Contents lists available at SciVerse ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

Unicode search of dirty data, or: How I learned to stop worrying and love unicode technical standard #18

Jon Stewart*, Joel Uckelman

Lightbox Technologies, Inc., 1400 Key Boulevard, Suite 1000, Arlington, VA 22209, USA

Keywords:
Unicode
Regular expression
Regex
Search
Ditigal forensics

A B S T R A C T

This paper discusses problems arising in digital forensics with regard to Unicode, character encodings, and search. It describes how multipattern search can handle the different text encodings encountered in digital forensics and a number of issues pertaining to proper handling of Unicode in search patterns. Finally, we demonstrate the feasibility of the approach and discuss the integration of our developed search engine, lightgrep, with the popular bulk_extractor tool.

1. Introduction

Digital evidence typically includes text in a variety of encodings, languages, and scripts. Document formats often denote the encoding(s) of contained text explicitly, but other file types do not, nor can we expect such metadata to exist for text fragments in slack and unallocated space. Due to the prevalence of UTF-16LE and UTF-8, many investigators learn a little about encodings early in their careers, but receive little training related to internationalization issues. Likewise, many tools simply treat UTF-16LE as ASCII separated by null bytes and those tools which do purport to handle Unicode leave many aspects of their handling undocumented. The increasingly global nature of cybercrime means that neither professional ignorance nor dodgy tools will be acceptable for long.

A number of regular expression engines provide searching over Unicode text, but generally assume that all input is sanitized text in a single, known encoding. This model does not apply in digital forensics, as the binary artifacts investigators analyze may mix encodings, and raw sector data in unallocated portions of media may switch encodings arbitrarily. In the course of developing *lightgrep*, a new regular expression search engine for digital forensics usage, we sanguinely thought that "Unicode Support" would be just another feature to implement. While support for some basic Unicode features is not hard, we discovered that proper handling requires considerable research and, sometimes, judgment. This paper documents some of what we have learned in this endeavor and describes our approach, keeping in mind that the exigencies of forensics sometimes necessitate different choices than in standard tools. The end result is a regular expression search engine that supports the full range of Unicode characters and over 180 encodings, and is available for investigators to use in the open source bulk_extractor tool (Garfinkel). Moreover, the approach serves as a useful framework for thinking about the problem, where necessary complexity regarding which encodings to use for searches and Unicode considerations are exposed, but matches and surrounding context are presented in a single, standard encoding.

2. Character encoding basics

A *coded character set* is a list of pairs, each consisting of a character and the unique integer, known as a *code point*, representing it. An *encoding* maps sequences of code points to sequences of bytes. Examples: Unicode is a coded character set consisting of 1,114,112 code points, intended to be sufficient for representing all text produced by humans.

* Corresponding author.
E-mail addresses: jon@lightboxtechnologies.com (J. Stewart), joel@lightboxtechnologies.com (J. Uckelman).

1742-2876/$ – see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.diin.2013.06.013

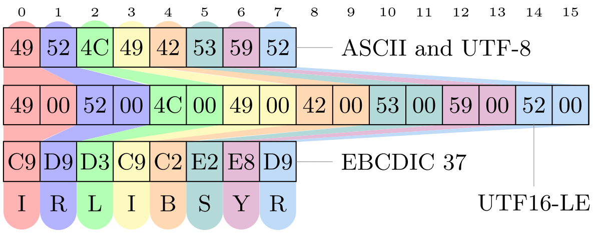


Fig. 1. Several encodings of the text “IRLIBSYR”.

UTF-8 and UTF-16 are encodings capable of representing all Unicode code points as bytes. ASCII, commonly used for simple text files (especially by English speakers), is both a coded character set and an encoding—the 128 code points in ASCII, numbered 0–127, are directly encoded as bytes 0–127. Numerous encodings which were developed specifically for one or more natural languages—such as Shift JIS, EUC-KR, KOI8-R, and ISO 8859-1—are slowly being supplanted by UTF-8 and UTF-16.

3. Multipattern search is multiencoding search

The multiplicity of encodings means that one piece of text can be represented as bytes in numerous ways. E.g., the text “IRLIBSYR” can be encoded as in Fig. 1. The UTF-16LE encoding, while containing values similar to the UTF-8 encoding, is twice the length, while the EBCDIC 37 encoding bears no resemblance to the other two. Searching a block of bytes for “IRLIBSYR” can mean searching for “IRLIBSYR” once for each possible encoding. Hundreds of encodings exist, and while the vast majority of them are obsolete and unlikely to be encountered in data from a contemporary computer, there are still a few dozen which are in common usage. Therefore, if we are trying to establish the existence of a particular string within data for which we don’t know the encoding beforehand, the search task can be rather large. Should we need to do this for hundreds, or even thousands of strings (or, in the general case, regular expressions), it will require a prohibitive amount of time to carry out the search serially, reading the data once for each pattern-encoding pair.

Fortunately, multipattern search rescues us in this situation: Searching in parallel for one string rendered in multiple encodings is structurally the same problem as searching in parallel for multiple distinct strings. Hence, a search for the pattern IRLIBSYR.*?SACPOP[1] in UTF-8, UTF-16LE, UTF-16BE, UTF-32LE, and UTF-32BE does not necessitate five passes over the data (once for each encoding), but merely adds five patterns to the pattern set from which we build our search automaton. For a detailed description of the techniques necessary for efficient multipattern search see Stewart and Uckelman (2011).

The literals in our regexes are Unicode characters, *not* bytes in a particular Unicode encoding (such as UTF-8). This distinction isn’t present when working with traditional ASCII regular expressions, where the characters and the bytes are identical. Distinguishing between characters and bytes in Unicode regular expressions makes the regexes *encoding-independent*, which is a useful property should one wish to search for the same character string in several different encodings. Furthermore, having literals be Unicode characters can improve regex readability over specifying them by code point: While 屁股 and 地洞 are easily confusable for those unfamiliar with Chinese, and thus specifying the characters by code point may be called for, of the patterns matching the Greek name for the city of Athens, even non-Grecophones are likely to prefer Αθήνα to \x{0391}\x{03B8}\x{03AE}\x{03BD}\x{03B1}.

We now give two examples of the search automata generated by single patterns interpreted in multiple encodings. (In these examples, we represent bytes which correspond to ASCII letters as the ASCII letters, for the sake of readability. E.g., “L” in the automaton in Fig. 2 should be understood as the byte 0x4C. The ∅ represents the byte 0x00, a null. For further clarity, we have omitted some of the labeling which the automata would have in practice, as it is not relevant for our examples.) In Fig. 2, we have the nondeterministic finite automaton (NFA) for the pattern Lolita in UTF-8, UTF-16LE, and UTF-32LE. All three encodings share the ‘L’ byte. On the second byte, UTF-8 diverges from the other two encodings, following the upper branch of the automaton, as the UTF-8 representation of “Lolita” is simply the 7-bit ASCII representation. Both UTF-16LE and UTF-32LE branch downward, with a null byte as their common second byte. On the third byte, UTF-16LE moves on to the second character “o”, while UTF-32LE is still padding out the “L” with null bytes. (Recall that UTF-32LE uses four bytes per character, while UTF-16LE uses only two bytes per character for characters with code points below U+10000.) In the NFAs we produce, each accepting state is labeled according to the pattern-encoding combination which produced it, so that for each hit, we may recover the pattern and encoding for which it is a hit. (The pattern-encoding labels are omitted from Figs. 2 and 3 to reduce clutter.)

Fig. 3 shows the search automaton for [€$][\t]*\d+, a (greatly simplified) pattern for matching amounts of money in dollars or euros, encoded in ISO-8859-15 and Windows-1252.[2] Possible matches for the pattern include “$1000000” and “€ 99”. The symmetry of this automaton is caused by ISO-8859-15 and Windows-1252 disagreeing on the byte for EURO SIGN: ISO-8859-15 uses 0xA4 and comprises the top half of the NFA, while Windows-1252 uses 0x80 and comprises the bottom half. The middle of the NFA is

[1] “IRLIBSYR” being the nuclear-armed coalition of Iran–Libya–Syria, led by notorious AMNAT–SOVWAR interloper Evan Ingersoll, who is trying to fend off Strikes Against Civilian Populations (SACPOP). For more, see (Wallace, 1996, pp. 321–342).

[2] The pattern [€$][\t]*\d+ is not one we recommend for matching arbitrary currency strings, as it misses trailing (instead of leading) currency symbols, many other currency symbols, thousands and decimal separators, and non-linebreaking whitespace other than spaces and tabs. The example pattern would completely miss “1.000.000,00 €”. A pattern which better captures these intricacies, such as (\p{Sc}\s*(\d*|(\d{1,3}([,.]\d{3})*))([,.]\d{2}?)?)|((\d*|(\d{1,3}([,.]\d{3})*))([,.]\d{2}?)?\s*\p{Sc}) in UTF-8 has an NFA with 94 vertices, does not fit on the page, and looks like a giant hairball, so makes for a poor example—though presents no problems for searching. Note that even this extended pattern does not capture quantities written using non-Arabic numerals or more exotic Unicode whitespace.

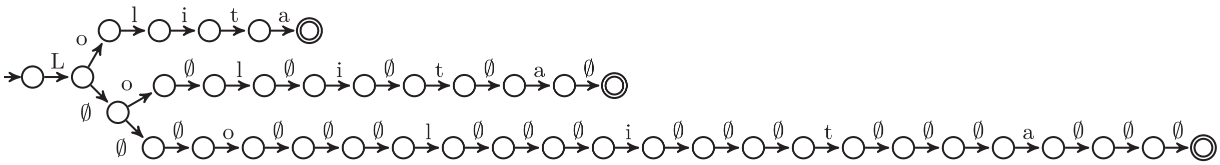


Fig. 2. Merged search automaton for Lolita in UTF-8, UTF-16LE, and UTF-32LE.

shared between the encodings, as they agree on the byte for DOLLAR SIGN.

4. Fun with Unicode

Unicode Technical Standard #18 (hereafter, UTS #18) provides guidelines for supporting Unicode characters in regular expressions (Davis and Heninger, July 2012).

UTS #18 defines three levels of support for Unicode in regexes. Level 1, Basic Unicode Support, requires support for specifying Unicode characters by their code points (RL1.1), selectors for Unicode properties (RL1.2), character class union, subtraction, and intersection (RL1.3), simple word boundary matching (RL1.4), simple loose matching (RL1.5), line boundary matching (RL1.6), and expression of the full range of Unicode characters (RL1.7). Level 2, Extended Unicode Support, requires support for matching canonical equivalents (RL2.1), extended grapheme clusters (RL2.2), default word boundaries (RL2.3), name properties (RL2.5), all Unicode properties (RL2.7), as well as support for full default case folding and (RL2.4) and wildcards in property names (RL2.6). Level 3, Tailored Support, requires locale-specific punctuation properties (RL3.1), grapheme clusters (RL3.2), and word boundaries (RL3.3), as well as context (RL3.6) and incremental (RL3.7) matching, the generation of possible match sets (RL3.9), and match subroutines (RL3.11). We now proceed to explain each of the requirements.

4.1. Level 1: basic Unicode support

RL1.1 requires Level 1-conformant regex languages to provide a way of specifying code points in hexadecimal. Following the venerable Perl Compatible Regular Expression library (PCRE), lightgrep meets this requirement with the metacharacter \x, which takes a hexadecimal code point as its argument. E.g., LATIN SMALL LETTER A (a) and ARABIC LIGATURE SALLALLAHOU ALAYHE WASALLAM (ﷺ), which is virtually illegible on-screen at 10pt, can be specified as \x{61} and \x{FDFA}, respectively.

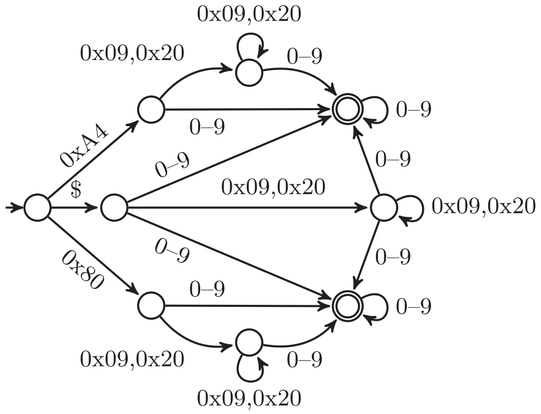


Fig. 3. Merged search automaton for [€$][\t]*\d+ in ISO-8859-15 and Windows-1252.

RL1.2 requires Level 1-conformant regex languages to support the character classes defined by the Unicode properties General Category, Script, Alphabetic, Uppercase, Lowercase, White Space, Noncharacter Code Point, Default Ignorable Code Point, Any, ASCII, and Assigned. Unicode properties are fundamentally sets of characters. The General Category and Script properties are multivalued, while the other properties are binary. For example, any given character is either Uppercase or not, while the Script for a letter might be Arabic, Latin, Greek, Cyrillic, or any of 100 other scripts defined in Unicode 6.2.[3] Following PCRE, we meet this requirement with the metacharacter \p, which takes a Unicode property name (with possible value) as its argument. E.g., \p{Script = Arabic} matches any character in Arabic script; \p{General Category = Currency Symbol} matches currency symbols such as $, ¢, €, and ¥; and \p{White Space} matches spaces, tabs, newlines, and carriage returns, as well as more exotic whitespace such as MONGOLIAN VOWEL SEPARATOR. \p{Any} is the class of all Unicode code points, while \p {Assigned} is the class of all code points with a character assigned to them. UTS #18 recommends, but does not require, leniency with regard to spaces, case, hyphens, and underscores. Hence, \p{Whitespace}, \p{wHiTeSpAcE}, and \p{_ _ _white space_ _ _} are equivalent. The Unicode Standard defines short forms for some properties (such as “Khmer” for “Script = Khmer”), which we also support.

RL1.3 requires Level 1-conformant regex languages to support union, intersection, and set difference on character classes. E.g., one might wish to search for anything Cyrillic or Greek (the union of Cyrillic and Greek scripts), or lowercase Latin letters (the intersection of the Latin letters and the lowercase letters, or, equivalently, the Latin letters minus the uppercase letters). We support all three operations (and also symmetric difference, which is suggested, but not required). The patterns [\p{Cyrillic}\p{Greek}] and [\p{Latin}&&\p {Lowercase}] correspond to the searches just mentioned.

RL1.4 requires Level 1-conformant regex languages to treat all characters having property Alphabetic or Decimal Number as well as the ZERO WIDTH NON-JOINER and ZERO WIDTH JOINER characters as being word characters for the purpose of

[3] For a complete list, see the Unicode 6.2 data file PropertyValueAliases.txt (The Unicode Consortium).

matching word boundaries, and additionally that combining marks (e.g., combining ring above) are not separated from their base characters by word boundaries. (A word boundary customarily occurs between any two consecutive characters where one is a word character and the other not, and also at the beginnings and ends of strings. Word boundaries, due to occurring between characters, have zero width.) We do not presently support word boundaries, but doing so is trivial once we add support for positive look-ahead and -behind assertions.

RL1.5 requires Level 1-conformant regex languages to support default case-insensitive matching for all Unicode characters if they support case-insensitive matching at all, and additionally to indicate which properties will be closed under simple default case folding when matching case-insensitively. In order to unpack this requirement, we need to understand default case-insensitive matching, closure, and simple default case folding. Default case-insensitive matching, as defined in The Unicode Consortium (§3.1.3), is too complex to explain here. It suffices to say that the standard defines the function toCasefold for every string, and two strings S,T are default case-insensitively equivalent iff toCasefold(S) = toCasefold(T). Thinking of case-folding as simple lowercasing is fairly close, but misses a few corner cases that the standard handles.

To accord with RL1.5, a pattern interpreted case-insensitively must match all strings which are case-insensitively equivalent to the strings that same pattern would match case-sensitively. This requirement touches on a feature of Unicode which surprises novices, namely that lowercasing and uppercasing are not inverses. According to the Unicode Standard, both latin capital letter k (K) and the kelvin sign (K) lowercase to latin small letter k, but latin small letter k (k) uppercases to latin capital letter k (K) only.[4] Both latin small letter s (s) and latin small letter long s (ſ) uppercase to latin capital letter s (S), but latin capital letter s lowercases to latin small letter s only.[5] A consequence of this is that the character class [A-Z] matches not 52, but 54 different letters when interpreted case-insensitively in an encoding supporting full Unicode.

Now for property closure: Recall that properties can be thought of as sets of characters. Let C_0 be a set, o a function from sets to sets, and $C_n = o(C_{n-1}) \cup C_{n-1}$ for $n > 0$. The set which is the closure of C_0 under o is the C_i such that $C_i = C_{i+1}$. (And, in fact, if C_i is closed under o, then $C_i = C_j$ for all $j \geq i$.) That is, the closure of C_0 under o is the set from which no new elements are generated when o is iteratively applied to it. E.g., the set {A} is not closed under simple default case folding because A case-folds to a, and a$\notin${A}. The set {A,a} is closed under simple default case-folding, since A case-folds to a and a case-folds to A. Now it should be clear what it would mean for a Unicode property to be closed (or not) under case-folding when matching case-insensitively. The property General Category = Letter, by virtue of containing all letters, whether upper-, lower- or uncased, cannot fail to be closed under simple default case-folding when matching case-insensitively in any correct implementation, but whether an implementation similarly closes the property General Category = Uppercase Letter is up to the implementers. It would be strange—though permissible—if \p{General Category = Uppercase Letter} in a case-insensitive pattern did not match lowercase letters. We chose to close all properties under simple default case-folding when matching case-insensitively, in accordance with the principle of least surprise.

RL1.6 requires Level 1-conformant regex languages to recognize all of LF (U+0A), CR (U+0D), CR LF (U+0A U+0D), NEL (U+85), line separator (U+2028), and paragraph separator (U+2029) as ending logical lines. (LF is the traditional end-of-line character on UNIX, CR LF on Windows, CR on MacOS prior to OS X. Customarily, ASCII regex implementations would recognize some combination of these as line endings, perhaps depending on the platform where they were running.) The start- and end-of-line assertions ^ and $ are not terribly useful when searching non-line-oriented data such as disk images. Hence, we support RL1.6 vacuously by not supporting line-break assertions at all.

RL1.7 requires Level 1-conformant regex languages to handle all code points, as well as treating UTF-16 surrogate pairs as single code points when matching. In the early days of Unicode, some encoders and decoders balked at valid characters above U+7FF (the last two-byte character in UTF-8) or U+FFFF (the last two-byte character in UTF-16); some UTF-16 decoders incorrectly decoded surrogate pairs as two invalid code points rather than single valid ones. These two requirements are aimed at heading off such shoddy implementations. We support all code points and correctly interpret UTF-16 surrogate pairs.

4.2. Level 2: extended Unicode support

RL2.1 deals with canonical equivalents of strings, but states no clear requirement. Ideally, if a pattern matches a string S, then it would also match every string T which has the same normalization as S. Unicode defines several normalizations (e.g., NFS, NFKD), and as it is unclear how to normalize data of mixed or unknown encoding before searching, we do not yet support RL2.1.

RL2.2 requires Level 2-conformant regex languages to provide a way of matching any extended grapheme cluster, a literal cluster, and extended grapheme cluster boundaries. An *extended grapheme cluster* is the glyph that from a user's point of view consists of a single character (This is a gloss. For the definition, see Davis (September 2012, §3)). E.g., the sequence of the two Unicode characters latin small letter o and combining double acute accent form the extended grapheme cluster ő used in Hungarian (which can also be produced by the single Unicode character latin small letter o with double acute). The metacharacter for matching any extended grapheme clusters is analogous to the dot metacharacter for matching any single code point. PCRE uses \X for this purpose. E.g., on the sequence ⟨o U+030B⟩ the regex. matches the o, while \X matches the o and its combining accent as a unit. The requirement for matching

[4] Note that in many fonts, the Kelvin sign is indistinguishable from a capital K.

[5] You might have seen a long s, also known as a medial s, in books and documents from the 17th and 18th centuries. E.g., the title page of the 1668 edition of Milton's *Paradise Lost* spells the title as *Paradiſe Loſt*. The U.S. Declaration of Independence contains numerous examples, such as "the pursuit of Happineſs".

literal clusters is intended to facilitate including them in character classes, e.g., as in [ő\q{o\x{030B}}] which follows the UTS #18 sample syntax and is intended to match either representation of ő. Support for literal clusters is vexing, as it means that a character class can no longer be expected always to match a single character—in the example, one member of the class is two Unicode characters long, instead of one. We do not yet support any of RL2.2.

RL2.3 requires Level 2-conformant regex languages to provide assertions for matching Unicode default word boundaries, as defined in Davis (September 2012, §4). This requirement is far more demanding than simple word boundaries as defined in RL1.4, and we do not presently support it.

RL2.4 requires Level 2-conformant regex languages to provide full default Unicode case folding if they provide case conversion at all. The salient difference between this requirement and that of RL1.3 can be seen for characters like U+DF, latin small letter sharp s (ß). In German, ß capitalizes to SS. This causes common words (e.g., straße) to change length when converted to uppercase (STRASSE). Similar things apply to latin capital ligature ij (Ĳ), latin small ligature ij (ĳ), and the sequences IJ and ij, all of which are considered a single letter in Dutch; the digraphs latin small letter dz (ǳ), latin capital letter dz (Ǳ), and latin capital letter d with small letter z (ǲ), used in some Slavic languages; as well as the various ligatures latin small ligature ff (ﬀ), latin small ligature fi (ﬁ), latin small ligature fl (ﬂ), etc. used in quality printing. Full default case folding should make the pattern straße case-insensitively match all of the following: strasse, straße, STRASSE. While this feature is undoubtedly useful, especially for investigators unaware of the subtleties of various languages, the primary difficulty with implementing RL2.4 is the same as with RL2.2, namely that character classes can sometimes contain character sequences in addition to single characters, and as such, we do not yet support it.

RL2.5 requires Level 2-conformant regex languages to support referring to characters by name. Many Unicode characters, such as apl functional symbol tilde diaresis (⍨) and pile of poo (💩), are not easy to type and have descriptive names more memorable than their hexadecimal code points; therefore, being able to specify them by name might on occasion be useful. Following PCRE, we provide the \N metacharacter, which takes the name of a character as its argument and matches the named character, as well as supporting the Name property via \p. E.g., thai character tho phuthao (ฒ) is matched both by \N{thai character tho phuthao} and \p{name = thai character tho phuthao}.

RL2.6 requires Level 2-conformant regex languages to support wildcards in Unicode property values. The suggested syntax is more expansive, permitting arbitrary regular expressions to appear as property value matchers. For example, \p{name = /l.*tel/} would produce a character class containing byzantine musical symbol teleia (𝀏), black telephone (☎), and love hotel (🏩), among others. We do not yet support this, as we do not yet have a compelling use case for it.

RL2.7 requires Level 2-conformant regex languages to support all (non-provisional, non-contributory, non-obsolete, non-deprecated) Unicode properties. This amounts to several dozen properties beyond those required by RL1.2, such as Hex Digit, Changes When Lowercased, and Ideographic. We support these properties.

4.3. Level 3: tailored support

RL3.1 requires Level 3-conformant regex languages to have locale-specific punctuation properties. For example, left-pointing double angle quotation mark («) and right-pointing double angle quotation mark (») might be considered Punctuation in a French locale, but not in an English one.

RL3.2 requires Level 3-conformant regex languages to have locale-specific collation order for collation grapheme clusters. The most striking effect of a locale's collation order on regexes involves ranges in character classes. The standard collation order used for ranges in character classes is ascending code point order. In the standard, locale-agnostic collation the character class [O–P] matches O and P only, which is correct for many languages—but not for German, which sorts Ö between O and P. (This problem is not solvable simply by declaring that Ö always sorts between O and P, as then the standard collation would be wrong for Swedish, where Ö sorts as the last letter of the alphabet.) Hence, a regex language conforming to this requirement should provide a collation order which respects the locale selected by the user.

RL3.3 requires Level 3-conformant regex languages to have locale-specific word boundaries. Default word boundaries (RL2.3) are insufficient for splitting words in some languages. Thai, for example, has no interword whitespace, so detecting word boundaries is not a simple matter of finding adjacent word and non-word characters.

We chose not to support RL3.1–3 because using them correctly requires language-specific expertise that forensics investigators are unlikely to possess—to appreciate the problem, consider for how many languages for which you are familiar with their collation order. Given that, implementing these for more than a handful of locales is a gargantuan task with little benefit for forensics.

RL3.6 requires Level 3-conformant regex languages to support unmatchable leading and trailing context. For a given input text, a target range is specified within which matches may occur, which range sits within a context range where matches may not occur, though the context may be referred to by zero-width look-around assertions. Look-around assertions clearly are useful in forensics, and we intend to implement them; but we see no obvious use in forensics for the remainder of this requirement.

RL3.7 requires Level 3-conformant regex languages to support incremental matching. Normally, a pattern which has not yet matched by the end of input fails to match. E.g., the pattern abc has no match on the text ab—but, were more text appended, cde, for example, then abc could still match. Incremental matching is highly desirable for forensics, in particular when searching data which does not fit into RAM, such as disk images. Typically, any large block of data is searched by reading a chunk into a buffer in RAM, searching the buffer, refilling the buffer with the next chunk of data, etc. until all data is exhausted. If partial matches are not carried over from one fill of the buffer to the next, then either matches spanning chunks will be missed, or some horrifically complicated algorithm will be required to find cross-chunk

Please cite this article in press as: Stewart J, Uckelman J, Unicode search of dirty data, or: How I learned to stop worrying and love unicode technical standard #18, Digital Investigation (2013), http://dx.doi.org/10.1016/j.diin.2013.06.013

matches. By keeping partial matches alive across buffer fills, problems with matches spanning two or more buffers vanish. We support incremental matching by permitting matches to span an unlimited[6] number of buffers.

RL3.9 requires Level 3-conformant regex languages to support the generation of possible match sets for any given pattern. Novice regex users have always been plagued by unexpected matches and nonmatches. With the expansion of regexes to Unicode, the possibility that a given regex could match unexpectedly increases—hence, there is great practical utility in being able to see sample matches for any given pattern, especially if that means correcting a faulty pattern before running an hours- or days-long search. We support generation of possible matches for a pattern by randomly walking its search automaton, reading off a match for each complete branch walked.

RL3.11 requires Level 3-conformant regex languages to permit registration of arbitrary submatching functions. We have no intention of supporting this, as the injection of arbitrary code into our match engine is incompatible with providing performance guarantees.

4.4. *UTS #18 conformance*

Support for Unicode in regexes is far from complete, both in our implementation and elsewhere, though there has been much progress since UTS #18 was first published in 1998. Table 1 summarizes UTS #18 conformance for lightgrep, as well as for ICU 50 (IBM, 2013), Perl 5.16 (perlunicode man page), Java 7 (Oracle, 2013), and the Python regex library (Barnett, 2013). (PCRE has approximately the same support as Perl, by design.) Among these, only lightgrep supports multipattern matching, which is of such great utility in forensics.

Table 1
UTS #18 Support.

	Lightgrep	ICU 50	Perl 5.16	Java 7	Python regex
Level 1: Basic Unicode Support	○	●	○	●	●
RL1.1 Hex Notation	●	●	●	●	●
RL1.2 Properties	●	●	●	●	●
RL1.3 Subtraction and Intersection	●	●	○	●	●
RL1.4 Simple Word Boundaries		●	●	●	●
RL1.5 Simple Loose Matches	●	●	●	●	●
RL1.6 Line Boundaries		●	○	●	●
RL1.7 Supplementary Code Points	●	●	●	●	●
Level 2: Extended Unicode Support	○	○	○	○	○
RL2.1 Canonical Equivalents				●	
RL2.2 Extended Grapheme Clusters		○	○		●
RL2.3 Default Word Boundaries		●			
RL2.4 Default Case Conversion					
RL2.5 Name Properties	●	●	●		●
RL2.6 Wildcards in Property Values					
RL2.7 Full Properties	●		●		●
Level 3: Tailored Support	○		○		
RL3.1 Tailored Punctuation					
RL3.2 Tailored Grapheme Clusters					
RL3.6 Context Matching			○		
RL3.7 Incremental Matches	●				
RL3.9 Possible Match Sets	●				
RL3.11 Submatchers					

● full support, ○ = partial support.

5. Encoding chains

Character encodings are a special case of something more general. Character encodings map (sequences of) code points to sequences of bytes. One could also imagine transformations which map (sequences of) code points to (sequences of) code points, and sequences of bytes to sequences of bytes, and that such transformations could be chained together, as in Fig. 4. (*Composed* would be the mathematical term here.) For example, the Caesar cipher, ROT-13, and URL-encoding can be thought of as maps from code points to points, while the XOR and ROR/ROL obfuscation schemes[7] common to malware are maps from bytes to bytes. It is not far-fetched to think that one might want to search for text which is both encoded and transformed in one of these additional ways.

A prime example of this is Outlook Compressible Encryption, which, in keeping with its oxymoronic name, is just a substitution cipher used to obfuscate email data within Microsoft Outlook PST archive files.[8] This provides no real security and is easily defeated by frequency analysis—nonetheless, OCE is an impediment to searching Outlook mailbox files, since it changes the byte patterns contained therein. This forces the investigator either to hand-adjust patterns to account for OCE (which requires knowledge of both the target character encoding and OCE), or to decipher the OCE'd portions of Outlook PST files before searching them (which will not find email fragments in unallocated space). A better solution—one which we have implemented—generalizes specifying multiple encodings for each pattern by letting the user specify one or

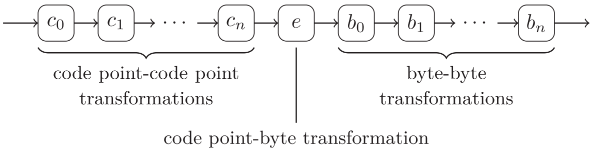

c_0
c_1
$\cdots$
c_n
e
b_0
b_1
$\cdots$
b_n
code point-code point transformations
byte-byte transformations
code point-byte transformation

Fig. 4. An abstract transformation chain.

[6] Where "unlimited" $<2^{64}$. We don't expect anyone to need offsets larger than 64 bits, at least not in 2013.

[7] Didier Stevens' XORSearch utility handles these schemes as well. See http://blog.didierstevens.com/programs/xorsearch/.

[8] The substitution scheme is documented in Joaquin Metz's libpff, http://code.google.com/p/libpff/.

more *transformation chains* per pattern. Continuing with the example above, specifying the chain UTF-8|OCE on the pattern IRLIBSYR would cause the byte sequence searched for to be UTF-8-encoded, then OCE'd, as seen in Fig. 5, thus permitting us to search for hits for this pattern in OCE'd Outlook mailboxes directly, without any human expertise required.

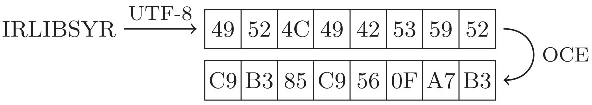


Fig. 5. The chain UTF-8|OCE applied to text "IRLIBSYR".

6. Interpreting hits in context

Search hits can, in isolation from their context, be uninformative, hard to interpret, or misleading. For example, each of "cabomba", "bombastic", and "dirty bomb" contains a hit for bomb—but the context, the characters on either side of the hit, are essential for recognizing the hit as relevant to aquatic plants, rhetoric, or terrorism.

It is possible to provide minimal context support by padding regexes with dots, thus rolling context into the hits. E.g., .{0,6}bomb.{0,6} grabs enough on either side of "bomb" in the example above to distinguish the three cases, at the expense of a massive, unacceptable slowdown in searching. The reason for poor performance when capturing leading context as part of a hit is straightforward: A match for .{0,6} can start anywhere in the input, so the regex engine has far more work to do when faced with .{0,6}bomb.{0,6} than with bomb.

The problem of context is exacerbated by encodings. One might simply wave off the difficulties with rolling context into the hit by manually examining the data, using less or a hex editor. While this is not an unreasonable thing to do if the data is ASCII, any investigator relying on this strategy will soon come to grief if the data is in a character encoding he is unable to read directly. Even worse, the data might be base64'd, OCE'd, or XOR'd with some constant byte value, which the investigator will then need to decode. Decoding the context manually could involve finding a starting point which available decoders recognize as valid, which costs yet more time and trouble. In such cases, the simple expedient of inspecting the data to see hit context falls down.

In Section 3, we showed how to put multipattern search to use for multiencoding search, and in Section 5, we described a method for chaining encodings together to make multiply-encoded data easily searchable. A consequence of these is that when we return a hit, we know which pattern and encoding chain produced the hit. Because we know the encoding chain which produced the hit, we can reverse that encoding chain to decode the context around the hit and display the decoded context to the user.

Sadly, this is not as straightforward nor are the results as univocal as we've made them sound. There are numerous ways in which decoding context might fail. The simplest is that the context is not in the same encoding as the hit. This can happen when searching serialized binary data, such as database files, where the fields adjacent to the one containing the hit might have different types from the hit. E.g., if an integer field precedes a hit in a UTF-16LE text field, then interpreting the part of the leading context overlapping the integer field as UTF-16LE is bound to fail to decode, or if it does decode, to produce garbage. Similarly, a hit and its context could be in different files. One way this could occur is when searching a raw disk image. If a hit runs up to the end of a file system block, there is no guarantee that the next byte on the disk belongs to the same file, either because the file containing the hit is fragmented or because the hit is actually at the end of its file. In either case, the trailing context could be anything, so there is no guarantee that it will decode along with the hit. Similar things can occur when searching unallocated space: The bytes preceding or trailing the hit could be associated with the hit, or could be part of some file which overwrote part of the hit's file and was then deleted itself. Another way that decoding context can fail is if the context was written with an incorrectly-implemented encoder. As we cannot anticipate all the possible ways in which encoders might be broken, we cannot properly decode improperly-encoded byte sequences.[9]

Finally, it could be the case that the hit's context fails to decode because the hit has been misidentified. For example, ASCII is interoperable with many encodings—UTF-7, UTF-8, all sixteen of the ISO-8859-X encodings, Shift JIS, and GB 18030, among others. This means that strings consisting entirely of 7-bit ASCII characters are represented by the same byte sequence in many encodings. Consequently, you cannot deduce the encoding from such sequences, and you are likely to get *mojibake* if you have guessed the wrong encoding for the surrounding context. Mojibake (文字化け), literally "character transformation", is the delightful term borrowed from Japanese for the result of rendering text in the wrong encoding. If, for example, we searched the previous sentence (which we assert, for the purposes of the example, to be UTF-8) for the pattern Mojibake encoded in Windows-1252 with fifteen characters of trailing context, then our hit plus context would be reported as "Mojibake (æ–‡å —åŒ–ã ‘)". Fig. 6 shows the carnage in detail.[10]

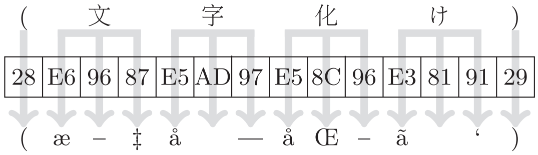


Fig. 6. UTF-8 misinterpreted as Windows-1252.

An even stranger thing can happen with UTF-8 and UTF-16LE: There are UTF-16LE strings which contain completely different UTF-8 strings as prefixes. For example the byte sequence which is "nonsense" in UTF-8 is 潮獮湥敳 in UTF-16LE (!) and not only that, both can manage to be correctly

[9] This puts one in mind of Charles Babbage being asked whether by putting the wrong figures into his Difference Engine, the right answers might come out.

[10] Note that the bytes AD and 81 are shown as blanks because in Windows-1252, byte AD is soft hyphen (U+AD), which is a nonprinting character, and byte 81 is unassigned.

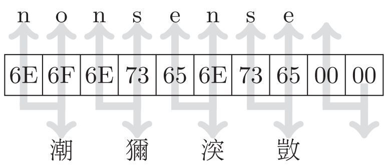


Fig. 7. Bytes in UTF-8 or UTF-16LE.

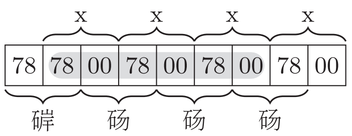
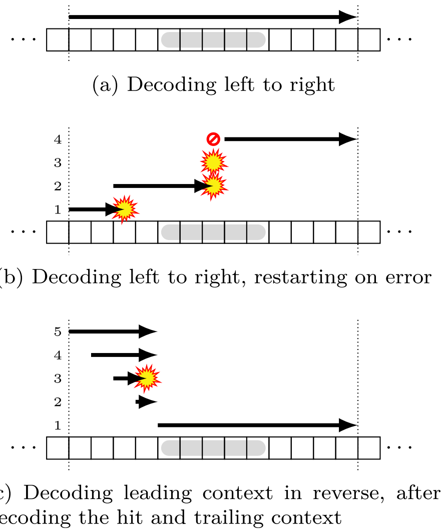


Fig. 8. Bad alignment when decoding UTF-16LE.



(a) Decoding left to right

(b) Decoding left to right, restarting on error

(c) Decoding leading context in reverse, after decoding the hit and trailing context

Fig. 9. Three methods of decoding hits and context.

null-terminated for their encoding (!!). Fig. 7 shows how. The explanation for this is that lowercase Latin letters in ASCII fall in the byte range [61,7A], while all four-digit code points starting with hex digits 6 or 7 are Chinese ideographs.[11]

We do not know whether 潮獮湥敳 is intelligible Chinese, but we can say that at the time of writing, Google returns 263,000 results for this character sequence. Regardless, we suppose that there must be some byte sequences which are both meaningful Chinese when decoded from UTF-16LE and words in some language using the Latin alphabet when decoded from UTF-8.

Finally, a word on what encoding bytes *are*: Bytes can be decoded or not, but don't themselves have an encoding. Bytes are just bytes. All we observe is whether we can decode a byte sequence successfully. As the above discussion makes plain, decoding a sequence without error is no guarantee that the bytes were written with that encoding. The lesson here from philosophy of science is that we can rule out hypotheses from observation, but never prove them conclusively. Hence, we can assert that a given sequence of bytes, say D8 00 DC 00 D8 00 DC 01, *could not* have been written with a (correct) UTF-8 encoder, but only that this sequence is *consistent* with being written by a UTF-16LE encoder. This is not to say that we cannot draw positive conclusions. The longer the sequence which can be validly decoded, the less likely it is to be accidental or mojibake. Encoded text tends to be highly structured when viewed as byte sequences. E.g., 100 random bytes will decode to 7-bit ASCII only one in 2^{100} times, because in 7-bit ASCII the high bit in each byte is always off. Similar things may be said about other encodings (though calculating the precise odds for, say, UTF-8, is not so easy.) Thus, *ceteris paribus*, it is probable that long sequences which decode successfully were written in the encoding they appear to have been written in.

6.1. Context decoding implementation

There are two implementation details regarding decoding hit context which are of interest: whether to measure the width of the context around a hit in bytes or in characters, and where in the context decoding should start.

Measuring context width in bytes has the advantage that we can trivially locate the start of leading context and the end of trailing context in the data, because these are simply offsets from the start and end of the hit, which we already know. Unfortunately, this is the only advantage of measuring context width in bytes. Measuring in bytes means that the limits of the context could fall in the middle of a character, and that the number of decoded characters for the same number of context bytes can vary from encoding to encoding. If the user wants decoded context, then we already know that the user is thinking in terms of characters, so presumably the number of context characters is more salient for the user than the number of bytes. Furthermore, unless the user is familiar with the technical details of the encoding in use, he will not know how many characters to expect a given number of bytes to produce when decoded. Therefore, we chose to let the user specify the width of the context in characters, not bytes.

The issue of where context decoding should start is more complex. A naïve strategy would be to start decoding at the leading end of the context, across the hit, and through to the trailing end, as seen in Fig. 9(a). This fails if, for example, the leading context is not a multiple of the bytes-per-character for the encoding, as seen in Fig. 8. If we start decoding UTF-16LE from the left end of the buffer, we get the Chinese ideographs shown below, while if we decode from the start of the hit (highlighted) we get a sequence of LATIN SMALL LETTER X. Clearly we must ensure that we have proper byte alignment when the decoder reaches the start of the hit.[12]

A little reflection on the examples above reveals that this is still inadequate, because there could be parts of the context which fail to decode. (E.g., suppose the context

[11] This applies also to uppercase Latin letters in ASCII, [41, 5A], since four-digit code points starting with 4 or 5 are also Chinese ideographs.

[12] Some encodings, such as UTF-8, are *variable width*, meaning that some characters produce more bytes than others when encoded. This makes proper byte alignment for left-to-right decoding even more vexed.

Please cite this article in press as: Stewart J, Uckelman J, Unicode search of dirty data, or: How I learned to stop worrying and love unicode technical standard #18, Digital Investigation (2013), http://dx.doi.org/10.1016/j.diin.2013.06.013

spans a field boundary in a database file.) We could work around this by advancing by one byte and resetting the decoder each time we encounter an invalid byte, as demonstrated in Fig. 9(b), but this too can fail catastrophically. Step 1 in the figure shows decoding from the left end of the context to byte offset 2, where the decoding fails. In step 2, the decoder restarts at offset 2, successfully decoding to offset 6, where decoding fails once again. In step 3, the decoder restarts at offset 6, but fails immediately. (This could occur if b_6 is an invalid start byte for the encoding.) In step 4, we give up on the byte at offset 6 as invalid, reset the decoder, and decode bytes 7–12. From the point of view of the user, this behavior is a disaster, as we failed to decode a byte in the middle of the hit.

As there is no guarantee that the hit and the bytes before it were encoded at the same time or are even related to it, the only way we can ensure that the hit is decoded successfully according to the encoding chain of the pattern which matched it is to begin decoding from the start of the hit, not from some point in its preceding context.

Furthermore, the problem of misdecoding the hit by beginning to decode to its left applies equally to decoding the leading context. Starting from the far left of the context window has the potential to create mojibake by the time the decoder reaches the end of the leading context. How, then, to decode the context to the left of the hit?

Consider the common case of a hit in the middle of some text. What we would ideally like to do is decode the longest sequences possible which are contiguous with the hit. For the trailing context, this means that decoding may continue rightwards after finishing the hit. For the leading context, it means that the most important character to decode is the one immediately before the hit, with importance decreasing leftwards. Suppose the hit is the byte sequence $b_m \ldots b_n$. Then we first try to decode only b_{m-1}, the byte to the left of the hit. If that succeeds, we then try to decode $b_{m-2}b_{m-1}$, and continue to extend our decoding window to the left one byte at a time until either decoding fails or we reach the limit of the context. If we hit the context limit, we are done—the entirety of the leading context has decoded successfully. If decoding fails, we keep the longest successfully decoded sequence contiguous with the hit. An example of this appears in Fig. 9(c). In step 1, we decode the hit and the trailing context. In step 2, we successfully decode the byte offset -1 from the beginning of the hit. In step 2, we back up one byte to offset -2, and attempt to decode toward the hit from there, but decoding fails at offset -1. In step 3, we back up another byte and successfully decode toward the hit from offset -3. Finally, in step 4, we retreat one more byte and successfully decode from offset -4 to the start of the hit.

This procedure guarantees both that the hit will decode successfully and that we decode the longest segments of context contiguous with the hit which we can while still remaining within the context window. The runtime of decoding the leading context this way is $O(n^2)$ in the worst case, due to needing decode $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ bytes, where n is the length of the leading context. This can be mitigated somewhat by clever choice of the decoding order: If we can successfully decode from offset o to the start of the hit, then there is no need to try decoding from p to the start of the hit, for any offset $p > o$. As decoders are fast and n will generally be small, this should not pose a significant problem when presenting decoded context for display purposes.[13]

Table 2
Search performance using lightgrep via bulk_extractor.

Image	Size	Hits	Bandwidth
Charlie-2009-11-12.E01	10 GB	543,719	131.7 MB/s
Jo-2009-11-12.E01	14 GB	536,906	123.8 MB/s
Pat-2009-11-12.E01	12 GB	522,220	124.7 MB/s

7. Search in practice

In cooperation with the Naval Postgraduate School, the lightgrep search engine has been open-sourced and integrated into bulk_extractor. It provides an alternate search mechanism to the GNU Regex and TRE libraries used in bulk_extractor's "scan_find" plugin and generally offers much greater performance. The International Components for Unicode (ICU) library is used by lightgrep to provide the definitions of code point names and properties, as well as mappings for legacy codepages. ICU nicely solves the daunting enumerative task of supporting Unicode and staying up-to-date as the standard evolves. (We do not use ICU's own regular expression library, as it lacks support for multipattern searches.)

While use of multiple encodings increases automata size substantially, search bandwidth with bulk_extractor is adequate for most sets of patterns. As an example, Table 2 shows the results of running the list of 10,198 U.S. place names in Grady Ward's Moby word list against images from the NPS 2009 M57 Patents-Redacted scenario in the NPS Digital Corpora. UTF-8 and UTF-16LE encodings were enabled for all these terms, effectively doubling the set of patterns. In all runs, processing bandwidth remained around 100 MB/s on a 12-core Xeon workstation, with the evidence stored on a three 7200 RPM drives in a RAID5. Owing to some short names and homonyms in the place names list, many hits were not relevant to an investigation, but we believe this illustrates the capability.

7.1. *Language identification with properties*

When conducting an investigation, it is useful to identify text in foreign languages. With the ability to form character classes from Unicode properties, it can be tempting to use patterns such as

[13] If leading context longer than is feasible to decode using a quadratic algorithm is required, it should be possible to decode the leading context from right to left in $O(n)$ time, i.e., decoding each byte only once—but not with off-the-shelf decoders, such as the ones provided by ICU, which are expecting to read left-to-right. While writing a right-to-left decoder for any given stateless encoding is unlikely to be difficult, writing one for each of the more than 200 encodings ICU supports is a daunting task. Decoding stateful encodings right-to-left introduces the additional complication that we may need to track a set of possible encoder states as we work our way rightward. We doubt that the collective difficulty of writing and testing right-to-left decoders for all encodings one might encounter is justified by the amount of use these decoders would receive over the simpler quadratic left-to-right, backtracking versions.

Please cite this article in press as: Stewart J, Uckelman J, Unicode search of dirty data, or: How I learned to stop worrying and love unicode technical standard #18, Digital Investigation (2013), http://dx.doi.org/10.1016/j.diin.2013.06.013

$$\backslash p\{script = Hangul\}\{5,30\}(\backslash p\{Common\}\{1,10\} \backslash p\{script = Hangul\}\{1,30\})\{4,\}$$

which would identify likely Korean text. Unfortunately, this is problematic. The most significant problem is that such text could still be gibberish—regular expressions are not equal to the challenge, generally. Another problem, though, is that the likelihood of false positives depends on the number of code points allowed as well as the chosen encoding. Some legitimate patterns may accept clearly erroneous strings given the wrong encoding. Windows-1256, an Arabic encoding, represents ARABIC LETTER YEH BARREE (ے) as byte FF, and it is not uncommon to encounter sectors full of byte FF in digital evidence. There is also the problem that Unicode script names also include non-letter characters, making it harder to construct regular expressions that match only on words. Finally, of course, many languages share the same script, so positive identification is not possible with script properties alone.

Language identification is a well-studied problem, however, and systems often use Markov chains to great effect (Brown, 2012). However, statistical techniques are typically more intensive than regular expression searching. What's more, language models exist at the level of characters and have nothing to do with how characters are encoded to bytes. There is still perhaps a role for regular expressions to play in identifying sections of possible text that could then be categorized by such models. The advantage of such a hybrid approach would be that the language models would only care about characters, not bytes, and that only a subset of all models would have to be considered, depending on the script. Accessing only likely models to classify a section of data could work well with modern CPU caches.

Finally, we note that the principle of using Unicode properties to restrict the set of permitted characters is far preferable to using the dot (.) metacharacter. Reducing the number of possible byte values allowed in an automaton generally results in significant performance increases, not to mention greatly reduced false positives.

8. Conclusion

While the esoterica of Unicode and UTS #18 may seem daunting and the number of encodings overwhelming, lightgrep serves as an existence proof that it is feasible to conduct Unicode-aware searches of digital media without knowledge of the encodings used. Multipattern search allows for the simultaneous searching of patterns in different encodings in their raw byte forms, without having to worry about decoding. The features proposed in UTS #18 are a guide to the library implementer and to the user in how to conduct Unicode-aware searches. Encoding chains are a declarative mechanism for confronting a number of the stateless obfuscation schemes commonly encountered in forensics. Finally, the ability to decode matches and their surrounding context into UTF-8 means that investigators and developers who use lightgrep need not confront head-on the variety of encodings found in digital evidence.

References

Barnett M. mrab-regex-hg: new implementation of python regex 2013. online.

Brown RD. Finding and identifying text in 900+ languages. Digital Investigation 2012;9(Suppl.):S24–43.

Davis M, Heninger A, editors. Unicode technical standard #18: unicode regular expressions. The Unicode Consortium; July 2012. Revision 15 [online].

Davis M, editor. Unicode standard annex #29: unicode text segmentation. The Unicode Consortium; September 2012. Revision 21 [online].

Garfinkel SL. bulk_extractor. https://github.com/simsong/bulk_extractor [online].

IBM. Regular expressions. ICU user guide 2013. online.

Oracle. java.util.regex. pattern class javadoc 2013. online.

perlunicode man page, 5.16.2. http://perldoc.perl.org/perlunicode.html [online].

Stewart J, Uckelman J. Searching massive data streams using multipattern regular expressions. In: Peterson G, Shenoi S, editors. Adv. in Digital Forensics, VII. Springer; 2011. p. 49–63.

The Unicode Consortium. The Unicode standard, version 6.2.0. http://www.unicode.org/versions/Unicode6.2.0/ [online].

Wallace DF. Infinite jest 1996.