CHAPTER

String analysis for cyber strings 8

W. Casey

8.1 STRING ANALYSIS AND CYBER DATA

A *string* is any sequence of symbols that is interpreted to represent a precise meaning. Written language, including this sentence, provides strings in which informational messages may be expressed as a sequence of words (each of which is a sequence of letters). However, natural languages such as English may give rise to ambiguous meaning. For example, consider the following statement: "Time flies like an arrow; fruit flies like a banana." In computational settings, it's important that strings (along with their encoding and interpretation) have discrete, precise meanings. Formal languages provide the general backdrop for our discussion of strings.

A central goal when analyzing cyber data is to seek a string representation for the problem's objects so that similarities in their string representations will provide a meaningful result for the analysis problem at hand. To emphasize this point, we consider signature based detection of cyber attacks and how the problem of determining safety (or that a system may be compromised) may be considered by the analysis of strings. We set the stage by providing a general background of cyber data and analysis techniques, followed by our historical examples. Then we focus the remainder of the chapter on common contemporary techniques used to analyze cyber sequential or string data.

8.1.1 CYBER DATA

Many different types of data arise from cyber security scenarios—here we will identify a few prominent data types and outline an organizational framework for thinking about cyber data. Generally, within cyber security scenarios the objects studied may or may not have much known about them. One way to think about information (known and unknown) for digital objects will be similar to that of physical objects, such as an antique. An antique is affected by a *provenance*, or a history of events, which affect its state. In the real world even a valuable historical object may have partial or disputed information concerning its provenance. For digital objects we consider provenance similarly; there can also be incomplete

Cybersecurity and Applied Mathematics. http://dx.doi.org/10.1016/B978-0-12-804452-0.00008-7

or partial awareness concerning the origin or histories of data objects (ie, files, programs, configuration settings, etc). With the notion of provenance in mind we may consider major types of cyber data.

The three main forms of data we consider are static, dynamic, and behavioral.

A *static* data analysis will focus on objects such as files, system configuration parameters, and programs (specified in a programming language or machine executable). This type of analysis may seek to identify evidence that the security of a system was breached or that a particular program is capable of breaching the system either as a direct effort (ie, malware) or indirectly (ie, a vulnerability) if attacked in a certain way. A computer system is comprised of thousands of programs and libraries, and for each of them a cyber security operator or analyst may only have a small amount of information concerning its provenance, so provenance information is usually thin thereby yielding advantages to attackers who may camouflage malware within the context of limited awareness (ie, the many system files and what they do).

A *dynamic* analysis will focus on data generated within a computer system when certain stimuli or inputs are provided. One form of dynamic data is a system log file, which contains metadata concerning the operation of various components. For example, we may monitor how a Microsoft system registry database changes before and after a given program is executed, and likewise what (if any) files are created as a consequence of executing a given program by evaluating logs or designing our own system monitors. Another and more interactive example arises when monitoring network traffic and identifying problematic communications (possibly to known command and control botnet servers). Still another example is fuzz testing in which a large variety of stimuli is provided to a program or library to find fault conditions, which may prove to be a software vulnerability. A dynamic analysis may be realized either as a system monitor or as an experiment in which the cyber analyst has created meaningful ways to observe system states.

A *behavioral* analysis will also be focused on dynamic data such as log and monitor data, but the focus of behavioral analysis will be to consider the sequences of actions and events as expressions of behaviors. Therefore, this also includes some model of behavior (such as baseline and anomalous behaviors). One type of behavioral analysis will include program tracing and a statistical model for learning a common behavior of a malware group, which distinguishes it from benign software or other malware groups. In this way, the behavioral data may include both the trace outputs as well as the model which describes a particular behavior.

8.1.2 MODES OF ANALYZING CYBER DATA

One common operational mode of analysis is a *forensic analysis* which usually takes place after an event, for example a data breach, to investigate what happened. The mode of forensic analysis is similar in nature to that of a crime scene, where an investigator and focuses on the artifacts left behind in order to gain some understanding of key questions, for example attribution (ie, who initiated the attack).

Another mode of analysis is *experimental*, often testing an object to identify how it compares to a reference data set. These types of analysis are commonly done for artifacts of unknown provenance to test if they are malware or contain vulnerabilities. Still another more operational mode is *online analysis*, and this may be thought of as a filter pipeline where actions are tested against a set of signatures in real time, with the possible outcome that a signature match may invoke a response to keep the system safe. Examples of this include network filters.

Another type of analysis mode is *formal methods and verification* and this approach is more logical in that it considers how programs or data is to be interpreted by the system and will attempt to compute or verify that certain unsafe states are not reachable. This type of analysis often employs computational processes to prove various properties of the software artifact.

Generally, in practice, various types of cyber data and analytical modes are mixed in ways to provide the best approaches for the problem at hand. In order to introduce the area of string analysis in cyber data we have selected a few of the most primitive string comparison methods which often are applied to the various data forms and as a part of a variety of modes for analyzing cyber data.

8.1.3 ALPHABETS AND FINITE STRINGS

Let Σ be a finite and nonempty set of *symbols*, also called an *alphabet*. For example, $\Sigma = \{0, 1\}$ is commonly called the *binary* alphabet, and its symbols are denoted by zero and one.

Given two symbols from Σ, that is $x, y \in \Sigma$, a juxtaposition is an operation which creates a composite object: either $x \circ y$ to emphasize that x is joined to y with an ordering of x before y, or $y \circ x$ indicating that y is joined to x with an ordering of y before x. Because $x \circ y$ may not be a symbol itself, the juxtaposition operation naturally extends the alphabet to a larger set of objects which can be created by juxtaposition between these objects or alphabet members. The limiting set of all objects created from an arbitrary number of juxtaposition operations, along with either alphabet members or other objects so created, will be the set of all strings over Σ. Objects that are created with juxtaposition can themselves be juxtaposed with other such objects, and the resulting the closure represents all such possible outcomes of objects or strings.

Following in this way, we provide a theoretical method to construct recursively the set of all strings over Σ. We will do this by creating an set of objects called Σ^k, which is the set of all strings constructible from Σ with $k - 1$ juxtapositions. We let $\Sigma^1 = \Sigma$, and next, we describe the recursion:

$$\Sigma^k = \{x \circ y | x \in \Sigma^{k-1} \text{ and } y \in \Sigma\}.$$

Notice that Σ^2 is the set of all strings created by one juxtaposition, and it includes all such single juxtapositions over all possible pairs of symbols. From the recursive form

above, we see that Σ^k is constructed from $k-1$ juxtapositions. An inductive proof would check the base case and notice that the recursive equation juxtaposes $x \in \Sigma^{k-1}$ to a symbol. Therefore there is one greater juxtaposition than the number used to create objects in Σ^{k-1}, which we can take as $k-2$ (ie, the inductive hypothesis), thus adding together to show our claim that objects in Σ^k are created with $k-1$ juxtapositions of symbols from Σ.

Given a string x (ie, an object created from juxtaposition), the *length* of string x, denoted $|x|$, may be defined as the first $k : x \in \Sigma^k$. Said differently, the string's *length* is the number of symbols comprising the string.

The *Kleene closure*, denoted Σ^*, is defined as

$$\Sigma^* = \cup_{k=0}^{\infty} \Sigma^k,$$

and it includes all finite strings over Σ.

A string over Σ may also be thought of as an ordered list of symbols. Here, the order of symbols is represented by natural number indices (denoted with a subscript). A string s of length n (ie, $|s| = n$) can list out the ordered sequence of symbols as $s = s_1 s_2 \ldots s_k \ldots s_n$, with the kth symbol denoted as $s_k \in \Sigma$ for $k \in \{1, 2, \ldots, n\}$.

Letting λ, ω be strings over Σ, we may make explicit the outcome of juxtaposition also called *concatenation*:

$$\lambda \circ \omega = \underbrace{\lambda_1 \lambda_2 \ldots \lambda_{|\lambda|}}_{\lambda} \underbrace{\omega_1 \omega_2 \ldots \omega_{|\omega|}}_{\omega}.$$

The kth symbol of $\lambda \circ \omega$ can be computed from the indices of λ and ω by considering the shifting of indices needed to join the head ω to the tail of λ. Letting $\phi = \lambda \circ \omega$, the kth symbol is:

$$\phi_k = \begin{cases} \lambda_k \text{ if } k \leq |\lambda| \\ \omega_{k-|\lambda|} \text{ if } k > |\lambda|. \end{cases}$$

For most string implementations and pseudo code, we use indices of nonnegative integers starting from 0 used for the first symbol of each string. Therefore, for a string of length k, the last symbol is referenced by the index $k-1$. For a string s, we refer to the jth element as $s[j]$. A substring is a contiguous region of the sting denoted as $s[j : k]$ to indicate the string of symbols $s[j] \circ s[j+1] \circ \cdots \circ s[k]$, meaning that it is inclusive of indices on both ends. The notation $s[j : k)$ indicates the substring $s[j : (k-1)]$ which is not inclusive of k. A *prefix* of s is any substring of the form $s[0 : j]$, for $j \in \{1, 2, \ldots |s|\}$. A *suffix* is any substring of the form $s[k : |s|)$.

8.1.4 FORMAL LANGUAGES

The concatenation operation provides a way to construct larger strings from smaller strings. When we consider how to complete the concatenation operation on strings,

we also consider the empty string, denoted as ϵ, which serves as both a left and right *idempotent* for concatenation. An idempotent leaves the object unchanged; therefore the following equations hold for all strings λ, ω:

$$\lambda \circ \epsilon = \lambda.$$
$$\epsilon \circ \omega = \omega.$$

Languages and regular expressions

A *language* over Σ is any subset of Σ^*. A *language recognizer* is a function, which given a string, decides (yes/no) if the string belongs to a certain language. Theoretical computer science is interested in the types of computing machines necessary to identify languages. One of the simplest machine models is the FSA, which may be thought of as a finite set of graph vertices called *states*; one of these vertices will have a special designation as a *start state*, and any subset of the vertices will also be designated as *accept states*. In addition, there is a *state transition rule* which is a set of labeled directional edges emanating from each state. To simplify the discussion, we continue to describe a deterministic finite automata (DFA), which is a type of FSA simpler than the non-deterministic variety. For the DFA we use an alphabet Σ to label edges. In a DFA each vertex is the source of exactly one directed edge labeled by each symbol of an alphabet. Given a string input, a DFA may decide (yes/no) to *accept* it as follows: We place a token on a vertex designated as the start state. Given a string s, we inspect one symbol at a time and apply the following rule to move the token to the next vertex: starting from the vertex with the token, we use the edge labeled with the current symbol to move the token to the next vertex (the edge is directed). Finally, after the last symbol of the string, we accept the string as a member of the language if the last vertex transitioned to is designated as an accept state.

Although simple and limited in memory (by the set of states), a DFA machine is able to recognize a language as the subset of all strings that it accepts. The types of languages that DFAs recognize turn out to be the same class as the non deterministic variety of FSA recognize, and are called the *regular languages*. Regular languages are well known to computer users who have used simple wild card substitutions (eg, `'ls *.py'` to list all the Python programs in a directory) or formed complicated *regular-expression*–based manipulation (eg, `sed 's/\(0[Xx][0-9A-Fa-f]\)/HEX-ADDRESS-FOUND:\1/g`). Known more commonly by their regular expression representations, the regular languages are a powerful and useful tool for computing. The usefulness of regular expressions is due to how they can be efficiently compiled into a DFA matcher. It is also possible to show an equivalence between regular languages, DFA recognizers, and regular expressions, therefore indicating that these are three ways of discussing the same thing. Regular expression matching is computationally very efficient and contributes to the underlying model for most malware signature-detection methods.

8.2 DISCRETE STRING MATCHING

8.2.1 HASHING

A *hash function* is a deterministic function which projects a domain of strings (or keys) into a range of bounded integers. The domain of a hash function is the space of all strings. However, with the use of an object's serialization function, we may extend the domain of a hash function to include objects, data structures, or, generally, anything which admits to an unambiguous string representation. Originally developed for information retrieval algorithms such as a hash table implementations,[1] the properties which make hash functions effective have become well known and useful for a number of computational tasks.

A particularly important consideration for a hash function concerns the distribution of range values for a randomly sampled set of keys. In particular, a hash function is said to be a *universal hash function* when the hash values of random samples tend to distribute uniformly over the range space. Notice also that the domain of a hash function is the set of all strings (infinite), and the range is a bounded set of integers (finite); therefore, the hash function will also be many to one. A *hash collision* occurs when two distinct key values map to the same integer value, and when a collision occurs in a hash table, extra steps will be required. Hash functions that are *universal* are very useful in information retrieval tasks because they can be analyzed probabilistically to understand the likelihood of hash collisions.[2] Despite the possibility of collisions, hash functions have enormous practical use because they can be computed efficiently, and in order to deal with collisions, the designer can select parameters of the hash function, the table, and the collision resolution strategy in order to tailor the efficient hashing techniques to particular problems.

Closely related to uniform hash functions are *cryptographic hash functions*, which are functions designed to be difficult to invert. A type of cryptoanalysis technique called *frequency analysis*, which studies a cryptographic function statistically, can be used to find the weaknesses of a cryptographic hash function. Therefore, in order to be secure, a cryptographic hash function must also be a *universal hash function*. Cryptographic hashes such as MD5, SHA256, and SHA1 are commonly used to perform integrity checks of communicated messages. The wide-scale use of cryptographic hash functions has also taken root in the security community where hash functions are also used to identify artifacts. It is common to refer to malware, binary artifacts, and library and reference objects by the 32 hex digits comprising its MD5 sum.

[1] A hash table can be used to implement an associative array for organizing a key-value store where a user may wish to add and remove data dynamically.
[2] A hash collision occurs when two keys map to the same output value.

Birthday party and universal hash function collisions

The range of a hash function is a finite (bounded) set of integers R, but the domain is an infinite set of strings. Therefore hash collisions are possible, and among a set of n objects, there is some probability that any two of them will have a common hash value. For example, if n is greater than $|R|$, a hash collision is guaranteed (eg, with probability 1) by the pigeon hole principle. Given a universal hash function, the question of a collision's likelihood among n objects arises naturally. This problem is widely known in probability as *the birthday problem*. Among n peoples we ask what is the probability that any two of them have the same birthday? A related question is how many people n do we need before the probability that two birthdays collide exceeds a certain value?

To consider the possibilities of a hash collision for universal hash functions we ask the same question for a set of n keys, or strings, or binary artifacts. Lets fix MD5 as the universal hash function under consideration. Under the assumption that the function is universal (ie, the outputs are uniformly distributed), the probability that any particular string hashes to a given value may be considered to be $\frac{1}{2^{64}}$. The probability that any pair of n strings hash to the same value can be considered with its complement event; that is, given n distinct strings, we wish to know the probability that the hash values are all distinct (ie, no collisions). We compute this probability in the context of universal hash functions for binary artifacts.

Let us assume that we have a random sample of n distinct binary strings (artifacts) and a universal hash function. Given an arbitrary order of binaries, let event E_k describe the event that the kth binary artifact hashes to a value different from each of the previous distinct $k-1$ hash values:

$$\text{Prob}(E_k) = \text{Prob}(E_{k-1}) \times \underbrace{\left(1 - \frac{k-1}{2^{64}}\right)}_{\text{Avoids previous values}}.$$

With the obvious base case that $\text{Prob}\, E_1 = 1$, letting R be the range of the universal hash function, we obtain the probability that, among n artifacts, all n binaries will hash to differing range values with probability:

$$\text{Prob}(E_n) = \prod_{k=1}^{n} \left(1 - \frac{k-1}{|R|}\right). \tag{8.1}$$

Additionally, we may be interested in the birthday problem; that is, for what value of n will $\text{Prob}(E_n) < \kappa$, implying that, with probability greater than κ, a collision is found. We now present an approximation method used to solve for such an n. To approximate these probabilities, we consider the product of exponents

$$P_n = \prod_{k=1}^{n} \exp\left(-\frac{k-1}{|R|}\right),$$

and we notice that the inequality $1 - x \leq \exp(-x)$, for all $x \in [0, 1]$, can be applied term by term to bound $\text{Prob}(E_n)$ by P_n:

$$\text{Prob}\,(E_n) < P_n, \text{ for every } n > 0.$$

Therefore, with this bound, $P_n < kappa$ indicates that $\text{Prob}(E_n) < \kappa$.

Notice that P_n can also rewritten with an exponential sum:

$$P_n = \exp - \left(\frac{1}{|R|} \sum_{i=1}^{n} (k-1) \right) = \exp\left(-\frac{n(n-1)}{2|R|} \right).$$

For example, the bound implies that the probability of a hash collision exceeds $\frac{1}{2}$ when n is sufficiently large that $P_n < \frac{1}{2}$ or

$$\begin{aligned} \exp\left(-\frac{n(n-1)}{2|R|} \right) &< \frac{1}{2}, \\ \left(-\frac{n(n-1)}{2|R|} \right) &< \log \frac{1}{2}, \text{ and} \\ n(n-1) &> 2|R| \log 2. \end{aligned} \tag{8.2}$$

Therefore, the minimum value of n that satisfies Eq. (8.2) also implies that a hash collision occurs with probability $\frac{1}{2}$ or greater. To approximate such a minimum value for n, we approximate the left hand side of Eq. (8.2) as n^2 and solve the following test equation:

$$n^2 = 2|R| \log(2). \tag{8.3}$$

When $|R|$ is itself a power of 2, say $|R| = 2^w$, the solution to the test equation (ie, Eq. (8.3)) is $n = 2^{\frac{w+1}{2}} \log(2)$. Therefore, the probability of a hash collision for MD5 (where $w = 64$) exceeds $\frac{1}{2}$ when $n \approx 2^{32.5} \log(2)$ or when n is around 4.2 billion objects. Starting from this value of n, we can determine more a accurate minimum value for n; however, the described bounds and approximations help us to obtain an estimate quickly.

Determining a hash collision itself may amount to an important finding for cybersecurity. For cryptographic hash functions, the ease with which a hash collision can be found or constructed may be exploited to subvert the integrity of a message. Generally, an application which uses a universal hash function will also consider the probability of collisions which is guaranteed when the input space is infinite and range values are bounded.

8.2.2 APPLICATIONS OF HASHING

With this understanding of hash functions and their inherent limitations due to hash collisions themselves due to hash functions' finite range, we next focus on how their efficiency can be used to study strings that are relevant in cybersecurity.

Hash functions are efficient when identifying matching strings. Given that data and code objects may be represented by strings, a hash function can be used to match objects as well. Take, for example, a binary executable of unknown origin as M. Often, the executable file format includes a natural division of code and data into sections, segments, or pages which we may use to represent M as a list of constituent string objects $M = [m_1, m_2, \ldots, m_K]$. Moreover these constituent strings may often be readily broken down further into data (as known data structures are simple to extract) or code (sections may be have a listing of functions making their extraction straightforward). Although further resolution of a data object into constituent objects is typically very powerful and revealing, for illustrative purposes, we need only consider the first-order division of M into constituent strings $[m_1, \ldots, m_K]$.

An adversary, aware that hashing techniques may be used to identify provenance in malware, may attempt to obscure the provenance information, but this extra effort will be costly due to the typical software pattern of code reuse. This inherent tradeoff faced by the adversary gives rise to the possibility that provenance may be recovered from binary artifacts such as malware. To detect the presence of code reuse with hash functions, we present several techniques below.

Bag of hashes, or bag of numbers

Given a universal hash function U (a mapping $\Sigma^* \rightarrow R$), and an artifact naturally divided into constituent strings $M = [m_1, m_2, \ldots, m_K]$, we may use U to derive a set called a *bag of numbers* representing the artifact as $\mathcal{R}_U(M)$:

$$\mathcal{R}_U(M) = \{U(m_1), U(m_2), \ldots, U(m_K)\}.$$

Using hash function U with range R, an object comprised of K constituent objects is therefore mapped into a set of R^K. This set, the bag of numbers, can be compared with other bags derived from reference objects. Bags can be compared with the Jaccard coefficient to measure similarity to previously known objects.

An important caveat to this analysis is the possibility of hash collisions which would introduce a false sense of similarity. For this reason it's important to understand the design goals and properties of the employed hash function U and under what conditions hash collisions become likely.

This technique may be applied in the study of portable document format (PDF) based malware. The PDF format defines a tree of constituent objects and stores these objects as streams (serialized representations). So, letting U be MD5, we can consider the bag of numbers to identify common objects across multiple PDF files. Similarly, binary executables are comprised of sections, segments, and pages; therefore, executables in a given file format may be compared using bag of numbers. Of particular interest are the strings which express the functions, procedures, and data objects within an executable. Although function specification strings may be made more difficult to extract from code, they can also be observed during runtime by monitoring the execution trace.

Normalizing bag of numbers

In many settings the use of a hash function is overly sensitive to small but unimportant modifications within a set of related strings. While we later present techniques which inherently address small string modifications, we now consider techniques which compensate for the overly sensitive universal hash functions, thus retaining the efficiency of hashing. This simple augmentation of the bag of numbers can broaden its use and also focus its precision tremendously. The technique presented here can be thought of as a simple bag of words that employs a string-rewriting function ϕ prior to the application of the hash function U. The string-rewriting function can modify strings to reduce variance either by the removal of noisy elements or the presentation of a string in a standard or canonical form in a process we refer to as *normalizing*.

Given a universal hash function U (a mapping $\Sigma^* \rightarrow R$), a string-rewriting function $\phi : \Sigma^* \rightarrow \Sigma^*$, and an artifact naturally divided into constituent strings $M = [m_1, m_2, \ldots, m_K]$, we may use U to derive a set called a *normalized bag of numbers*, representing the artifact as $\mathcal{R}_{U \circ \phi}(M)$:

$$\mathcal{R}_{U \circ \phi}(M) = \{U(\phi(m_1)), U(\phi(m_2)), \ldots, U(\phi(m_K))\}.$$

We next illustrate these techniques on two functions listed in Table 8.1 and extracted from a binary executables. The functions are specified in assembly language and are not identical, but they are structurally similar and functionally related. Letting U be the MD5 hash function, we consider two functions ϕ_{PIC} and ϕ_{RIC}, which will provide a means to compute a position-independent code (PIC) normalized bag of numbers, as well as a a register-independent code (RIC) normalized bag of words. The selected functions show a case where similarities and matching provide meaningful conclusions concerning related functions and come with few data manipulations in addition to the basic hash function.

To illustrate our string-rewriting functions ϕ_{PIC} and ϕ_{RIC}, we select the function labeled ISPUNCT, and for each line of code, we attribute positional data (second column) and register data (third column). These attributes are used to group lines of code into sets with a common attribute defining the method we use to split function into constituent parts. Further, while rewriting each line of code, we replace each item with a parameterized string (Table 8.2).

For each line of code, the function ϕ_{PIC} can be executed by identifying all hexadecimal numbers (using a regular expression) and extracting them into a buffer. The distinct extracted numbers, in left to right order, as found in the line of code, become the attribute for the line of code. Next, to rewrite the line of code, each distinct hexadecimal number is replaced with a string. For example, we replace the jth distinct hexadecimal number with string $P.j$ everywhere it occurs in the line of code. Next, the rewritten lines of code are reordered first by attributes and second by original index. Finally, the strings with a common attribute are concatenated into a constituent string. So, for example, function ISPUNCT is rewritten and grouped by attribute in Table 8.3; therefore, the function ISPUNCT will be represented by

Table 8.1 Two Functions Expressed in Assembly and Extracted From the Program "gawk"

Ispunct	Isspace
pushq %rbp	pushq %rbp
movq %rsp, %rbp	movq %rsp, %rbp
cmpl $127, %edi	cmpl $127, %edi
ja 0x100021f0f	ja 0x100021bcf
movslq %edi, %rax	movslq %edi, %rax
movq 229669(%rip), %rcx	movq 230501(%rip), %rcx
movl 60(%rcx,%rax,4), %eax	movl 60(%rcx,%rax,4), %eax
shrl $13, %eax	shrl $14, %eax
andl $1, %eax	andl $1, %eax
popq %rbp	popq %rbp
ret	ret
movl $8192, %esi ; 0x100021f0f	movl $16384, %esi ; 0x100021bcf
callq 0x10004bd7a	callq 0x10004bd7a
testl %eax, %eax	testl %eax, %eax
setne %al	setne %al
movzbl %al, %eax	movzbl %al, %eax
popq %rbp	popq %rbp
ret	ret
nopw %cs:(%rax,%rax)	nopw %cs:(%rax,%rax)

Functions ISPUNCT *and* ISSPACE *exhibit a high degree of similarity in their structure. The callq commands indicate that flow of control is transferred to differing locations where specific actions are performed, and the brevity and similarity of these functions indicate that they may be wrappers or templates. Notice there are only a few differences in lines of code, including differing jump locations at line 4 represented in hexadecimal notation.*

$[f_0, f_1, f_2]$ with f_0 being the concatenated code of the first 17 lines of Table 8.3 and refer to the lines of ISPUNCT without hexadecimal numbers. On the other hand, f_1 is line `ja P.0`, and string f_2 is the string `callq P.0`. The bag of numbers for function ISPUNCT, normalized with ϕ_{PIC}, is shown in Table 8.5.

For each line of code, the function Φ_{RIC} can be executed by identifying all register addresses and extracting them into a buffer. The distinct register addresses, in left to right order, as found in the line of code, become the attribute for the line of code. Next, to rewrite the line of code, each distinct register address will be replaced with a string: we replace the jth distinct register address with string $P[j]$ everywhere it occurs in the line of code. Next, the rewritten lines of code are reordered first by attributes and second by original index. Finally, the strings with a common attribute are concatenated into a constituent string. So, for example, function ISPUNCT is rewritten and grouped by attribute in Table 8.4; therefore, the function ISPUNCT is represented by $[f_0, f_1, f_2, \ldots, f_{13}]$, with f_0 being the concatenation of the first four

Table 8.2 The Function ISPUNCT With Attributes

	ISPUNCT (Lines of asm)	Positional Attribute	Register Attribute
1	pushq %rbp	0	{rbp}
2	movq %rsp, %rbp	0	{rsp, rbp}
3	cmpl $127, %edi	0	{edi}
4	ja 0x100021f0f	0x100021f0f	{}
5	movslq %edi, %rax	0	{edi, rax}
6	movq 229669(%rip), %rcx	0	{rip, rcx}
7	movl 60(%rcx,%rax,4), %eax	0	{rcx, rax, eax}
8	shrl $13, %eax	0	{eax}
9	andl $1, %eax	0	{eax}
10	popq %rbp	0	{rbp}
11	ret	0	{}
12	movl $8192, %esi ; 0x100021f0f	0	{esi}
13	callq 0x10004bd7a	0x10004bd7a	{}
14	testl %eax, %eax	0	{eax , eax}
15	setne %al	0	{al}
16	movzbl %al, %eax	0	{al, eax}
17	popq %rbp	0	{rbp}
18	ret	0	{}
19	nopw %cs:(%rax,%rax)	0	{cs, rax , rax}

Each line of code is attributed with various values depending on content. The positional attribute column identifies a hexadecimal address (if one exists within the line of code) or otherwise evaluates to zero. The register attribute column identifies the set of registers implemented for each line of code; these values are indicated with a set whose default value is an empty set.

lines in Table 8.4. The bag of numbers for function ISPUNCT, normalized with ϕ_{PIC}, is shown in Table 8.6.

Notice that the resulting set of integers, represented in Tables 8.5 and 8.6 as hexadecimal strings, are independent of the attribute values and indicate the bag of numbers for code-independent code groupings and register-independent code groupings.

With these two normalized bags of words described, we return to the problem of comparing related functions. In this case, ISPUNCT and ISSPACE shown in Table 8.1 are mapped to the PIC-normalized bag of numbers:

$$\begin{aligned}\mathcal{R}_{\text{MD5}\circ\phi_{\text{PIC}}}(\text{ISPUNCT}) = [&8853573ca3512634642a5f574d1df63a,\\ &2dc4564e7ac59eb6c0ab4fb9aff26bbc,\\ &9679a36898b4fd96f15936295ea146b5],\ \text{and}\\ \mathcal{R}_{\text{MD5}\circ\phi_{\text{PIC}}}(\text{ISSPACE}) = [&a612737df118f1dd73a5dd88fd78ba7e,\\ &2dc4564e7ac59eb6c0ab4fb9aff26bbc,\\ &9679a36898b4fd96f15936295ea146b5].\end{aligned}$$

Table 8.3 Address-Sensitive Regrouping of Function Contents for Hashing

ISPUNCT (Lines Organized by Position)	Positional Attribute
pushq %rbp	
movq %rsp, %rbp	
cmpl $127, %edi	
movslq %edi, %rax	
movq 229669(%rip), %rcx	
movl 60(%rcx,%rax,4), %eax	
shrl $13, %eax	
andl $1, %eax	
popq %rbp	0
ret	
movl $8192, %esi ; 0x100021f0f	
testl %eax, %eax	
setne %al	
movzbl %al, %eax	
popq %rbp	
ret	
nopw %cs:(%rax,%rax)	
ja P.0	0x100021f0f
callq P.0	0x10004bd7a

Notice the parameterized replacement of attribute values, thereby normalizing away the variability but maintaining the a structural meaning.

The PIC-normalized bags of numbers between the two functions have two numbers (out of three) in common.

We now display a shortened form of the RIC-normalized bag of numbers:

$$\begin{aligned}\mathcal{R}_{\mathrm{MD5}\circ\phi_{\mathrm{RIC}}}(\textsc{Ispunct}) = [&1636.48f1, 1179.62ff, 01f7.eb04, f377.9eaf,\\ &ef3a.26ab, 0fba.128f, ca2a.08c0, 1760.b11e,\\ &9464.ef9f, 2991.8ea5, 8eaf.2a6f, 525d.874b,\\ &54cf.176a],\ \text{and}\\ \mathcal{R}_{\mathrm{MD5}\circ\phi_{\mathrm{RIC}}}(\textsc{Isspace}) = [&b9d5.c06f, 1179.62ff, 01f7.eb04, f377.9eaf,\\ &bf08.8804, 0fba.128f, ca2a.08c0, 1760.b11e,\\ &b7e0.efb6, 2991.8ea5, 8eaf.2a6f, 9dd5.59a2,\\ &54cf.176a].\end{aligned}$$

The RIC-normalized bag of numbers for the two functions has nine numbers (of 13) matching. Both results show strong similarities, and more importantly, we illustrate with this example how an analyst (familiar with various data/executable file formats) may use hash functions to great effect by combining them with data

Table 8.4 Regrouping Lines of Code by the Register Groupings

ISPUNCT (Lines Organized by Position)	Register Attribute
ja 0x100021f0f	
ret	{}
callq 0x10004bd7a	
ret	
setne P[0]	{al}
movzbl P[0], P[1]	{al, eax}
nopw P[0]:(P[1],P[1])	{cs, rax , rax}
shrl $13, P[0]	{eax}
andl $1, P[0]	
testl P[0], P[0]	{eax , eax}
cmpl $127, P[0]	{edi}
movslq P[0], P[1]	{edi, rax}
movl $8192, P[0]	{esi}
pushq P[0]	
popq P[0]	{rbp}
popq P[0]	
movl 60(P[0],P[1],4), P[2]	{rcx, rax, eax}
movq 229669(P[0]), P[1]	{rip, rcx}
movq P[0], P[1]	{rsp, rbp}

The function contents are reorganized for hashing. Notice the parameterized replacement of registers, thereby normalizing away the variable while maintaining the meaning with the attribute known.

Table 8.5 ISPUNCT Hashes by Positional Value

Positional Attribute	Hash Value
0	8853573ca3512634642a5f574d1df63a
0x100021f0f	2dc4564e7ac59eb6c0ab4fb9aff26bbc
0x10004bd7a	9679a36898b4fd96f15936295ea146b5

normalization functions ϕ that remove the unimportant variations in data to induce a common hash. In this way, the technique may be extended generally to many problems.

Cryptographic hashing and universal hash functions are simplistic, efficient, and useful for digesting large and complex data objects into a bag of numbers where they can be compared to a reference set. For large data sets, it's important to understand the properties of the underlying universal hashing properties. With additional knowledge of how unimportant variations in data may be normalized, the normalized bag of words offers endless possibilities to capture similarity signals in data. However, it's not always the case that we have knowledge of how our data should be interpreted,

Table 8.6 ISPUNCT Hashes by Register Use Attribute

Register Attribute	Hash Value
{}	1636e4a1b173928d704847bde7ec48f1
{al}	11790bdd714acb69d25005272c6562ff
{al, eax}	01f7993fcda73368dbd21eb36ce8eb04
{cs, rax, rax}	f377c762ce0a6ca886b668696f659eaf
{eax}	ef3a597d462aaa438580218f89bd26ab
{eax, eax}	0fba84064444a0580c0c30844bc9128f
{edi}	ca2a4f537d823ddedaa58baa6ad908c0
{edi, rax}	1760137b565cff8d0362f3ffec48b11e
{esi}	9464fee12647af58e837fb0b7a04ef9f
{rbp}	29912154aa22885ccb685f10ff5b8ea5
{rcx, rax, eax}	8eaf278cbeac5b50765b97d9e5842a6f
{rip, rcx}	525db9fa4ef126036667343cbd94874b
{rsp', rbp}	54cf2569d9d48a75a2bb76131e29176a

and for this reason, it's important to consider more generally methods for reasoning about string similarity with few assumptions. We conclude this section with a discussion of several other techniques that have various degrees of implementation in cybersecurity practice.

8.2.3 OTHER METHODS

Fuzzy hashing techniques include a large number of approximate techniques, including *context-sensitive hashing*, which may be considered a means to derive natural boundaries in a string by use of context clues within the string. Another common hashing technique is Bloom filters, often used to determine set membership.

A Bloom filter is a randomized algorithm for testing whether a given object is a set member, and it is built on top of a set of k independent hash functions (recall each hash function outputs a bounded integer) which establish a set of bits to set within a fixed-width byte array. Therefore, if we are checking set membership, we can determine the bits which would necessarily be set (assuming the object is a member of the set), and with this, we have a way of accurately refuting the set membership and checking set membership probabilistically.

Another method which introduces a modulus of continuity to a hashing function is to consider the histogram or counts over various symbols in the alphabet. These values have many of the desired reduction properties of a hash function, and they are sensitive to slight variations arising from string mutations.

Suppose that Σ is the alphabet with k symbols. Given a string s of length n, let

$$c_j = \sum_{i=1}^{n} \delta(s[i], \Sigma_j),$$

where $\delta(x, \Sigma_j)$ is 1 if a given symbol x is equal to the jth symbol from Σ and is 0 otherwise. The histogram hash is $H : \Sigma^* \to [0,1]^k : s \to \langle \frac{c_j}{n} \rangle_{j=1}^k$, where $n = |s|$. This hash can be used to create a k-dimensional probability vector projection of a given string. When a string s is modified to string s' via a few mutations or otherwise minor changes, we can expect $H(s), H(s')$ vectors to be close in distance as well. A final heuristic method which may prove to be very useful for the analysis of cyber data is the method called winnowing which was developed for detecting code plagiarism.

Additionally, many subgraph matching and approximate matching techniques exist for analyzing control flow structures in code, and we forego the discussion of those methods here.

8.3 AFFINE ALIGNMENT STRING SIMILARITY

In some cybersecurity-data–driven problems, we wish to reason about data for which we may have no or little information concerning how it is to be formally interpreted. In this scenario, we may be analyzing a firmware program within a poorly documented file format, or we may be reverse-engineering a program that uses some nonstandard data structures and function calling conventions. Often times these problems may focus on finding patterns which indicate data structures, objects, or the entry/return structures of function calls. To consider these problems, we generally imagine strings with common prefixes and suffixes but having interesting forms of variations in-between, and to address this we broaden our approach to consider string-alignment techniques from biology. In this section, we consider several important measures from bioinformatics and their application to code comparison by focusing on the Needleman-Wunsch global affine alignment algorithm. We summarize the common optimization technique known as dynamic programming which underlies the major string similarity measures, including Levenshtein distance, Needleman-Wunsch, and Smith-Waterman similarities. The underling optimization concept is summarized in the *principle of optimality* which is used to construct dynamic programming implementations for optimal alignments. We construct the Needleman-Wunsch alignment algorithm to illustrate dynamic programming and show how it may be used to explore the functions of a binary program, such as would be necessary in a reverse-engineering task.

8.3.1 OPTIMALITY AND DYNAMIC PROGRAMMING

In optimization problems where a solution can be built or synthesized from the solutions of smaller subproblems, a particularly interesting property has been discovered called *the principle of optimality*. As an example of a problem whose solution can be synthesized from like or similar subproblems, consider the problem of determining a minimal distance path through a graph starting from a vertex A and ending at vertex Z. Because a minimal distance path may itself go through some

intermediate point (P), we may ask how the problem of minimizing a path between any of the two endpoints and the intermediate point P may relate to the problem of minimizing a path from A to Z. Naturally, these problems are directly related in that the optimal solution, the minimal distance path from A to Z, will also have to be constructed from optimal paths of subproblems in particular for any pair of verities named in the solution. Let $Opath(X, Y)$ be the optimal path from vertex X to vertex Y, listed as a string over vertices. If the optimal solution from A to Z computed as $Opath(A, Z)$ includes a vertex P, then we reason that there is a prefix relation among the solutions to subproblem $Opath(A, P)$ and $Opath(A, Z)$. This reasoning maybe supported by exploring the logical possibility that P is found in $Opath(A, Z)$ and the optimal path $Opath(A, P)$, somehow doesn't match a prefix of $Opath(A, Z)$, then would there not be a more optimal path from A to Z? The previous argument can form a proof by contradiction that the optimal solutions must be a synthesis of solutions to related subproblems.

When an optimization problem can be seen as a synthesis of optimal solutions for subproblems, the principle of optimality applies. The principle of optimality leads to a solution in the form of dynamic programming where solutions are constructed for the most trivial subproblems first and those solutions are extended with branch and bound to larger problems.

8.3.2 GLOBAL AFFINE ALIGNMENT

We begin by describing the problem and related subproblems. We then show the principle of optimality and define a recursion to solve the problem as a Needleman-Wunsch algorithm. Given two lists of integers L_1, L_2 of length n, m, we imagine an edit procedure which turns L_1 into L_2. We let the alphabet Σ be a bounded set of integers including all the symbols of L_1 and L_2. We envision a cursor position in both L_1 and L_2 at position (i, j), with $0 \leq i < n$, $0 \leq j < m$, and for each cursor position, one of three edits operations can be performed:

- *Mutate*, modifying $L_1[i]$ to match that of $L_2[j]$ and increment i, j
- *Insert*, moving cursor i ahead one position leaving j constant
- *Delete*, moving cursor j ahead one position leaving i constant

When a mutate operation is performed, a cost depending on the two characters will be assessed. All costs are parameterized by a substitution matrix $S : \Sigma \times \Sigma \to \mathbb{R}$ as $S(L_1[i], L_2[j])$. After a mutate operation, the cursor will be updated from (i, j) to $(i + 1, j + 1)$.

When an insert operation is performed, it will cost either ϵ, if an insert was previously performed, or otherwise δ. This differing cost is intended to model a onetime cost of δ (usually large in magnitude) for initializing a cut-and-paste and cost ϵ (usually smaller in magnitude) for extending the cut-and-paste patch of symbols. Together, these two parameters provide a linear or affine function $y = \epsilon x + \delta$ which gives our problem its name. Notice that, in Levenshtein distance, we would have $\epsilon = \delta$.

Similarly, when a delete operation is performed, it will cost either ϵ, if a delete operation was previously used, or δ otherwise. A deletion maybe thought of as an insertion for the other string.

Given the two strings and these edit operations, the problem of *global affine alignment* is to modify $L_1[i]$ into $L_2[j]$ with a maximum reward. To be clear, we will have $\delta < \epsilon < 0$ as cost penalties, and $S(\sigma,\sigma) \geq 0$ for $\sigma \in \Sigma$. Further, we will require that S is symmetric and $S(\sigma,\sigma) \geq S(\sigma,\lambda)$ for any $\sigma,\lambda \in \Sigma$. The solution will be comprised of both a score which is optimal and a sequence of edit operations which will modify L_1 into L_2 in order to attain the optimal score.

To organize subproblems, we consider the substrings of L_1, L_2 as $L_1[a_1 : b_1]$ and $L_2[a_2 : b_2]$ with $a_i \leq b_i$. To establish the principle of optimality, we need to show that, if an optimal global affine alignment between L_1, L_2 also forms an alignment between substrings $L[a_1 : b_1]$ and $L[a_2 : b_2]$, then the alignment of $L[a_1 : b_1]$ and $L[a_2 : b_2]$ must also be optimal. This optimization principle can be established with a proof by contradiction argument similar to that of the minimal path problem.

In Algorithm 8.1 and supporting routines Algorithms 8.2–8.4, we outline a global affine alignment algorithm known as Needleman-Wunsch for any two sequences of integer values L_1 and L_2. Throughout, the algorithms will assume a setting for δ and ϵ and S.

ALGORITHM 1 NEEDLEMAN-WUNSCH ALGORITHM

Data: Given L_1, L_2 ordered lists of integers of length n, m, parameters ϵ, δ, and S are given.
Result: Global alignment similarity score,
`InitializeBoundaryConditions`(m,n) % to initialize M, E, F, P_M, P_E, P_F
for (i,j) *in* `IndexOrder` *(m,n)* **do**
  $v_M = \max(M[i-1,j-1], E[i-1,j-1], F[i-1,j-1])$
  $$P_M[i,j] = \begin{cases} (i-1,j-1) \text{ if } v_M = M[i-1,j-1] \\ (i,j-1) \text{ if } v_M = E[i-1,j-1] \\ (i-1,j) \text{ if } v_M = F[i-1,j-1] \end{cases}$$
  $M[i,j] = v_M + S(L_1[i-1], L_2[j-1])$ % Similarity contribution
  $v_E = \max(M[i-1,j] - \delta, E[i-1,j] - \epsilon, F[i-1,j] - \delta)$
  $$P_E[i,j] = \begin{cases} (i-1,j-1) \text{ if } v_E = M[i-1,j] - \delta \\ (i,j-1) \text{ if } v_E = E[i-1,j] - \epsilon \\ (i-1,j) \text{ if } v_E = F[i-1,j] - \delta \end{cases}$$
  $E[i,j] = v_E$
  $v_F = \max(M[i,j-1] - \delta, E[i,j-1] - \delta, F[i,j-1] - \epsilon)$
  $$P_F[i,j] = \begin{cases} (i-1,j-1) \text{ if } v_F = M[i,j-1] - \delta \\ (i,j-1) \text{ if } v_F = E[i,j-1] - \delta \\ (i-1,j) \text{ if } v_F = F[i,j-1] - \epsilon \end{cases}$$
  $F[i,j] = v_F$
end
return $\max(M[n,m], E[n,m], F[n,m])$

At the core of Needleman-Wunsch algorithm are the decisions about which edit history advances the cursor position from $(0,0)$ to (i,j) in the optimal manner. At each cursor position, various optimal histories are considered in correspondence with the three edit operations. Notice that Algorithm 8.3 specifies that the cursor positions

are explored as a wavefront starting from $(0,0)$, and in such a way that prior to the exploration of (i,j), M, E, and F are computed for cells $(i-1,j-1)$, $(i,j-1)$, and $(i-1,j)$. At each cursor position, Algorithm 8.1 explores edit histories, but advances the histories that are scoring optimally, thereby implementing a branch-and-bound strategy cutting out the considerations of a vast number of edit histories at each step. The algorithm cuts these suboptimal possibilities because of the principal of optimality which essentially guarantees that the histories which are cut cannot in some way contribute to an overall optimal solution to the full problem.

To initialize the recurrence, the following boundary values are set in Algorithm 8.2.

ALGORITHM 2 INITIALIZE BOUNDARY CONDITIONS

Result: Initialize data M, E, F, P_M, P_E, P_F for Needleman-Wunsch.
Data: Given lengths m, n, and parameters δ, ϵ,
$M = Zeros(n,m), E = Zeros(n,m), F = Zeros(n,m)$,
$P_M = Zeros(n,m), P_E = Zeros(n,m), P_F = Zeros(n,m)$
for *i in* $[1,\ldots,(n+1)]$ **do**
  $E[i,0] = -\delta - \epsilon(i-1)$
  $P_E[i,0] = (i-1,0)$% Set the back-pointer
  $F[i,0] = -\infty$
  $M[i,0] = -\infty$
end
for *j in* $[1,\ldots,(m+1)]$ **do**
  $F[0,j] = -\delta - \epsilon(j-1)$
  $P_F[0,j] = (0,j-1)$ % Set the back-pointer
  $E[0,j] = -\infty$
  $M[0,j] = -\infty$
end
$M[0,0] = 0$
$B[0,0] = 0$
$E[0,0] = -\infty$
$F[0,0] = -\infty$

To create an ordering over the array that guarantees that M, E, and F are computed for cells $(i-1,j-1)$, $(i,j-1)$, and $(i-1,j)$ prior to (i,j), we create a wavefront ordering using the following Algorithm 8.3. This ordering can be used in all of the dynamic programming alignment algorithms.

ALGORITHM 3 INDEX ORDER

Data: Given lengths m, n.
Result: An array index order or itinerary listing indices for dynamic programming.
$order = []$
for *d in* $[1,\ldots,(m+n+1)]$ **do**
  for *s in* $[\max(1,d-(m-1)), min(d,n)]$ **do**
    $order.append([d-s,s])$
  end
end
return *order*

Finally, we provide a simple edit-history recovery method which operates on the values in M, E, and F in order to recover an edit transcript. The transcript is recovered from a traceback of optimal values in the array indices starting from (n, m) and going back to $(0, 0)$. The steps are sketched in Algorithm 8.4.

ALGORITHM 4 TRACEBACK

Data: Given arrays M, E, F, P_M, P_E, and P_F.
Result: Traceback for the optimal solution given as an edit transcript path listed as coordinates.
$v = max(M[m,n],\ E[m,n], F[m,n])$
$path = []$
$i, j = (n, m)$
$path.append((i, j))$
while $(i, j) \neq (0, 0)$ **do**
  $v = \max(M[i,j], E[i,j], F[i,j])$
  $$i,j \leftarrow \begin{cases} (i-1, j-1) \text{ if } v = M[i,j] \\ (i, j-1) \text{ if } v = E[ij] \\ (i-1, j) \text{ if } v = F[i,j] \end{cases}$$
  $path.append((i, j))$
end

8.3.3 EXAMPLE ALIGNMENTS

To normalize two functions for comparison, we define a tokenization technique. Each token is compared to dictionary keys which map to integer values when they have been previously observed. If they have not been observed, new integers distinct from all the other values of the dictionary are assigned to them (Table 8.7).

The normalized view of our functions are ISPUNCT represented by the following sequence: 0 1 2 3 1 13 120 101 121 15667 26 101 8 2 15668 52 31 9 71 31 8 110 21 111 1017 21 112 90 21 44 1 45 9 4583 10 29 97 22 21 21 98 66 99 66 21 44 1 45 46 47 8 8; and ISSPACE represented by the following sequence: 0 1 2 3 1 13 120 101 121 242 26 101 8 2 243 52 31 9 71 31 8 110 21 111 244 21 112 90 21 44 1 45 9 245 10 29 97 22 21 21 98 66 99 66 21 44 1 45 46 47 8 8.

Both sequences have length 52.

The global alignment can be represented by the view

```
!"#$".;(<‘;()#aU@*h@)162n63{6M"N*h+>$766%c&c6M"NOP))
|||||||||X||||X|||||||||X||||||||X||||||||||||||||||
!"#$".;(<W;()#XU@*h@)162Y63{6M"N*Z+>$766%c&c6M"NOP))
```

In this edit view, we can see the symbols which align with a pipe symbol linking them (ie, "—"), when mutations are introduced an "X" represents the edit which took place.

We include a few additional function comparisons which show a more interesting ability to create and extend a gap in order to accommodate additional matching later on in the string. Take, for example, the two functions LOOKUP and REMOVE-SYMBOL with the following alignment:

Table 8.7 Two Functions and Their Dictionary-Normalized Integer Representations

ISPUNC	Normalized	ISSPACE	Normalized
pushq %rbp	0 1	pushq %rbp	0 1
movq %rsp, %rbp	2 3 1	movq %rsp, %rbp	2 3 1
cmpl 27, %edi	13 120 101	cmpl 27, %edi	13 120 101
ja 0x100021f0f	121 15667	ja 0x100021bcf	121 242
movslq %edi, %rax	26 101 8	movslq %edi, %rax	26 101 8
movq 229669(%rip), %rcx	2 15668 52 31	movq 230501(%rip), %rcx	2 243 52 31
movl 60(%rcx,%rax,4),%eax	9 71 31 8 110 21	movl 60(%rcx,%rax,4), %eax	9 71 31 8 110 21
shrl 3, %eax	111 1017 21	shrl 4, %eax	111 244 21
andl, %eax	112 90 21	andl , %eax	112 90 21
popq %rbp	44 1	popq %rbp	44 1
ret	45	ret	45
movl 192, %esi	9 4583 10	movl 6384, %esi	9 245 10
callq 0x10004bd7a	29 97	callq 0x10004bd7a	29 97
testl %eax, %eax	22 21 21	testl %eax, %eax	22 21 21
setne %al	98 66	setne %al	98 66
movzbl %al, %eax	99 66 21	movzbl %al, %eax	99 66 21
popq %rbp	44 1	popq %rbp	44 1
ret	45	ret	45
nopw %cs:(%rax,%rax)	46 47 8 8	nopw %cs:(%rax,%rax)	46 47 8 8

```
!"#$"!&!'!R!(#-&>Q#)RJNNS]U)#&-#R=*$xJpp>)#)(<,(K^U(p_HKL(#((A((B‘,RC
 (Za#:(-#&=#
|||||||||||||||X||||    X|X|||X||X||||||||||||X||X|X|X|           XXX|X
 XX|X|||||X||
!"#$"!&!'!R!(#-'>Q#)---&SvU)#'-#&=*$xJpp>)#)R<,RKwURp----------x:),C
 (&Zy#:(-#'=#

RW>v766--------------Z--------a#r(---'#')M(MRM'M&M"NOP))
X||||||               |         X|X|   X|X|||||||||||||  ||
&W>v766BzKL(#(R#R(A((Z{J,,p|R}U#H()#)R#()M(MRM'M&M"NO-))
```

This alignment, represented in two alignment rows of 80 symbols each, reveals that the algorithm with $\delta, \epsilon = -10, -1$, and $S(a,b) = \begin{cases} 10 \text{if} a = b \\ 0 \text{o.w.} \end{cases}$, can introduce sizable gaps (on both sides) in order to preserve string matching such as what we see early on in the head, as well as the tail.

When we reconsider the alignment with $\delta, \epsilon = -50, -10$, we get a slightly different answer which is likely to use mutation:

```
!"#$"!&!'!R!(#-&>Q#)RJNNS]U)#&-#R=*$xJpp>)#)(<,(K^U(p----------------
 _HKL(#((A((B
|||||||||||||||X||||    X|X|||X||X||||||||||||X||X|X|X|
 XXXXXXXXXXXX
!"#$"!&!'!R!(#-'>Q#)---&SvU)#'-#&=*$xJpp>)#)R<,RKwURpx:),C(&Zy#:(-#'
 =#&W>v766BzK

‘,RC(Za#:(-#&=#RW>v766Za#r('#')M(MRM'M&M"NOP))
XXXXXXXXX|XXXXXXXXXXXXXXXXXX|X|||||||||||||  ||
L(#(R#R(A((Z{J,,p|R}U#H()#)R#()M(MRM'M&M"NO-))
```

This simple example with various affine alignment parameters shows that the method is flexible and able to accommodate a large number of similarity concepts. In particular, it forms a demonstration that alignment algorithms are valuable computational tools capable of identifying various similarity notions in comparable objects.

8.4 SUMMARY

Strings are common in cybersecurity data. String data in cybersecurity problems are usually formal encodings of data, objects, or functions. Due to large volumes of data, efficient techniques for analyzing strings are needed. Cryptographic and universal hash functions have also been utilized for their efficiencies; however, it's important to understand both their properties and limitations concerning the possibility of hash collisions. We present several examples of how hash functions may be used to find common objects with the bag of numbers technique. The bag of numbers technique is extended with the normalized bag of numbers to retain the efficiencies and precision of hashing but address the effects of noise patterns in data.

In the context of reverse engineering and code analysis, where little or nothing is known about the code's provenance, the alignment algorithms offer distinct value in finding pattern motifs, such as calling conventions, or in identifying commonly occurring structures. In the design of protocols, we often hear the analogy that signals can be constructed like trains (a sequence of engines and cars comprise a train), and we may expect common patterns and sequences in particular in the header and trailer sections. So, perhaps one way to consider the utility of alignment algorithms is that it can do well in matching such preserved patterns and identifying the variants. The Needleman-Wunsch algorithm can be used to determine motifs and sources of variations even when we know little about its origin or history.