpl=/ecfrbrowse/Title07/7cfr331_main_02.tpl`.

[5] Code of Federal Regulations, "POSSESSION, USE, AND TRANSFER OF SELECT AGENTS AND TOXINS", title 9, section 121. `http://www.ecfr.gov/cgi-bin/text-idx?tpl=/ecfrbrowse/Title09/9cfr121_main_02.tpl`.

[6] Code of Federal Regulations, "SELECT AGENTS AND TOXINS", title 42, section 73. `http://www.ecfr.gov/cgi-bin/text-idx?tpl=/ecfrbrowse/Title42/42cfr73_main_02.tpl`.

[7] CHECKOWAY, S., MCCOY, D., KANTOR, B., ANDERSON, D., SHACHAM, H., SAVAGE, S., KOSCHER, K., CZESKIS, A., ROESNER, F., AND KOHNO, T. Comprehensive experimental analyses of automotive attack surfaces. In *Proceedings of the 20th USENIX Conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association.

[8] Oxford Nanopore Technologies. `http://nanoporetech.com`. Accessed: 2017-06-28.

[9] CHURCH, G. M., GAO, Y., AND KOSURI, S. Next-generation digital information storage in DNA. *Science 337*, 6102 (2012), 1628–1628.

[10] CLELLAND, C. T., RISCA, V., AND BANCROFT, C. Hiding messages in DNA microdots. *Nature 399*, 6736 (1999), 533–534.

[11] Fortify static code analyzer. `http://www8.hp.com/us/en/software-solutions/static-code-analysis-sast/`. Accessed: 2017-02-15.

[12] FREDRIKSON, M., LANTZ, E., JHA, S., LIN, S., PAGE, D., AND RISTENPART, T. Privacy in pharmacogenetics: An end-to-end case study of personalized warfarin dosing. In *USENIX Security Symposium* (2014), pp. 17–32.

[13] Galaxy platform. `https://usegalaxy.org/`. Accessed: 2017-06-28.

[14] Galaxy tool shed. `https://toolshed.g2.bx.psu.edu/`. Accessed: 2017-02-15.

[15] GOLDMAN, N., BERTONE, P., CHEN, S., DESSIMOZ, C., LEPROUST, E. M., SIPOS, B., AND BIRNEY, E. Towards practical, high-capacity, low-maintenance information storage in synthesized DNA. *Nature 494*, 7435 (2013), 77–80.

[16] HADFIELD, J. Index mis-assignment to Illumina's PhiX control. `http://core-genomics.blogspot.com/2016/10/index-mis-assignment-to-illuminas-phix.html`. Accessed: 2017-02-15.

[17] Breakthrough resolution. overcome limitations. solve more cases. `https://www.illumina.com/areas-of-interest/forensic-genomics.html`. Accessed: 2017-02-16.

[18] Is this the biggest threat yet to illumina? `https://www.fool.com/investing/general/2016/03/18/is-this-the-biggest-threat-yet-to-illumina.aspx`. Accessed: 2017-06-01.

[19] KIRCHER, M., SAWYER, S., AND MEYER, M. Double indexing overcomes inaccuracies in multiplex sequencing on the illumina platform. *Nucleic acids research* (2011), gkr771.

[20] KOSCHER, K., CZESKIS, A., ROESNER, F., PATEL, S., KOHNO, T., CHECKOWAY, S., MCCOY, D., KANTOR, B., ANDERSON, D., SHACHAM, H., AND SAVAGE, S. Experimental security analysis of a modern automobile. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2010), SP '10, IEEE Computer Society, pp. 447–462.

[21] LI, R., FAN, W., TIAN, G., ZHU, H., HE, L., CAI, J., HUANG, Q., CAI, Q., LI, B., BAI, Y., ET AL. The sequence and de novo assembly of the giant panda genome. *Nature 463*, 7279 (2010), 311–317.

[22] LIAN, W., SHACHAM, H., AND SAVAGE, S. Too lejit to quit: Extending JIT spraying to ARM. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015* (2015), The Internet Society.

[23] MASON, J., SMALL, S., MONROSE, F., AND MACMANUS, G. English shellcode. In *Proceedings of the 16th ACM conference on Computer and communications security* (2009), ACM, pp. 524–533.

[24] Minimize index hopping in multiplexed runs. `https://www.illumina.com/science/education/minimizing-index-hopping.html`. Accessed: 2017-06-28.

[25] MITRA, A., SKRZYPCZAK, M., GINALSKI, K., ROWICKA, M., AND OUDEJANS, C. Strategies for achieving high sequencing accuracy for low diversity samples and avoiding sample bleeding using illumina platform. *PLoS One 10*, 4 (2015).

[26] ONE, A. Smashing The Stack For Fun And Profit. *Phrack 49* (1996).

[27] OREGON STATE UNIVERSITY. Illumina barcodes. `http://cgrb.oregonstate.edu/core/illumina-hiseq-`

3000/illumina-barcodes. Accessed: 2017-02-15.

[28] ORGANICK, L., ANG, S. D., CHEN, Y.-J., LOPEZ, R., YEKHANIN, S., MAKARYCHEV, K., RACZ, M. Z., KAMATH, G., GOPALAN, P., NGUYEN, B., TAKAHASHI, C., NEWMAN, S., PARKER, H.-Y., RASHTCHIAN, C., STEWART, K., GUPTA, G., CARLSON, R., MULLIGAN, J., CARMEAN, D., SEELIG, G., CEZE, L., AND STRAUSS, K. Scaling up DNA data storage and random access retrieval. *bioRxiv* (2017).

[29] OWASP. Reviewing code for buffer overruns and overflows. https://www.owasp.org/index.php/Reviewing_Code_for_Buffer_Overruns_and_Overflows. Accessed: 2017-02-15.

[30] PACIFIC BIOSCIENCES OF CALIFORNIA. Smrt sequencing: Read lengths. http://www.pacb.com/smrt-science/smrt-sequencing/read-lengths/. Accessed: 2017-02-16.

[31] Sequence Read Archive. https://trace.ncbi.nlm.nih.gov/Traces/sra/. Accessed: 2017-02-16.

[32] SHI, X., AND WU, X. An overview of human genetic privacy. *Annals of the New York Academy of Sciences 1387*, 1 (2017), 61–72.

[33] SINHA, R., STANLEY, G., GULATI, G. S., EZRAN, C., TRAVAGLINI, K. J., WEI, E., CHAN, C. K. F., NABHAN, A. N., SU, T., MORGANTI, R. M., ET AL. Index switching causes spreading-of-signal among multiplexed samples in illumina hiseq 4000 dna sequencing. *bioRxiv* (2017), 125724.

[34] Sure Genomics. http://www.suregenomics.com/. Accessed: 2017-06-28.

[35] The automated lab. https://www.nature.com/news/the-automated-lab-1.16429. Accessed: 2017-06-28.

[36] UC SANTA CRUZ. Software tools used to create the ENCODE resource. https://genome.ucsc.edu/ENCODE/encodeTools.html. Accessed: 2017-02-15.

[37] UNTERGASSER, A., CUTCUTACHE, I., KORESSAAR, T., YE, J., FAIRCLOTH, B. C., REMM, M., AND ROZEN, S. G. Primer3 - new capabilities and interfaces. *Nucleic acids research 40*, 15 (2012), e115–e115.

[38] WANG, S., BONOMI, L., DAI, W., CHEN, F., CHEUNG, C., BLOSS, C. S., CHENG, S., AND JIANG, X. Big data privacy in biomedical research. *IEEE Transactions on Big Data* (2016), 1–1.

[39] WETTERSTRAND, K. Dna sequencing costs: Data from the nhgri genome sequencing program (gsp). http://www.genome.gov/sequencingcostsdata. Accessed: 2017-02-16.