TALLINN UNIVERSITY OF TECHNOLOGY

TUT Centre for Digital Forensics and Cyber Security

Department of Computer Science

ITC70LT

Dipl.-Ing. Jens Getreu 130546IVCMM

Forensic-Tool Development with Rust

Master thesis

Prof. Olaf Manuel Maennel

Supervisor

Tallinn 2017

Table of Contents

List of Figures

List of Tables

Preface

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Jens Getreu

Abstract

Within the framework of this study the suitability of the *Rust* ecosystem for forensic tool development was evaluated. As case study, the tool *Stringsext* was developed. Starting from analysing the specific requirements of forensic software in general and those of the present case study, all stages of the software development life-cycle have been executed, up to the first production release. *Stringsext* is a reimplementation and enhancement of the *GNU-strings* tool, a widely used program in forensic investigations. *Stringsext* recognizes Cyrillic, CJKV characters and other scripts in all supported multi-byte-encodings while *GNU-strings* fails in finding these in UTF-16 and other encodings.

During the case study it has become apparent that the *Rust* ecosystem provides good support for secure coding principles and unit testing. Furthermore, the benchmarks showed a satisfactory performance of the resulting *Stringsext* binaries comparable to the original *C* version.

This thesis is written in English and is 81 pages long, including 9 chapters, 19 figures and 11 tables.

Annotatsioon

Käesoleva uurimustöö eesmärgiks on analüüsida programmeerimiskeele *Rust* ökosüsteemi sobivust kohtuekspertiisis kasutatava tarkvara loomiseks. Sellel eesmärgil arendati välja tööriist Stringsext. Läbiti kõik tarkvaraarenduse tsüklid, kohtuekspertiisi-tarkvara valdkonnaspetsiifiliste nõuete analüüsist kuni valmis tarkvaraversioonini. *Stringsext* on *GNU-strings*'i — kohtuekspertiisis laialdaselt kasutatava tööriista — edasiarendus ja täiendus. *Stringsext* toetab kirillitsa ja CJKV-tähemärkide otsingut mitmebaidilist kodeeringut kasutavast tekstist, sh. ka kodeeringud, mida *GNU-strings* ei toeta, näiteks UTF-16.

Töö tulemusena ilmnes, et *Rust*'i ökosüsteem pakub head tuge turvalisusele keskenduva tarkvara arendamiseks ja moodultestide (*Unit* test) kirjutamiseks. Lisaks näitasid reeperid (*benchmark*) et tarkvara jõudlus oli võrreldav programmeerimiskeeles C kirjutatud *GNU-strings*'ga.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 81 leheküljel, 9 peatükki, 19 joonist, 11 tabelit.

Table of abbreviations and terms

ASCII	American Standard Code for Information Interchange.
API	Application Programming Interface.
BIG5	Chinese character encoding method.
BOM	Unicode byte order mark.
CJK	Chinese, Japanese, and Korean languages
CJKV	Chinese, Japanese, Korean and Vietnamese languages
CVE	Common Vulnerabilities and Exposures.
CWE	Common Weakness Enumeration.
EUC-JP	Multibyte character encoding system used primarily for Japanese, Korean, and simplified Chinese.
GNU	Recursive acronym for “GNU’s Not Unix!” used for an extensive collection of computer software.
KOI8-R	Character encoding, designed to cover Russian, which uses a Cyrillic alphabet.
NIST	National Institute of Standards and Technology.
TDD	Test Driven Development.
UTF	Unicode Transformation Format.
WHATWG	Web Hypertext Application Technology Working Group.

Chapter 1. Introduction

My first interest in the Rust programming language woke in a cryptography seminar where the participants where asked to break encryption schemes. Some of these exercises required a lot of computational power. This is why I was looking for an alternative to Python, which I normally use for this purpose. Finally, I came up with an - in this context - uncommon choice: the Rust programming language. I chose it mainly for its *zero cost abstractions* (cf. Section 5.3, “Zero-Cost Abstractions”) resulting in efficient code comparable to *C* and *C++*. Building on this initial experience, I implemented more projects in Rust and discovered some of its outstanding properties, e.g. memory safety (cf. Section 5.1, “Memory safety”), making it interesting, in particular, for IT-forensics.

Later, as part of a joint project, I worked in a team together IT-forensic experts, where I became acquainted with tools and methods customary in the sphere of forensics. Many software products e.g. *Forensic Toolkit (FTK)* or *XRY* encompass a workflow with a large variety of specialized tools for data acquisition and analysis. These tools are very handy and give a quick overview of artefacts that could be relevant for the present case. Although the software usually covers the most common data structures, not all can be analysed automatically. This is why forensic practitioners use a set of little specialized utilities like the Unix commands `file` or `strings` and many others. The latter, hereafter referred as *GNU-strings*, is a program that extracts ASCII characters from arbitrary files. It is mainly useful for determining the ASCII contents of non-text files (cf. Chapter 3, *GNU-strings in forensic examination*). *GNU-strings*' main limitation is that it has no multi-byte-encoding support.

The software tool *Stringsext*, developed in this present work, is meant to fill this gap by implementing multi-byte-encoding support. Special requirements relative to forensic tool development (cf. Chapter 2, *Tool Requirements in Digital Forensics*) lead us to an experiment: Implementing a forensic tool in the very young and innovative programming language Rust! Is Rust a suitable choice? The following case study will provide some answers and guidelines for similar projects. *Stringsext's* source code is publicly available [1] under: `https://github.com/getreu/stringsext`. The project's main page has links to the developer documentation and to the compiled binaries for various architectures.

What are the special requirements in the field of digital forensics?
Digital Forensics also known as digital forensic science is a branch of forensic science studying crime and its traces. “Traces are the most elementary information that results from crime. They are silent witnesses that need to be detected, analysed, and understood to make reasonable inferences about criminal phenomena, investigation or demonstration for investigation and court purposes” [2 p. 14]. The branch science dealing with digital traces and digitised information is referred as *digital forensic science*. It can be described as “the process of identifying, preserving, analysing and presenting digital evidence in a manner that it legally acceptable.” [3 p. 12]. When digital traces are presented in court to support an assertion the term *digital evidence DE* or *electronic evidence* is in common use. In this work I use the term *digital evidence* following common practise in Great Britain.

Most human interaction with electronic devices leaves traces in some electronic memory. In cases the user does not directly communicate with its device additional traces may also be found in all intermediate (network) devices. Due to the cross-linked nature of computer systems the total amount of data that needs to be taken into consideration when investigating a crime is enormous. In this ocean of information the investigator has to find specific drops of information constituting digital evidences. Furthermore, imagine someone throws a stone in the sea. It will change the state of the water particles in various places, but only a tiny share of this change is suitable to prove that the stone was indeed thrown in water.

Following the principles in Transactional Analysis as founded by the psychologist Eric Berne, the term “transaction” can be defined as the smallest atomic interaction in a human - computer system communication. For digital forensic practitioners the well-known and well documented cause-effect relationships between human transactions and digital traces is of utmost importance. A cause-effect-relationship is usually part of a non linear chain of events. For example an attacker may send "phishing" emails to its victims that try to lure them to identity-stealing sites. The stolen identity is then sold and will be used in other crimes. Behind the scenes many systems are involved in such a scenario. A typical transaction of interest could be “the user has opened a browser window and visited the site xy” which leaves traces in some computer memory. In the domain of digital forensics an observation of a well known cause-effect-relationship between an electronic trace (effect) and what has happened (cause) is an called **artefact**. It

embodies any “item of interest that help an investigation move forward.” [4 p. 125]. A more formal definition named *Curated (digital) Forensic Artefact (CuFA)* proposed in [4 p. 131] embraces that it must:

- be curated via a procedure which uses forensic techniques.
- have a location in a useful format (when applicable).
- have evidentiary value in a legal proceeding.
- be created by an external force/artificially.
- have antecedent temporal relation/importance.
- be exceptional (based on accident, rarity, or personal interest).

Forensic examiners - the law enforcement personnel who deal with digital evidence - face inter alia two challenges:

1. to collect and to preserve the huge amount of data that may be related to a crime and
2. to search and identify artefacts in the collected data.

The latter aspect includes so called *string search* which is useful when dealing with unknown binaries . Most executable binary code contains human readable character sequences called strings. A very common used program to extract strings from a binary executable code is the so called *GNU-strings* program. Also, the software tool *Stringsext* developed in this present work is made for this purpose: The new development is designed to overcome some of *GNU-strings* shortcomings. Where possible, it maintains *GNU-strings*' user-interface.

Chapter 2. Tool Requirements in Digital Forensics

This chapter describes general requirements towards forensic tools. They partly emerge from legal and technical demands and motivate, inter alia, the choice of the programming language Rust.

2.1. Tool validation

Like in other established forensic disciplines the forensic soundness or reliability of digital evidence is determined by the validity and correctness of forensic software used in examination. In other words, to guarantee that the digital evidence is forensically sound, all tools used to collect, preserve and analyse digital evidences must be validated. Tool validation can also be formally required to comply with standards like the *ISO 17025 Laboratory Accreditation standard*.

It should be noted that the forensic community's definition of *validation* and *verification* differs from what used in software engineering. Two commonly used definitions state the following:

A short and catchy definition was proposed by Beckett and Slay [5]:

Validation
: is the confirmation by examination and the provision of objective evidence that a tool, technique or procedure functions correctly and as intended.

Verification
: is the confirmation of a validation with a laboratories tools, techniques and procedures.

It means that establishing a reliable technical method to observe a cause and effect relation between a human action and a resulting artefact is called *validation*. The test, whether a technical device is suitable or not to execute the above method reliably, is called *verification*.

Craiger [6 p. 92] defines *validation* and *verification* as follows:

Software verification
: provides objective evidence that the design outputs of a particular phase of the software development life cycle meet all the specified requirements for that phase. *Software verification* looks for **consistency, completeness, and correctness of the software and its supporting documentation, as it is being developed**, and provides support for a subsequent conclusion that software is validated. Software testing is one of many verification activities intended to confirm that software development output meets its input requirements. Other verification activities include various static and dynamic analyses, code and document inspections, walkthroughs, and other techniques.

Software validation
: is a part of the design **validation for a finished device**…considers software validation to be 'confirmation by examination and provision of objective evidence that software specifications conform to user needs and intended uses, and that the particular requirements implemented through software can be consistently fulfilled.' In practice, software validation activities may occur both during, and as at the end of the software development life cycle to ensure that all requirements have been fulfilled. …the validation of software typically includes **evidence that all software requirements have been implemented correctly and completely** and are traceable to system requirements. A conclusion that software is validated is highly dependent upon comprehensive software testing, inspections, analyses, and other verification tasks performed at each stage of the software development life cycle.

Common to both definitions of validation is the mapping of the tool's requirements to tests confirming that they fully and correctly implemented. At first glance the above approach might seem simple to implement but in many cases it is impossible to carry out:

> Traditional research discourse on tool testing in this discipline concerns validation of a tool, that is, all the functions of a tool, and with the failure of a validation of a tool the traditional thinking is to invalidate the tool. In most cases forensic tools are quite complex and provide hundreds of specific functions, of which only a few may ever be used by an examiner. Even trivial testing of all functions of a forensic tool for every version

under all conditions, conservative estimates would indicate significant cost [5].

To cope with this difficulty Beckett and Slay [5] suggest a model so called *Model of tool neutral testing* or *functionality oriented validation*. Instead of testing if a software product meets all its requirements an independent set of forensic functions and their specifications is defined. This allows to decouple the validation procedure from the implementation of the forensic tool itself. A forensic function is an activity required in forensic investigation that produces known valid results for a given set of test cases.

Figure 2.1. Model of tool neutral testing

The first difficulty consists in breaking down the multitude of activities in forensic investigation in function categories and subcategories as shown in Figure 2.2, “An overview of searching function” [3 p. 17].

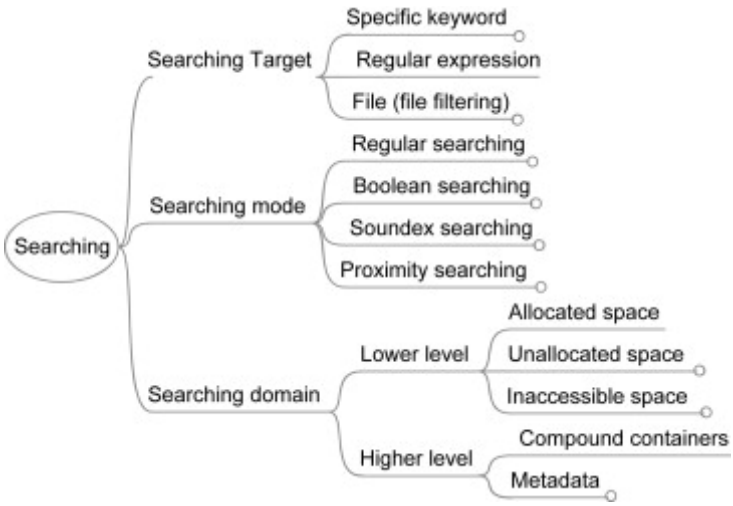

Figure 2.2. An overview of searching function

The search target mapping as shown below illustrates under the subcategory “Character encoding” the main deficit of *GNU-strings* supporting only ASCII encoding. In global cyberspace forensic tools must identify a multi-

tude of encodings. This leads us to the main motivation and requirement of *Stringsext*: Section 4.2, “Character encoding support”

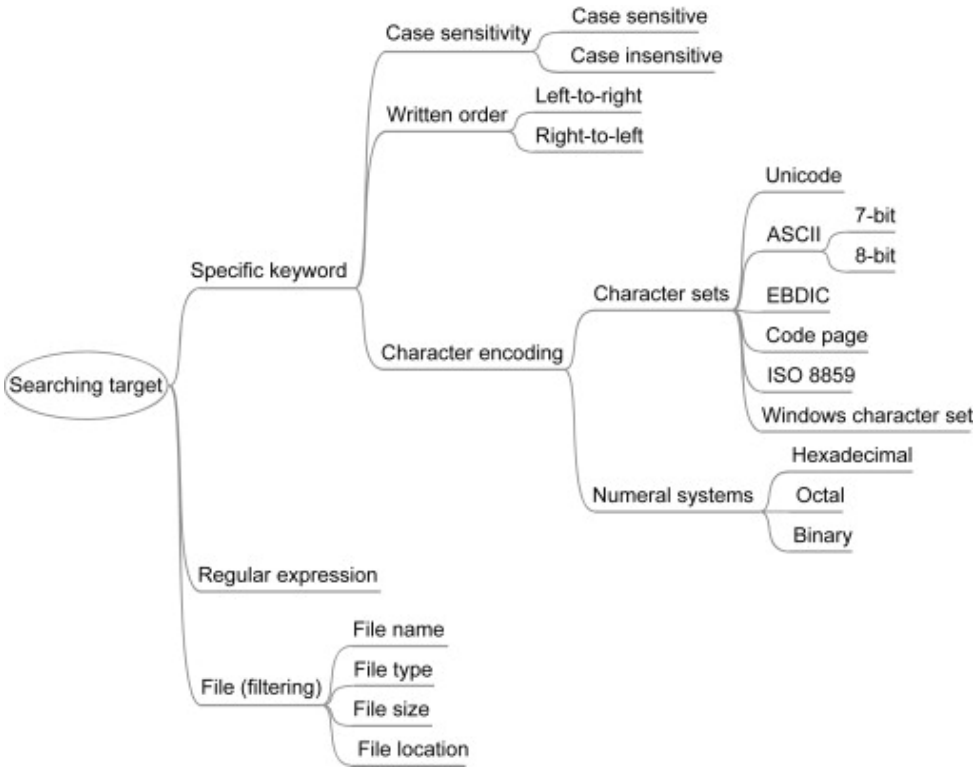

Figure 2.3. The search target mapping

The *functionality oriented validation* can be classified as “black box testing” examining functionality without any knowledge of internal implementation, without seeing the source code. “Black box testing” of functions and their specifications allows conducting numerous tests with acceptable costs. It requires test cases with known valid results. With *Stringsext* this approach is used to test the correctness of the implementation when dealing with large real-world data (cf. Section 4.8, “Functionality oriented validation”).

When the internal computation is as complex as in *Stringsext*, “white box testing” is essential. The method chosen in the present development “test harness” is detailed in the Section 4.7, “Automated test framework”.

2.2. Security

The relation between the criminal and the forensic examiner can be described as follows: “Make it hard for them to find you and impossible for them to prove they found you” [7]. Have your recognised the statement? Is widely cited when it comes to define anti-forensics. This traditional "hide and seek" relation might soon take a new dimension: Eggendorfer [8]

stresses with good reasons that forensic tools are software too and therefor vulnerable to attacks.

GNU-strings is part of the *GNU binutils* collection which became publicly available in 1999 [9]. Today it has reached the notable age of 17 years. *GNU-strings* is a comparatively small program with 724 lines of code only. It is all the more surprising that in 2014 the security researcher Zalewski discovered a serious security vulnerability *CVE-2014-8485* [10].

> The setup_group function in `bfd/elf.c` in `libbfd` in GNU binutils 2.24 and earlier allows remote attackers to cause a denial of service (crash) and possibly execute arbitrary code via crafted section group headers in an ELF file.
>
> — CVE-2014-8485

Zalewski headlined his bug report “Don’t run strings on untrusted files.” Needless to say that this advice can not be followed in the context of a forensic investigation. In the meantime the bug was fixed but users remain confused and bewildered.

The above bug is part of a vulnerability class related to memory safety problems. *GNU strings* is written in C, a language whose abstractions can not guarantee memory savety. In order to exclude potential vulnerabilities of the same kind from the start, *Stringsext* was developed with the *Rust* programming language which is discussed further in the Chapter 5, *The Rust programming language*.

2.3. Code efficiency

The searching domain in forensic investigations is often as large as the seized data-carrier. Nowadays hard-disk images hold several TiB of data. Memory images of the RAM are smaller, but still some GiB in size. In order to address so big search domains, forensic software must operate very efficiently. This is why forensic software is often programmed in C or C ++. But not only the programming language matters: Efficient code requires carefully chosen abstractions, efficient algorithms avoiding unnecessary data-copies and program-loops.

Chapter 3. *GNU-strings* in forensic examination

This chapter first analyses *GNU-strings*' limitations concerning multi-byte-encodings and international scripts (cf. Section 3.1, “Test case 1 - International character encodings”). Further, a use case shows how *GNU-strings* is typically used in forensic examination (cf: Section 3.2, “Typical usage”). Based upon this we derive a set of requirements for *Stringsext* (cf: Section 3.3, “Requirements derived from typical usage”).

Forensic examiners use the GNU program *strings* to get a sense of the functionality of an unknown program. E.g. extracted URLs to malicious sites can be an indicator of malware. Also, user prompts, error messages, and status messages can give hints for further investigation.

3.1. Test case 1 - International character encodings

As discussed above the main motivation for developing *Stringsext* are the missing multi-byte character encoding semantics in *GNU-strings*. *GNU-strings* encoding support consists of a rudimentary filter accessed with the option `--encoding`. For details see the Table 3.1, “*GNU-strings* manual page (extract)”. How well does *GNU-strings* detect Unicode?

The Figure 3.1, “Test case international character encodings” shows the content of a text file chosen as test case.

```
Arabic: A lie has short legs. (Lit: The rope of lying is short.)
حبل الكذب قصير

Chinese: Teachers open the door. You enter by yourself.
師傅領進門，修行在個人

French: pasta
Les pâtes

Greek: History
Ιστορία

German: Greetings
Viele Grüße

Russian: Congratulations
Поздравляю

Eurosign (U+20AC)
€

Violinschlüssel (U+1D11E)
𝄞
```

Figure 3.1. Test case international character encodings

The above file is then is converted into different encodings using the following script:

Test case preparation

```
#!/bin/sh
cp orig.txt encoded-utf8.txt
iconv -f utf8 -t utf16le orig.txt >encoded-utf16le.txt
iconv -f utf8 -t utf32le orig.txt >encoded-utf32le.txt
iconv -f utf8 -t utf16be orig.txt >encoded-utf16be.txt
iconv -f utf8 -t utf32be orig.txt >encoded-utf32be.txt
```

In order to observe *GNU-strings* Unicode detection capabilities, all the above test-files are searched for valid graphic strings with the command `strings` using all possible variation of its encoding filter.

The following figures show *GNU-strings* output.

```
strings -f -t x encoded*
encoded-utf16be.txt:     125 LW(P
encoded-utf8.txt:         3 Arabic: A lie has short legs. (Lit: The rope o
encoded-utf8.txt:        61 Chinese: Teachers open the door. You enter by
encoded-utf8.txt:        bc French: pasta
encoded-utf8.txt:        ca Les p
encoded-utf8.txt:        d6 Greek: History
encoded-utf8.txt:        f5 German: Greetings
encoded-utf8.txt:       107 Viele Gr
encoded-utf8.txt:       116 Russian: Congratulations
encoded-utf8.txt:       145 Eurosign (U+20AC)
encoded-utf8.txt:       15c Violinschl
encoded-utf8.txt:       168 ssel (U+1D11E)
```

Figure 3.2. GNU-strings, single-7-bit

```
strings -f -t x -e S encoded* #S = single-8-bit-byte characters
encoded-utf16be.txt:     116 ^+P
encoded-utf16be.txt:     11c
encoded-utf16be.txt:     122 O LW(P
encoded-utf16le.txt:     116 +^ P
encoded-utf16le.txt:     11b
encoded-utf16le.txt:     121 (W
encoded-utf8.txt:         0 Arabic: A lie has short legs. (Lit: The rope c
encoded-utf8.txt:        44 حبل الكذب قصير
encoded-utf8.txt:        61 Chinese: Teachers open the door. You enter by
encoded-utf8.txt:        99 師傅領進門，修行在個人
encoded-utf8.txt:        bc French: pasta
encoded-utf8.txt:        ca Les pâtes
encoded-utf8.txt:        d6 Greek: History
encoded-utf8.txt:        e5 Ιστορία
encoded-utf8.txt:        f5 German: Greetings
encoded-utf8.txt:       107 Viele Grüße
encoded-utf8.txt:       116 Russian: Congratulations
encoded-utf8.txt:       12f Поздравляю
encoded-utf8.txt:       145 Eurosign (U+20AC)
encoded-utf8.txt:       15c Violinschlüssel (U+1D11E)
encoded-utf8.txt:       177 𝄞
```

Figure 3.3. GNU-strings, single-8-bit option

```
strings -f -t x -e l encoded* #l = 16-bit littleendian
encoded-utf16le.txt:       2 Arabic: A lie has short legs. (Lit: The rop
encoded-utf16le.txt:      a6 Chinese: Teachers open the door. You enter
encoded-utf16le.txt:     130 French: pasta
encoded-utf16le.txt:     14c Les p
encoded-utf16le.txt:     162 Greek: History
encoded-utf16le.txt:     192 German: Greetings
encoded-utf16le.txt:     1b6 Viele Gr
encoded-utf16le.txt:     1d0 Russian: Congratulations
encoded-utf16le.txt:     21a Eurosign (U+20AC)
encoded-utf16le.txt:     244 Violinschl
encoded-utf16le.txt:     25a ssel (U+1D11E)
```

Figure 3.4. GNU-strings, 16-bit little-endian option

```
strings -f -t x -e b encoded* #b = 16-bit bigendian
encoded-utf16be.txt:       2 Arabic: A lie has short legs. (Lit: The rop
encoded-utf16be.txt:      a6 Chinese: Teachers open the door. You enter
encoded-utf16be.txt:     130 French: pasta
encoded-utf16be.txt:     14c Les p
encoded-utf16be.txt:     162 Greek: History
encoded-utf16be.txt:     192 German: Greetings
encoded-utf16be.txt:     1b6 Viele Gr
encoded-utf16be.txt:     1d0 Russian: Congratulations
encoded-utf16be.txt:     21a Eurosign (U+20AC)
encoded-utf16be.txt:     244 Violinschl
encoded-utf16be.txt:     25a ssel (U+1D11E)
```

Figure 3.5. GNU-strings, 16-bit big-endian option

```
strings -f -t x -e L encoded* #L = 32-bit littleendian
encoded-utf16le.txt:       2 Arabic: A lie has short legs. (Lit: The rop
encoded-utf16le.txt:      a6 Chinese: Teachers open the door. You enter
encoded-utf16le.txt:     130 French: pasta
encoded-utf16le.txt:     14c Les p
encoded-utf16le.txt:     162 Greek: History
encoded-utf16le.txt:     192 German: Greetings
encoded-utf16le.txt:     1b6 Viele Gr
encoded-utf16le.txt:     1d0 Russian: Congratulations
encoded-utf16le.txt:     21a Eurosign (U+20AC)
encoded-utf16le.txt:     244 Violinschl
encoded-utf16le.txt:     25a ssel (U+1D11E)
```

Figure 3.6. GNU-strings, 32-bit little-endian option

```
strings -f -t x -e B encoded* #B = 32-bit bigendian
encoded-utf16be.txt:       2 Arabic: A lie has short legs. (Lit: The rop
encoded-utf16be.txt:      a6 Chinese: Teachers open the door. You enter
encoded-utf16be.txt:     130 French: pasta
encoded-utf16be.txt:     14c Les p
encoded-utf16be.txt:     162 Greek: History
encoded-utf16be.txt:     192 German: Greetings
encoded-utf16be.txt:     1b6 Viele Gr
encoded-utf16be.txt:     1d0 Russian: Congratulations
encoded-utf16be.txt:     21a Eurosign (U+20AC)
encoded-utf16be.txt:     244 Violinschl
encoded-utf16be.txt:     25a ssel (U+1D11E)
```

Figure 3.7. GNU-strings, 32-bit big-endian option

Results

As shown in the Figure 3.3, “GNU-strings, single-8-bit option”, the encoding filter `-e S` is the only filter that finds international characters at all.

UTF-8 is the only encoding in which *GNU strings* is able to find international characters.

The Figure 3.4, “GNU-strings, 16-bit little-endian option” and the Figure 3.5, “GNU-strings, 16-bit big-endian option” confirm that with *UTF-16* no international characters are recognized. The same holds true for *UTF-32*: see Figure 3.6, “GNU-strings, 32-bit little-endian option” and Figure 3.7, “GNU-strings, 32-bit big-endian option”. This limitation is of partic-

ular importance in forensic investigations: The Microsoft-Windows operating system handles Unicode characters in memory as 2 byte *UTF-16* words. As a result when dealing with Microsoft-Windows memory images, *GNU-strings* is not able to detect any international characters!

It should not be forgotten that *GNU-strings* can not analyse multi-byte encodings in general. This is why other very common encodings e.g. *big5* or *koi8-r* were not tested even though they are widely used.

The above-outlined limitations led to *Stringsext*'s main requirement: Section 4.2, “Character encoding support”.

3.2. Typical usage

The following script [1] shows how forensic examiners typically use the program *GNU-strings*`:

Typical usage of GNU-strings

```
#!/bin/bash
strings -a -t d $1 > $1.strings.temp                    ❶ ❷ ❸
strings -a -t d -e l $1 >> $1.strings.temp              ❹
strings -a -t d -e L $1 >> $1.strings.temp
strings -a -t d -e b $1 >> $1.strings.temp
strings -a -t d -e B $1 >> $1.strings.temp
strings -a -t d -e s $1 >> $1.strings.temp
strings -a -t d -e S $1 >> $1.strings.temp
sort -n -u -b $1.strings.temp > $1.strings              ❺ ❻ ❼ ❽
rm $1.strings.temp                                      ❾
```

Please refer to Table 3.1, “*GNU-strings* manual page (extract)” for details about the used options above.

The first and only parameter of the above script `$1` is the filename of the binary data to be examined.

❶ The examination is carried out by the `strings` command. `strings` is invoked in total seven times. Each run it scans the whole data, searches for valid graphic ASCII strings and appends its result to the temporary file `$1.strings.temp`.

[1] The script was kindly provided by an employee of the German CERT.

❷ `-a` means that the whole file is to be scanned, not only a part of it.

❸ The option `-t d` means that each line of output is prepended by the decimal offset indicating the location of the string that is found.

❹ Even though `strings` can only recognise pure ASCII encodings the option `-e` allows specifying some variations concerning the memory layout in which the characters are stored. For example `-e b` means that one ASCII character is stored in two bytes (16 bit) in Big-Endian order. For the other variants please refer to Table 3.1, "*GNU-strings* manual page (extract)".

❺ It may surprise that `strings` with `-t d` is set up to print the offset in decimal although hexadecimal notation is generally preferred when dealing with binary data. The reason lies in the post-treatment performed in this line: The `-n` or `--numeric-sort` option tells the `sort` command to interpret the beginning of the line as decimal number and sort criteria. Since `sort` is limited to the decimal number notation, `strings` is tied to it too. Please refer to Table 3.2, "*sort* manual page (extract)" for details about `sort`.

❻ The option `-u` tells `sort` to omit repeated lines. This only works because the concatenated file does not contain labels indicating which of the `strings` run has printed a given line. As the information is lost anyway, it makes sense to remove identical lines. Anyway, it is preferable to indicate the encoding together with the finding.

❼ The `-b` option interfaces with `strings` output formatting: offset-numbers are indented with spaces.

❽ The sorted and merged output of the strings search is stored in the file `$1.strings`.

❾ The temporary file `$1.strings.temp` is deleted.

3.3. Requirements derived from typical usage

With the above observations of how *GNU-strings* is used, I define the following requirements for the new development *Stringsext* in order to improve its usefulness and/or usability:

1. GNU-strings can not scan for more than one encoding simultaneously. This scales badly when more encodings are of interest. In the above example, the same input data is scanned seven times. Seeing that the examined data is usually stored on relatively slow hard-disks or even network shares the new scan algorithm should perform every scan for

a certain encoding *concurrently*. The above observation leads to requirement detailed in the Section 4.3, “Concurrent scanning”.

2. Large binary data usually contains strings in several encodings. Hereby it frequently happens that a byte sequence represents valid strings in more than one encoding. The overall context together with additional knowledge from other sources will lead to an assumption of the original encoding of a given byte sequence. For this to be practical *Strings-utf* presents possible valid string interpretations next to each other. Technically it requires that *Stringsext* merges the output of the different encoding scanners before printing. The above observation leads to requirement detailed in the Section 4.5, “Merge findings”.

3. The shift to concurrent processing and subsequent merging solves also a shortcoming in the approach shown in Typical usage of GNU-strings: the doubled disc space consumption caused by the file `$1.strings.temp`. In order to avoid temporary files, the search field has to be divided into small chunks of some memory pages in size. Before starting to search in the next chunk the findings of all encoding scanners in the current chunk is merged in memory and printed. This way no temporary file is needed. The above observation leads to requirement detailed in the Section 4.4, “Batch processing”.

4. In the present example, the output of `strings` is forwarded to the `sort` command. Even though external sorting with *Stringsext* will not be necessary anymore due to its build in merging ability, other post-treatments like `grep` remain very useful. Therefore, *Stringsext* should provide a mode with a machine friendly output formatting for line oriented tools like `grep` or `agrep`. The above observation leads to requirement detailed in the Section 4.6, “Facilitate post-treatment”.

Table 3.1. GNU-strings manual page (extract)

```
NAME
       strings - print the strings of printable characters in files.

SYNOPSIS
       strings [-afovV] [-min-len]
               [-n min-len] [--bytes=min-len]
               [-t radix] [--radix=radix]
               [-e encoding] [--encoding=encoding]
               [-] [--all] [--print-file-name]
```

```
                [-T bfdname] [--target=bfdname]
                [-w] [--include-all-whitespace]
                [--help] [--version] file...
```

-a, --all
Scan the whole file, regardless of what sections it contains or whether those sections are loaded or initialized. Normally this is the default behaviour, but strings can be configured so that the -d is the default instead.

The - option is position dependent and forces strings to perform full scans of any file that is mentioned after the - on the command line, even if the -d option has been specified.

-e encoding, --encoding=encoding
Select the character encoding of the strings that are to be found. Possible values for encoding are: s = single-7-bit-byte characters (ASCII, ISO 8859, etc., default), S = single-8-bit-byte characters, b = 16-bit big-endian, l = 16-bit little-endian, B = 32-bit big-endian, L = 32-bit little-endian. Useful for finding wide character strings. (l and b apply to, for example, Unicode UTF-16/UCS-2 encodings).

-t radix, --radix=radix
Print the offset within the file before each string. The single character argument specifies the radix of the offset o for octal, x for hexadecimal, or d for decimal.

Table 3.2. sort manual page (extract)

```
NAME
       sort - sort lines of text files

SYNOPSIS
       sort [OPTION]... [FILE]...
       sort [OPTION]... --files0-from=F
```

-b, --ignore-leading-blanks
ignore leading blanks

-n, --numeric-sort
compare according to string numerical value

-u, --unique
with -c, check for strict ordering; without -c, output only the first of an equal run

Chapter 4. Specifications

In the Chapter 2, *Tool Requirements in Digital Forensics* and the Section 3.3, “Requirements derived from typical usage” we determined the needs for *Stringsext* from the user’s perspective. This chapter provides a precise idea of the problem is to be solved. It serves also as a guidance to implement tests of each technical requirement.

4.1. User interface

The user interface of *Stringsext* should reproduce *GNU-strings*' user interface as close as possible. Where applicable, options should follow the same syntax. When used in ASCII-only mode, the output of *Stringsext* should be bit-identical with *GNU-strings*' output.

4.2. Character encoding support

Besides ASCII, *Stringsext* should support common multi-byte encodings like UTF-8, UTF-16 big Endian, UTF-16 little Endian, KOI8-R, KOI8U, BIG5, EUC-JP and others. The string findings in these encodings should be presented in chronological order and merged. The user should be able to specify more than one encoding at the same time.

4.3. Concurrent scanning

Each search encoding specified by the user is assigned to a separate thread hereafter referred as “scanner”.

4.4. Batch processing

Because of the differing complexity of the decoding process depending on the chosen encoding, the scanners run at different speeds. In order to limit memory consumption it must be assured that the scanners do not drift apart. This is guaranteed by operating in batch mode: all scanners operate simultaneously on the same search field chunk. Only when all scanners have finished searching and reported their findings, the next chunk can be processed.

4.5. Merge findings

When a scanner completes the current search field chunk, it sends its findings to the *merger-thread*. When all threads' findings are collected, the merging algorithm brings them in chronological order. Then the *printer* formats the findings and prints them to the output channel.

4.6. Facilitate post-treatment

Stringsext should have at least one print mode allowing post-treatment with line-oriented tools like `grep`, `agrep` or a spreadsheet program. The output of the other modes should be optimised for human readability.

4.7. Automated test framework

To take into account the increased requirements of the forensic community in correctness and reliability the *test driven development* method should be applied. *Unit tests* programming various test cases check automatically for correct results. Furthermore, the chosen methodology makes sure that the unit-tests are working as intended. For details please refer to Chapter 6, *Software development process and testing*.

4.8. Functionality oriented validation

Well designed *unit testing* reduces the defect rate significantly. Unit testing allows to verify whether a piece of code produces valid output for a given test case. The difficulty consists in finding relevant test cases challenging all internal states of the program. Unfortunately no indicators arise from these tests on how the program behaves on input data other then the test cases. This is why tests under real world conditions are indispensable.

In addition to the mentioned *unit tests* *Stringsext* should be evaluated according the *functionality oriented validation* method. This common method to validate forensic software is discussed in detail in the Section 2.1, “Tool validation”. In the present case a comparative test should be executed as follows:

> The same hard-disk image of approximate 500MB is analysed twice: first with *GNU-strings* then with *Stringsext*. If both outputs are identical, the test is passed.

4.9. Efficiency and speed

This requirement emerges from special requirements on tools in forensic investigations which are detailed in the Section 2.3, “Code efficiency”.

Applied to *Stringsext* the following is required:

The programming language should

- allow a fine control over pointers and memory allocation,
- offer zero cost abstractions,
- no or minimal runtime.

Programming style and techniques should promote *efficient coding* by

- avoiding as much as possible copying the input data,
- carefully chosen abstractions,
- efficient algorithms avoiding unnecessary
  - data-copies and
  - program-loops.

4.10. Secure coding

In the narrow sense, “security coding” is more a design goal than a functional requirement. Secure coding denotes the practice of developing computer software in a way that reduces the accidental introduction of security vulnerabilities to a level that can be fully mitigated in operational environments. This reduction is accomplished by preventing coding errors or discovering and eliminating security flaws during implementation and testing.

From the code security point of view the requirement defined in the Section 4.2, “Character encoding support” is the most critical: The NIST *National Vulnerability Database* lists under the heading “character encoding” 22 vulnerabilities. To give an idea of the severity of this kind of vulnerability, here a short summary of the most recent one, published in September 2016, is CVE-2016-3861:

> LibUtils in Android 4.x before 4.4.4, 5.0.x before 5.0.2, 5.1.x before 5.1.1, 6.x before 2016-09-01, and 7.0 before 2016-09-01

mishandles conversions between Unicode character encodings with different encoding widths, which allows remote attackers to execute arbitrary code or cause a denial of service (heap-based buffer overflow) via a crafted file, aka internal bug 29250543 [11]:

Table 4.1. CVSS Severity (version 2.0)

CVSS v2 Base Score	9.3 HIGH
Vector	(AV:N/AC:M/Au:N/C:C/I:C/A:C) (legend)
Impact Subscore	10.0
Exploitability Subscore	8.6

Table 4.2. CVSS Version 2 Metrics

Access Vector	Network exploitable - Victim must voluntarily interact with attack mechanism
Access Complexity	Medium
Authentication	Not required to exploit
Impact Type	Allows unauthorized disclosure of information; Allows unauthorized modification; Allows disruption of service

The technical cause of the CVE-2016-3861 vulnerability is an exploitable heap-based buffer overflow. Buffer overflows belong to the vulnerability category *memory safety issues* which are typical for the system programming languages *C* and *C++*.

To avoid similar vulnerabilities, *Stringsext* is implemented using the Rust programming framework. A short description of the programming language and its security guaranties can be found in the Chapter 5, *The Rust programming language*.

Chapter 5. The Rust programming language

This chapter presents some of Rust's core properties that led to the choice of implementing *Stringsext* in Rust.

In forensic tool development code efficiency (cf. Section 2.3, "Code efficiency") and security (cf. Section 2.2, "Security") is of primary importance. Rust supports these requirements with its *zero cost abstractions* and its *guaranteed memory safety*.

5.1. Memory safety

All memory-related problems in *C* and *C++* come from the fact that *C* programs can unrestrainedly manipulate pointer to variables and objects outside of their memory location and their lifetime. The Table 5.1, "Common weaknesses in C/C++ that affect memory" shows a selection of most common memory safety related vulnerabilities [12]. This is why memory safe languages like *Java* do not give programmers direct and uncontrolled access to pointers. The *Java* compiler achieves this with a resource costly runtime and a garbage collector. The related additional costs in terms of runtime resources exclude programming language like Java for most forensic tool development.

Table 5.1. Common weaknesses in C/C++ that affect memory

CWE ID	Name
119	Improper Restriction of Operations within the Bounds of a Memory Buffer
120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
125	Out-of-bounds Read
126	Buffer Over-read ('Heartbleed bug')
122	Heap-based Buffer Overflow
129	Improper Validation of Array Index
401	Improper Release of Memory Before Removing Last Reference ('Memory Leak')

415	Double Free
416	Use After Free
591	Sensitive Data Storage in Improperly Locked Memory
763	Release of Invalid Pointer or Reference

For many years program efficiency and memory safety seemed to be an insurmountable discrepancy. Now, after 10 years of development, a new programming language called *Rust* promises to cope with this balancing act. *Rust*'s main innovation is the introduction of semantics defining *data ownership*. This new programming paradigm allows the compiler to guarantee memory safety at *compile-time*. Thus, no resource costly runtime is needed for that purpose. In *Rust* most of the weaknesses listed in Table 5.1, “Common weaknesses in C/C++ that affect memory” are already detected at compile time. Moreover, *Rust*'s memory safety guarantees that none of these weaknesses can result in an undefined system state or provoke data leakage.

Rust's main innovation is the introduction of new semantics defining *ownership* and *borrowing*. They translate to the following set of rules which Rust’s type system enforces at compile time:

1. All resources (e.g. variables, vectors…) have a clear owner.
2. Others can borrow from the owner.
3. Owner cannot free or mutate the resource while it is borrowed.

By observing the above rules *Rust* regulates how resources are shared within different scopes. Memory problems can only occur when a resource is referenced by multiple pointers (aliasing) and when it is mutable at the same time. In contrast to other languages, *Rust*'s semantics allow the type system to ensure *at compile time* that simultaneous aliasing and mutation mutually exclude each other. As the check is performed at compile-time, no run-time code is necessary. Furthermore, *Rust* does not need a garbage collector: when owned data goes out of scope it is immediately destroyed.

Table 5.2. Ressource sharing in Rust

Resource sharing type	**Aliasing**	**Mutation**	**Example**
move ownership	no	yes	`let a = b`

Resource sharing type	Aliasing	Mutation	Example
shared borrow	yes	no	`let a = &b`
mutable borrow	no	yes	`let a = &mut b;`

The following code samples [13] illustrate how well the Rust compiler detects non-obvious hidden memory safety issues.

The following sample code returns a pointer to a stack allocated resource `s` that is freed at the end of the function: we find ourselves with a "Use after free" condition! The compiler aborts with the error message `s does not live long enough`.

Vulnerable code sample 1

```
fn as_str(data: &u32) -> &str {
      let s = format!("{}", data);
      &s
}
```

Here the corrected memory safe code:

Secure code sample 1

```
fn as_str(data: &u32) -> String {
      let s = format!("{}", data);
      s
}
```

The `push()` method in the next example causes the backing storage of `data` to be reallocated. As a result we have a dangling pointer, here `x`, vulnerability! The code does not compile in Rust.

Vulnerable code sample 2

```
let mut data = vec![1, 2, 3];
let x = &data[0];
data.push(4);
println!("{}", x);
```

Here the corrected memory safe version that compiles:

Secure code sample 2

```
let mut data = vec![1, 2, 3];
data.push(4);
let x = &data[0];
println!("{}", x);
```

5.2. Iterators

A very common group of programming mistakes is related to improper handling of indexes especially in loops, e.g. “CWE-129: Improper Validation of Array Index” (cf. Table 5.3, “Common weaknesses in C/C++ affecting memory avoidable with iterators”[12]).

Table 5.3. Common weaknesses in C/C++ affecting memory avoidable with iterators

CWE ID	Name
119	Improper Restriction of Operations within the Bounds of a Memory Buffer
125	Out-of-bounds Read
129	Improper Validation of Array Index

In addition to traditional imperative loop control structures, *Rust* offers efficient iteration with functional style iterators. Like in Haskel iterators are lazy and avoid allocating memory for intermediate structures (you allocate just when you call `.collect()`).

Besides performance considerations, iterators considerably enhance the robustness and safety of programs. They enable the programmer to iterate through vectors without indexes! The following code shows an example.

Vigenère cipher in Rust

```
fet p: Vec<u8> = s.into_bytes();   //plaintext
let mut c: Vec<u8> = vec![];    //ciphertext

for (cypherb, keyb) in p.iter()
    .zip(  key.iter().cycle().take(p.len())  ) {
        c.push(*cypherb ^ *keyb as u8);
    }
```

It must be noted that even with iterators *out of bounds*-errors may occur. Nevertheless, iterators should be preferred because they reduce the probability of errors related to indexes drastically.

5.3. Zero-Cost Abstractions

It is the language design goal *Zero-Cost Abstractions* that makes the *C/C++* language so efficient and suitable for system programming. It means that libraries implementing abstractions, e.g. vectors and strings, must be designed in a way that the compiled binary is as efficient as if the program had been written in Assembly. This is best illustrated with memory layouts: Figure 5.1, “Memory layout of a Rust vector” shows a vector in *Rust*. Its memory layout is very similar is to a vector in *C/C++*.

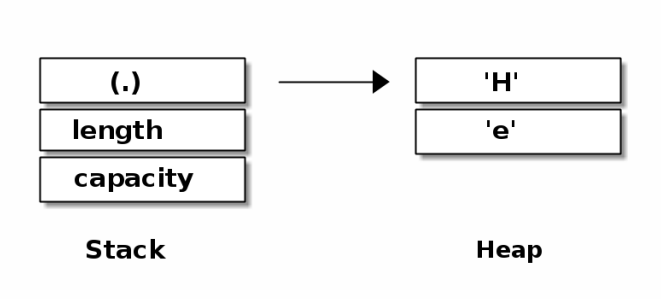

Figure 5.1. Memory layout of a Rust vector

In contrast, the memory safe language *Java* enforces a uniform internal representation of data. In Java a vector has 2 indirections instead of 1 compared to *Rust* and *C/C++* (cf. Figure 5.2, “Memory layout of a Java vector”). As the data could be represented in a more efficient way in memory, we see that *Java* does not prioritise the *Zero-Cost-Abstraction* goal.

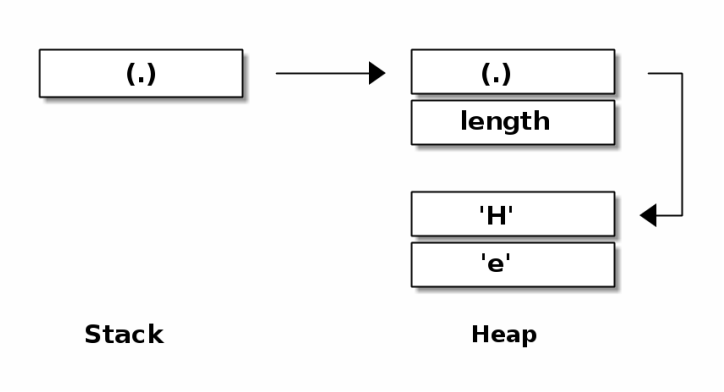

Figure 5.2. Memory layout of a Java vector

5.4. Recommendations for novice Rust programmers

This chapter introduces two fields of Rust programming that I struggled with at the beginning. Even when the code does not explicitly annotate lifetimes and does not use dynamic dispatching, the underlying concepts are vital for the understanding of Rust's error messages.

5.4.1. Borrow scope extension

My recommendation for novice programmers is to take the time to understand Rust's confusing concept of *lifetimes* in detail before starting a bigger project. In some cases the borrow-scope is not obvious to see. For example, a second borrower can extend the initial borrow scope. Liao [14] calls the phenomena "borrow scope extension".

Borrow cope extension, source code

```
struct Foo {
    f: Box<isize>,
}

fn main() {
    let mut a = Foo { f: Box::new(0) };
    let y: &Foo;
    if false {
        // borrow
        let x = &a; // share the borrow with new borrower y,
                    // hence extend the borrow scope
        y = x;
    }
    // error: cannot assign to `a.f` because it is borrowed
    a.f = Box::new(1);
}
```

The following error message only shows the initial borrower whose scope ends in line 13. The actual problem is caused by line 12 `y=x` which extends the initial borrow scope.

Borrow scope extension, error message

```
error[E0506]: cannot assign to `a.f` because it is borrowed
```

```
  --> <anon>:15:5
   |
10 |         let x = &a;
   |                  - borrow of `a.f` occurs here
...
15 |     a.f = Box::new(1);
   |     ^^^^^^^^^^^^^^^^^ assignment to borrowed `a.f` occurs here
```

To reason about borrows and lifetimes Liao [14] introduces the following lifetime scheme, which I find very useful in general. The brackets and variable names refer to the above source code.

Borrow scope extension, lifetime scheme

```
             { a { x y } * }
  resource a   |___________|
  borrower x       |___|     x = &a
  borrower y         |_____| y = x
borrow scope       |=======|
  mutate a.f             |   error
```

5.4.2. Structure as a borrower

Stringext's two main structures `Mission` and `Finding` are extensively borrowed throuout the source code. When a structure holds a reference the type-system has to make sure that the object it points to lives at least as as long as the structure itself. The following source code shows an example.

structure as a borrower, source code

```
struct Foo {
    f: Box<usize>,
}

struct Link<'a> {
    link: &'a Foo,
}

fn main() {
    let a = Foo { f: Box::new( 0 )};
    let mut x = Link { link: &a };
    if false {
        let b = Foo { f: Box::new( 1 )};
        x.link = &b; //error: `b` does not live long enough
```

```
    }
}
```

Structure as a borrower, error message

```
error: `b` does not live long enough
  --> src/main.rs:16:19
   |
16 |         x.link = &b;
   |                   ^ does not live long enough
17 |     }
   |     - borrowed value only lives until here
18 | }
   | - borrowed value needs to live until here
```

In the above example, the borrower `x` is borrowing `a`. The borrow scope ends at the end of the main block. The commented line `x.link = &b;` tries to borrow `b` instead and fails, because `b` must live at least as long as `x`! The following lifetime scheme illustrates the lifetime dependencies.

Structure as a borrower, lifetime scheme

```
                { a x { b * } }
resource a        |___________|
resource b              |___|
borrower x          |_________| x.link = &a
borrower x              |_|     x.link = &b
ERROR!
borrow scope x      |=========|
b should live at least  ¦.....¦
```

Chapter 6. Software development process and testing

The nature and appropriateness of the software development process impinges on the quality of the resulting software product. For the development of *Stringsext* the *test driven development* methodology was used. This chapter describes the reasons for this decision and reports on the experience.

6.1. Risk management

Based on the functional requirements described in the Chapter 4, *Specifications* and especially in the Section 4.3, “Concurrent scanning”, the Section 4.4, “Batch processing” and the Section 4.5, “Merge findings” the algorithm of the data-flow was defined (cf. Section 7.3, “Scanner Algorithm”).

Once a todo-list was established, *Stringsext’s* core functions were identified and ordered by risk: What would be the impact on the whole project if the implementation of a certain function turns out to be impossible or difficult to realise in the Rust?

Specifically, sorted after risk:

1. Section 7.5, “Integration with a decoder library”,
2. Section 7.1, “Concurrency”,
3. Section 7.7, “Polymorphic IO”,
4. Section 7.8, “Merging vectors”,
5. Section 7.6, “Valid string to graphical string filter”.

For every core function alternative technical solutions were suggested, implemented, tested and evaluated (cf. Chapter 7, *Analysis and Design*). This approach allowed at an early assurance that Rust provides abstractions and solutions for each of the core functions. It needs to be emphasized that the above isolated partial solutions do not reveal any indications about their intercompatibility or temporal behaviour. This risk is addressed in the next step.

6.2. Prototype

Regarding the above core functions the encoder library presented the highest risk. Was its low level interface suitable for the intended purpose? Was it fast enough? In order to answer these questions a first prototype with very little functionality was built. The first prototype showed as proof of concept, that the library meets the expectations.

6.3. Test Driven Development

From this point on, the actual development of *Stringsext* was launched. To meet the high demands in reliability and correctness it had been developed using the *Test Driven Development* (TDD) method suggested by Beck [15].

6.3.1. Writing tests

In conventional software development models, test are written after the design and implementation phase. In *Test Driven Development* this order is inverted: every new feature begins with writing a unit test or modifying an existing one.

Unit tests are isolation tests. They verify one piece of functionality only and have no dependencies on other test or on the order the tests are executed. They should not rely on external components such as data from filesystems, pipes, networks or databases. These external components have to be simulated by the test-code. The setting-up of the test environment code is often referred as *test fixture*. A *test case* is a set of input data and parameters for the to-be-tested code or function. Once the to-be-tested code is executed, the result is compared with the expected result. The expected result must be included in the test case and their relationship should be as apparent as possible [15 p. 130].

Rust has an integrated advanced support for unit testing. Here an example [16]:

Rust's Unit-Test feature

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}
```

```
#[test]
fn it_works() {
    assert_eq!(4, add_two(2));
}
```

The test-code in the Rust's Unit-Test feature is labelled with the compiler directive `#[test]`. This code is compiled only when Rust's compiler is invoked with `cargo test`.

The test-function `it_works()` calls the to-be-tested code function `add_two()` with the test case `2`. The assertion macro `assert_eq!()` compares the expected result `4` with the result of the to-be-tested code and breaks the test-run in case of failure.

6.3.2. Development cycle

Beck [15 p. 9] defines the development cycle is as follows:

1. Add a little test.

   "Write a little test that doesn't work, and perhaps doesn't even compile at first."

2. Run all tests and fail (*Red*-state).
3. Make a little change.

   "Make the test work quickly, committing whatever sins necessary in the process." Do not write code that the test does not check.

4. Run the tests and succeed (*Green*-state).
5. Refactor to remove duplication (*Refactored*-state).
6. Commit in your versioning system [1] .

Beck [15 p. 11] gives the following guidelines on how to execute the above steps:

> 1. Write a test. Think about how you would like the operation in your mind to appear in your code. You are writing a sto-

[1] The commit stage is not part of the original process, but added here for completeness.

ry. Invent the interface you wish you had. Include all the elements in the story that you imagine will be necessary to calculate the right answers.

2. Make it run. Quickly getting that bar to go to green dominates everything else. If a clean, simple solution is obvious, then type it in. If the clean, simple solution is obvious but it will take you a minute, then make a note of it and get back to the main problem, which is getting the bar green in seconds. This shift in aesthetics is hard for some experienced software engineers. They only know how to follow the rules of good engineering. Quick green excuses all sins. But only for a moment.
3. Make it right. Now the system is behaving, put the sinful ways of the recent past behind you. Step back onto the straight and narrow path of software righteousness. Remove the duplication that you have introduced and get to green [sic: should be refactored] quickly.

> The goal is clean code that works […]. First we'll solve the "that works" part of the problem. Then we'll solve the "clean code" part. This is the opposite of architecture-driven development, where you solve "clean code" first, then scramble around trying to integrate into the design the things you learn as you solve the "that works" problem.
>
> — The general Test Driven Development cycle *K. Beck*

6.3.3. Evaluation and conclusion

It was my first programming experience with *Test Driven Development* and it took me some time to develop the discipline to always start coding by writing a test and always observing the 6 stages of the development cycle. With the new system launched, it soon became clear what level of efficiency gains could be achieved. Concerning the *Test Driven Development Cycle* I observed the following:

From Red-state to Green state

Making the test fail first and check weather it succeeds after changing the code, validates the test itself! It proves not only that the new code

implements the new feature correctly, *it also proves that the test is observing the right functionality.*

From Green-state to Refactored state

At the beginning I was very sceptical about the “Make the test work quickly, committing whatever sins necessary in the process” suggestion. Wouldn’t it be more economic to write clean code right away? After some experience I fully agree with the above approach: Very often the solution is found only after several attempts. Maybe I need a library function that does not work the way I expected? Maybe I need a language construct I use for the first time? Finding a solution is creative process, that will work the best when the programmer sets himself (temporarily) free from coding conventions. Furthermore, the trial and error process is more time-economic when refactoring occurs only at the end of the process. Separating the “solution finding” (go into green) from “making it right and beautiful” (refactoring) helps to focus on what’s essential.

Critics of this method argue, that the development process is not structured enough and the project manager has very little control over it. It surely depends on the project, but in the present setting the balance between forward planning and creative freedom was just right. Moreover, the structuring effect of writing tests in the first place should not be underestimated. In order to design a test the programmer must necessarily address the functional requirements before writing the production code itself. Writing tests also supports code documentation: the testing code shows in an isolated environment how to interface with the to-be-tested code. This is very helpful when you need to understand someone else’s code, especially in case of a more complex low level API. Reading the testing code together with the to be tested code gives you an idea about the minimum environment a piece of code requires. This way the testing code supports and completes the *Rustdoc* in-code-documentation.

6.4. Documentation

Documentation is an important part of any software project. Rust projects and APIs are documented by annotating the source code with special comment-tags `\\\` and `\\!`. Annotations are usually placed just before the line it refers to. Documentation comments are written in Markdown. The Rust distribution includes a tool, *Rustdoc*, that generates the documentation.

Rustdoc's consists of linked html pages, similar to *Javadoc*. The *Stringsext* project makes extensive use of *Rust*'s documentation feature.

The user manual is written in *reStructuredText* format and compiled to a man-page with *Docutils* (cf. Table 8.3, “Manual page - stringsext - version 1.0”).

Chapter 7. Analysis and Design

This chapter discusses technical solutions complying with the specifications defined in the Chapter 4, *Specifications* and their implementation in Rust.

7.1. Concurrency

The Figure 7.1, “Data processing and threads” shows the data flow in *Stringsext*. All *scanner* instances as well as the *merger-printer* are designed as threads. Rust uses OS-level threads and its type and ownership model guarantees the absence of data races, which are defined as:

- two or more threads in a single process access the same memory location concurrently, and
- at least one of the accesses is for writing, and
- the threads are not using any exclusive locks to control their accesses to that memory.

Rust supports by default two models of inter-thread communication:

- shared memory [1] and
- message channels.

> To communicate between different concurrent parts of the codebase, there are two marker traits in the type system: “Send” and “Sync”. A type that is “Send” can be transferred between threads. A type that is “Sync” can be shared between threads.
>
> Thanks to the type and ownership system, Rust allows safe shared mutable state. In most programming languages, shared mutable state is the root of all evil. In Rust, the compiler enforces some rules that prevent data races from occurring.
>
> The alternative to shared memory are channels. A channel can be used like a Unix pipe. It has two ends, a sending and a re-

[1] Rust inherits C11’s memory model for atomics

ceiving end. Types that are “Send” can be sent through the pipe [17].

stringsext imports the crate “scoped_threadpool” used to distribute the *shared-memory* `input_slice` to its scanner-threads. Once a scanner has accomplished its mission, it sends its result through a dedicated *message channel* to the *merger-printer*-thread. The following code extract illustrates the implementation:

Inter-thread data exchange

```
pool.scoped(|scope| { ❶
    for mission in missions.iter_mut() {  ❷
        let tx = tx.clone();  ❸
        scope.execute(move || {  ❹
            let m = Scanner::scan_window (
                            &mut mission.offset, ❺❻
                            mission.encoding,
                            mission.filter_control_chars,
                            byte_counter,
                            input_slice ); ❼

             mission.offset = if mission.offset >= WIN_STEP { ❽
                 mission.offset - WIN_STEP
             } else {
                 0
             };

            match tx.send(m) { ❾
                 Ok(_) => {},
                 Err(_) => { panic!("Can not send FindingCollection:"); },
             };
        });
    }
});
```

❶ Pool with sleeping threads ready to receive a mission.

❷ Every thread has a context `mission` with its own variables it can read and write.

❸ Every thread gets a dedicated result-sending-channel.

❹ In a `scoped_treadpool` every thread has read access its parent’s stack.

❺ Note that `Scanner::scan_window()` is stateless!

❻ Its output is: `m`, the result as `FindingCollection` and `mission.offset` which is pointing to the byte where the scanner has stopped.

❼ The parent's stack access allows threads to read the `input_slice` concurrently.

❽ Prepare `mission.offset` for the next iteration: Update `mission.offset` to indicate the position where the next iteration should resume the work.

❾ Send the result to the *merger-printer*.

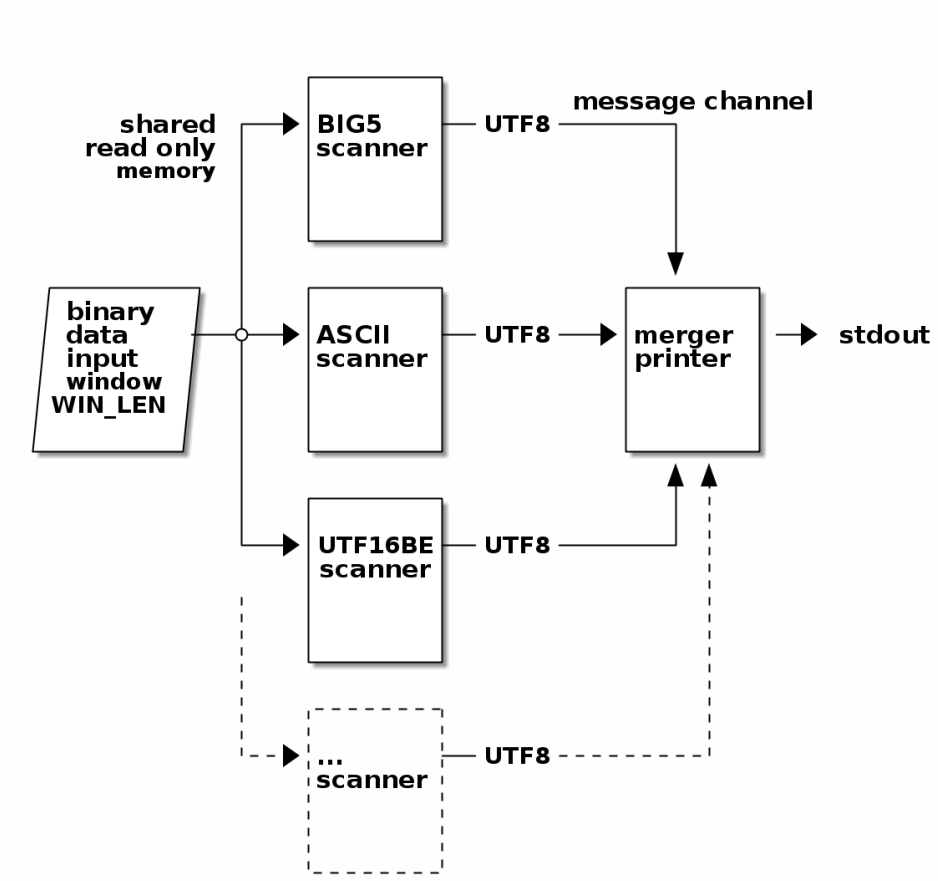

Figure 7.1. Data processing and threads

7.2. Reproducible output

Concurrent computing gives no guarantee in which order partial-output is available. With regard to *Stringsext* this means that the order in which the different scanners report their results, depend on racing conditions that are not predictable. In order to illustrate this phenomenon, the Figure 7.2, "Non reproducible output" shows the merged output of *Stringsext*. It is not

surprising that for example at position `47` and `c47` the scanners *ASCII* and *UTF-8* find strings at the same location since the ASCII-encoding is a subset of the UTF-8 encoding. Whilst at position `47` the output of the UTF-8 scanner is first listed, at position `c47` the output of the ASCII-scanner is printed first! Even though the output of *Stringsext* is always correct, the order in which the results are presented changes unpredictably!

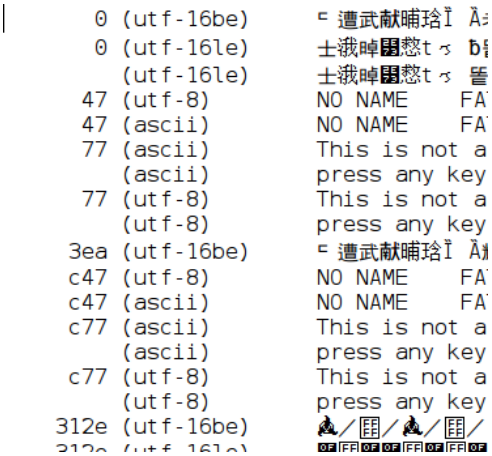

Figure 7.2. Non reproducible output

An important requirement for forensic tools is *reproducibility* meaning that the same input-data always produces bit-identical output-data.

The retained solution is to extend the sort by criteria which is used by the merger thread to order the findings. Now it proceeds as follows: first it sorts by the offset of the finding and then by the encoder name that has reported the finding. The Figure 7.3, “Reproducible output” shows the result. Please note that for all identical positions the ASCII-scanner result is always listed first.

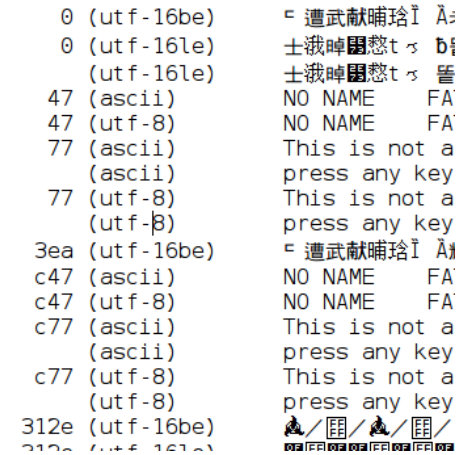

Figure 7.3. Reproducible output

7.3. Scanner Algorithm

The input data is processed in batches chunk by chunk. Each chunk is browsed in parallel by several “scanner” threads. This section describes the algorithm:

1. A *scanner* is a thread with an individual search `Mission` defined by the *encoding* it searches for.
2. The input data is divided into consecutive overlapping memory chunks. A chunk is a couple of 4KB memory pages, `WIN_LEN` bytes in size.
3. Scanners wait in pause state until they receive a pointer to a memory chunk with a dedicated search `Mission`.
4. All scanner-threads search simultaneously in one memory chunk only. This avoids that the threads drift to far apart.
5. Every scanner thread searches its encoding consecutively byte by byte from lower to higher memory.
6. When a scanner finds a valid string, it encodes it into a UTF-8 copy, called hereafter “finding”. Valid strings are composed of control characters and graphical characters.
7. Before storing a finding in `Finding` object, the above valid string is split into one or several graphical strings. Hereby all control characters are omitted. The graphical strings are then concatenated and the result is stored in a `Finding` object. A `Finding`-object also carries the memory location (offset) of the finding and a pointer describing the search mission. Goto 5.
8. A scanner stops when it passes the upper border `WIN_STEP` of the current memory chunk.
9. The scanner stores its `Finding`-objects in a vector referred as `Findings`. The vector is ascending in memory location.
10. Every scanner sends its `Findings` to the *merger-printer-thread*. In order to resume later, it updates a marker in its `Mission`-object pointing to the exact byte where it has stopped scanning. Besides this marker, the scanner is *stateless*. Finally, the scanner pauses and waits for the next memory chunk and mission.
11. After all scanners have finished their search in the current chunk, the *merger-printer-thread* receives the `Findings` and collects them in a vector.

12. The *merger-printer-thread* merges all `Findings` from all threads into one timeline and prints the formatted result through the output channel.
13. In order to prepare the next iteration, pointers in the `Mission`-objects are set to beginning of the next chunk. Every scanner resumes exactly where it stopped before.
14. Goto 3.
15. Repeat until the last chunk is reached.

7.4. Memory layout

The above algorithm splits the search field into overlapping memory chunks called `WIN_LEN`. Every chunk is also split into 3 fields: `WIN_STEP`, `FINISH_STR_BUF` and `UTF8_LEN_MAX`. This section explains how the algorithm operates on these fields.

`WIN_LEN` is the length of the memory chunk in which strings are searched in parallel.

Memory map

```
|<WIN_STEP1 -------------->|<WIN_STEP2 --------------->|<WIN_STEP3 -----
                           |<WIN_OVERLAP1>|            |<WIN_OVERLAP2>|
|<WIN_LEN1  --------------------------- >|
                           |<WIN_LEN2 ------------------------------>|
```

As shown above, `WIN_LEN` defines an overlapping window that advances `WIN_STEP` bytes each iteration.

`WIN_LEN = WIN_STEP + WIN_OVERLAP` is the size of the memory chunk that is processed during one iteration. A string is only found when it starts within the `WIN_STEP` interval. The remaining bytes can reach into `WIN_OVERLAP` or even beyond `WIN_LEN`. In the latter case the string is split.

Constant definition in source code

```
    pub const WIN_LEN: usize = WIN_STEP + WIN_OVERLAP;
```

`WIN_OVERLAP` is the overlapping fragment of the window. The overlapping fragment is used to read some bytes ahead when the string is not finished. `WIN_OVERLAP` is subject to certain conditions: For example the

overlapping part must be smaller than `WIN_STEP`. Furthermore, the size of `FINISH_STR_BUF = WIN_OVERLAP - UTF8_LEN_MAX` determines the number of bytes at the beginning of a string that are guaranteed not to be spit.

This size matters because the scanner counts the length of its findings. If a string is too short (< `ARG.flag_bytes`), it will be skipped. To avoid that a string with the required size gets too short because of splitting, we claim the following condition:

Constraint

```
1 <=  FLAG_BYTES_MAX <=  FINISH_STR_BUF
```

In practice we chose for `FINISH_STR_BUF` a bigger size than the minimum to avoid splitting of strings as much as possible. Please refer to the test function `test_constants()` for more details about constraints on constants. The test checks all the necessary conditions on constants to guarantee the correct functioning of the program.

Constant definition in source code

```
    pub const FINISH_STR_BUF: usize = 0x1800;
```

The scanner tries to read strings in `WIN_LEN` as far as it can. The first invalid byte indicates the end of a string and the scanner holds for a moment to store its finding. Then it starts searching further until the next string is found. Once `WIN_OVERLAP` is entered the search ends and the `start` variable is updated so that it now points to *restart-at-invalid* as shown in the next figure. This way the next iteration can continue at the same place the previous had stopped.

The next iteration can identify this situation because the `start` pointer points into the previous `FINISH_STR_BUF` interval.

Memory map

```
|<WIN_STEP1 ------------------------->|<FINISH_STR_BUF>|<UTF8_LEN_MAX>|
                                      |<WIN_OVERLAP1>---------------->|
|<WIN_LEN1 ------------------------------------------------------------>|

   <==string==><invalid bytes><=====string===><invalid bytes>
                                              ^
```

```
                                        |
                                `restart-at-invalid`
```

A special treatment is required when a sting extends slightly beyond `WIN_LEN`. In this case the scanner most likely runs into an incomplete multi-byte character just before the end of `WIN_LEN`. The cut surface *restart-at-cut* is then somewhere in the `UTF8_LEN_MAX` interval as the following figure shows.

The remaining part will be printed later during the next iteration. But how does the following iteration know if a string had been cut by the previous iteration? In the next interval the scanner first checks if the previous scan ended in the `UTF8_LEN_MAX` interval. If yes, we know the string has been cut and we the remaining bytes at the beginning of the new interval regardless of their size.

Memory map

```
<...---- WIN_STEP1 ------------>|<FINISH_STR_BUF>|<UTF8_LEN_MAX>|
                                |<WIN_OVERLAP1>---------------->|
<...---- WIN_LEN1 ------------------------------------------->|

   <==string==><invalid bytes><====string=============|==========...>
                                                     ^ incomplete
                                                     | valid Multi-
                                                     | byte-Char
                                                     |
                                                  `restart-at-cut`
```

To satisfy all the above constraints `WIN_OVERLAP` must satisfy two conditions concurrently:

Constraint

```
                                  WIN_OVERLAP <= WIN_STEP
FINISH_STR_BUF  + UTF8_LEN_MAX  =  WIN_OVERLAP
```

Constant definition in source code

```
    pub const WIN_OVERLAP: usize = FINISH_STR_BUF + UTF8_LEN_MAX as usize;
```

As Files are accessed through 4KiB memory pages, we choose `WIN_STEP` to be a multiple of 4096 bytes.

Constant definition in source code

```
    pub const WIN_STEP: usize = 0x2000; // = 2*4096
```

The `from_stdin()` function implements its own reader buffer `BUF_LEN` to allow stepping with overlapping windows. The algorithm requires that `BUF_LEN` is greater or equal than `WIN_LEN` (the greater the better the performance).

Constraint

```
    WIN_LEN <= BUF_LEN
```

Every time `BUF_LEN` is full, the last `WIN_OVERLAP` part must be copied from the end to the beginning of `BUF_LEN`. As copying is an expensive operation we choose:

Constraint

```
    BUF_LEN = 4 * WIN_STEP + WIN_OVERLAP
```

The above reduces the copying to every 4th iteration.

Constant definition in source code

```
pub const BUF_LEN: usize = 4 * WIN_STEP + WIN_OVERLAP;
```

In Unicode the maximum number of bytes a multi-byte-character can occupy in memory is 6 bytes.

Constant definition in source code

```
    pub const UTF8_LEN_MAX: u8 = 6;
```

7.5. Integration with a decoder library

To meet the requirements defined in the Section 4.2, “Character encoding support” *Stringsext*'s scanners perform a code conversion of their findings towards UTF-8 (see also Figure 7.1, “Data processing and threads”). Encoding conversion is a very complex matter: the Unicode specification alone has 1036 pages [18]! And Unicode is not the only encoding involved in *Stringsext*'s data processing.

It will come as no surprise that encoding conversion is related to numerous vulnerabilities (see the Section 4.10, “Secure coding” for details).

Basically, there are two ways to interface a third party library in Rust:

1. Writing bindings for a C library using the Foreign Function Interface [16].
2. Using a native Rust library.

In order to address potential security issues discussed in the Section 4.10, “Secure coding” the second option “native Rust library” had been chosen. *Stringsext* uses the so called *rust/encoding* library developed by Seonghoon [19].

rust/encoding provides encoder and decoder functionality for the following encodings specified by the WHATWG encoding standard:

- 7-bit strict ASCII (ascii)
- UTF-8 (utf-8)
- UTF-16 in little endian (utf-16 or utf-16le)
- UTF-16 in big endian (utf-16be)
- Single byte encodings in according to the WHATWG encoding standard:
  - IBM code page 866
  - ISO 8859-1 (distinct from Windows code page 1252)
  - ISO 8859-2, ISO 8859-3, ISO 8859-4, ISO 8859-5, ISO 8859-6, ISO 8859-7, ISO 8859-8, ISO 8859-10, ISO 8859-13, ISO 8859-14, ISO 8859-15, ISO 8859-16
  - KOI8-R, KOI8-U
  - MacRoman (macintosh), Macintosh Cyrillic encoding (x-mac-cyrillic)
  - Windows code pages 874, 1250, 1251, 1252 (instead of ISO 8859-1), 1253, 1254 (instead of ISO 8859-9), 1255, 1256, 1257, 1258
- Multi byte encodings according to the WHATWG Encoding standard:
  - Windows code page 949 (euc-kr, since the strict EUC-KR is hardly used)

  - EUC-JP and Windows code page 932 (shift_jis, since it's the most widespread extension to Shift_JIS)
  - ISO-2022-JP with asymmetric JIS X 0212 support (Note: this is not yet up to date to the current standard)
  - GBK
  - GB 18030
  - Big5-2003 with HKSCS-2008 extensions
- Encodings that were originally specified by WHATWG encoding standard:
  - HZ

7.6. Valid string to graphical string filter

The *rust/encoding* library was originally not designed to search for strings in binary data. Nevertheless, the low level API function `decoder.raw_feed()` returns and decodes chunks of *valid* strings found in the input stream. Those *valid strings* are then always re-encoded to UTF-8 and comprise:

- **Graphical characters** represent a written symbol. When printed, toner or ink can be seen on the paper.

> *GNU-strings* and *Stringsext* consider SPACE and TAB as graphical characters.

- **Control characters** have no visual or spatial representation. They control the interpretation or display of text.

As the *rust/encoding* library returns *valid* strings and *Stringsext* prints by default only *graphical* strings, and additional filter must be applied:

Control character filter

```
let len = $fc.v.last().unwrap().s.len();
let mut out = String::with_capacity(len); ❶
{
    let mut chunks = (&$fc).v.last().unwrap().s ❷
        .split_terminator(|c: char|
                          c.is_control()
```

```
                        && c != ' ' && c !='\t'
        )  ❸
        .enumerate()
        .filter(|&(n,s)| (s.len() >= minsize ) ||  ❹
                        ((n == 0) && $fc.completes_last_str) ❺
        )
        .map(|(_, s)| s );

    if let Some(first_chunk) = chunks.next() { ❻
        if !$fc.v.last().unwrap().s.starts_with(&first_chunk) { ❼
            out.push_str(&CONTROL_REPLACEMENT_STR); ❽
        }
        out.push_str(first_chunk);
        for chunk in chunks { ❾
            out.push_str(&CONTROL_REPLACEMENT_STR);
            out.push_str(chunk);
        }
    }
};
```

❶ `out` is the filtered string containing all concatenated graphical strings.

❷ `(&$fc).v.last().unwrap().s` is the *valid input* string comprising control and graphical characters.

❸ Iterator over chunks of graphical strings.

❹ Filter out too short strings,

❺ unless they do not complete a cut off string from the previous scanner run.

❻ Read the first graphical string.

❼ Had there been control characters before it?

❽ Place a `CONTROL_REPLACEMENT_STR` character. The actually inserted character depend on the `--control-chars` command-line option: For `--control-chars=r` the character `\u{fffd}` is inserted. For `--control-chars=i` the character `\n` (newline) is inserted.

❾ Concatenate all the remaining graphical strings and place a `CONTROL_REPLACEMENT_STR` in between each.

All scanners use the above filter unless the command-line option `--control-chars=p` is given. Then the whole valid string is printed with all its control characters.

The only exception to this occurs for the options `-e ascii -c i`. This combination invokes a specially designed ASCII-graphical-strings-only de-

coder. This approach made it possible to generate a bit identical output compared to *GNU-strings* for this setting.

The option `--control-chars=r` addresses especially the requirement defined in the Section 4.6, “Facilitate post-treatment”. All control characters are replaced by `\u{fffd}` keeping the filtered string always in one line. Together with the formatting option `-t` the printed lines have the following syntax:

```
<offset>'\t('<encoding name>')\t'<graphical string>'\u{fffd}'<graphical...
```

7.7. Polymorphic IO

GNU-strings can read its input from a file, or from a pipe. Both requires different optimisation strategies. The fastest way to read a file sequentially is through the *memory mapping* kernel interface. *danburkert/memmap-rs* is a Rust library for cross-platform memory-mapped file IO. An interesting feature of *memory mapping* is that it can map files much larger than the available RAM to a *virtual address space*. This allows to map the whole file regards to its size and iterate over it with a sliding window. The following code extract shows this technique. Note that there is only one call of the `Mmap::open()` wrapper occurring outside the loop. This reduces the overhead caused by the wrapper.

Memory mapping the entire file

```
let mut byte_counter: usize = 0;
let file = try!(Mmap::open(file, Protection::Read)); ❶
let bytes = unsafe { file.as_slice() };
let len = bytes.len();
for chunk in bytes.windows(WIN_LEN).step(WIN_STEP) {
    sc.launch_scanner(&byte_counter, &chunk); ❷
    byte_counter += WIN_STEP;
}
```

❶ Map the whole file contents in virtual address space.

❷ Launch the scanner threads providing a chunk of memory.

An alternative technique consists mapping memory pages sequentially page by page. The following code shows this approach. Note that the call of `Mmap::open_with_offset()` happens inside the loop!

Memory mapping page by page

```
let len = try!(file.metadata()).len() as usize;
let mut byte_counter: usize = 0;
while byte_counter + WIN_LEN <= len {
    let mmap = Mmap::open_with_offset(&file, Protection::Read,  ❶
                                      byte_counter,WIN_LEN).unwrap();
    let chunk = unsafe { mmap.as_slice() };
    sc.launch_scanner(&byte_counter, &chunk);  ❷
    byte_counter += WIN_STEP;
}
```

❶ Map a few numbers of memory pages only.

❷ Pass them to the scanner threads.

Tests with big files showed that memory mapping page by page is faster despite its overhead. One reason is that the other solution does not launch the scanner right from the start: the operating system reads the whole data in memory before giving the control back to the calling program. This does not only consume a lot of memory but also holds back the scanners when the program starts. This is why the solution *Memory mapping page by page* was selected: it reads only very few memory pages into memory and the scanner can start their work much earlier.

Note that the function `as_slice()` is tagged “unsafe”. It means that the file-reading operation is only safe as long as no other process writes that file simultaneously. I consider this requirement to be met for all use cases of *Stringsext* and we do not implement any additional file locking mechanism.

The second operation mode “reading input from a pipe” raised another challenge: None of the standard input readers is able to read by overlapping chunks. To solve the problem a circular-buffer was implemented. The following shows an extract of the source code.

Circular input buffer

```
while !done {
    // Rotate the buffer if there isn't enough space ❶
    if data_start + WIN_LEN > BUF_LEN {
        let (a, b) = buf.split_at_mut(data_start);
        let len = data_end - data_start;
        a[..len].copy_from_slice(&b[..len]);
        data_start = 0;
```

```
        data_end = len;
    }
    // Read from stdin
    while data_end < data_start + WIN_LEN {          ❷
        let bytes = try!(stdin.read(&mut buf[data_end..]));
        if bytes == 0 {
            done = true;
            break;
        }
        else {data_end += bytes; }
    }
    // Handle data.                                  ❸
    while data_start + WIN_LEN <= data_end {
        sc.launch_scanner(&byte_counter,
                &buf[data_start..data_start + WIN_LEN]);
        data_start += WIN_STEP;
        byte_counter += WIN_STEP;
    }
}
```

❶ Make sure that there is always enough space to receive the next input chunk.

❷ Fill the buffer from `stdin`.

❸ Empty the buffer by reading `WIN_LEN` bytes.

7.8. Merging vectors

The *merger-printer* thread in the Figure 7.1, “Data processing and threads” receives vectors of `Findings` from the connected upstream scanners. Every input vector is sorted by memory offset.

To merge the input vectors two alternative solutions have been developed.

The first solution is based on a contribution of Jake Goulding, aka Shepmaster, who posted the following code realising an iterator able to merge 2 vectors [20].

Merging iterator for two vectors

```
use std::iter::Peekable;
use std::cmp::Ordering;

struct MergeAscending<L, R>
    where L: Iterator<Item = R::Item>, R: Iterator,
```

```
{
    left: Peekable<L>,
    right: Peekable<R>,
}

impl<L, R> MergeAscending<L, R>
    where L: Iterator<Item = R::Item>, R: Iterator,
{
    fn new(left: L, right: R) -> Self {
        MergeAscending {
            left: left.peekable(),
            right: right.peekable(),
        }
    }
}

impl<L, R> Iterator for MergeAscending<L, R>
    where L: Iterator<Item = R::Item>, R: Iterator, L::Item: Ord,
{
    type Item = L::Item;
    fn next(&mut self) -> Option<L::Item> {
        let which = match (self.left.peek(), self.right.peek()) {
            (Some(l), Some(r)) => Some(l.cmp(r)),
            (Some(_), None)    => Some(Ordering::Less),
            (None, Some(_))    => Some(Ordering::Greater),
            (None, None)       => None,
        };
        match which {
            Some(Ordering::Less)    => self.left.next(),
            Some(Ordering::Equal)   => self.left.next(),
            Some(Ordering::Greater) => self.right.next(),
            None                    => None,
        }
    }
}
```

The following testing code illustrates how to merge two vectors.

Testing code merging iterator for two vectors

```
#[test]
fn merge_two_iterators_concrete_types() {
    let left  = [1, 3, 5, 7, 9];
    let right = [3, 4, 5, 6];
    let result: Vec<_> =
        MergeAscending::new(left.iter(), right.iter()).collect();
```

```
    let expected = vec![1, 3, 3, 4, 5, 5, 6, 7, 9];
    // result == expected?
    assert!( result.iter().zip(expected).all(|(&a,b)| a-b == 0 )
    );
}
```

Jake Goulding's code, as shown above, can only merge two vectors. The following macro, realised by the author of this present work, extends the above code by adding successively iterators. The resulting algorithm to find the next element is basically a linear search. Its complexity is `O(N * k)`, where `N` is the total length of iterables and `k` is the number of iterables.

Merging iterator for multiple vectors

```
macro_rules! merging_iterator_from {
    ($vv: ident) => {{
        let mut ma: Box<Iterator<Item=_>> =
                    Box::new($vv[0].iter().map(|&i|i));
        for v in $vv.iter().skip(1) {
            ma = Box::new(MergeAscending::new(ma, v.iter().map(|&i|i)));
        };
        ma
    }}
}
```

The following testing code illustrates how to merge 5 vectors.

Testing code merging iterator for multiple vectors

```
#[test]
fn merge_five_iterators() {
    let vv: Vec<Vec<_>> = vec![
        vec![1, 3, 5, 7, 9],
        vec![3, 4, 6, 7],
        vec![0, 6, 8],
        vec![1, 2, 12],
        vec![10]
    ];

    let result: Vec<_> = merging_iterator_from!(vv).collect::<Vec<_>>();

    let expected = vec![0, 1, 1, 2, 3, 3, 4, 5, 6, 6, 7, 7, 8, 9, 10, 12];
    // result == expected?
    assert!( result.iter().zip(expected).all(|(&a,b)| a-b == 0 )
```

```
    );
}
```

For test purposes, the above 4 code samples can be concatenated in one file.

The benefit of this solution is its simplicity and that it does not require any external library. Sure, linear search is not the fastest algorithm, but seeing the little number of vectors we have to merge this is not necessarily a drawback.

Shortly after I implemented the above solution, the iterator `kmerge` was published in the *rust/itertools* library. It implements the heapsort algorithm. The complexity of the approach is `O(N * log(k))`, where `N` is the total length of iterables and `k` is the number of iterables. Its better performance, compared to the first solution, is practically negligible in the present case as number of iterables is relatively small.

The next testing code sample shows how to merge 3 vectors.

Testing code kmerge library

```
extern crate itertools;
use itertools::free::kmerge;

#[test]
fn merge_three_iterators() {
    let vv = vec![
        vec![0, 2, 4],
        vec![1, 2, 5],
        vec![3, 7]
    ];

    let result = kmerge(&vv).collect::<Vec<_>>();

    let expected = vec![0, 1, 2, 2, 3, 4, 5, 7];

    // result == expected?
    assert_eq!(result.len(),expected.len());
    assert!( result.iter().zip(expected).all(|(&a,b)| a-b == 0 )
    );
}
```

For *Stringsext* I finally chose the second solution with `kmerge`. Its slightly better performance is surely desirable, but most of all it was my intention of

keeping *Stringsext*'s code base as small as possible that led to the decision using the external `kmerge`-function of the *rust/itertools* library.

Chapter 8. Stringsext’s usage and product evaluation

The initial motivation for developing *Stringsext* were the various shortcomings of *GNU-strings* especially when it comes to handle international character encodings. Does *Stringsext* support foreign scripts better? Is it as fast?

8.1. Test case 2 - international character encodings

To evaluate *Stringsext*'s capabilities to handle international scripts with Unicode we chose the same text file as input we used with *GNU-strings* in the Section 3.1, “Test case 1 - International character encodings”:

```
Arabic: A lie has short legs. (Lit: The rope of lying is short.)
حبل الكذب قصير

Chinese: Teachers open the door. You enter by yourself.
師傅領進門，修行在個人

French: pasta
Les pâtes

Greek: History
Ιστορία

German: Greetings
Viele Grüße

Russian: Congratulations
Поздравляю

Euro sign
€ (U+20AC)

Treble clef
𝄞 (U+1D11E)
```

Figure 8.1. Unicode test-file: orig.txt

The following bash-script automates the test case generation: To provide a copy of the test file in UTF-8, UTF-16be, UTF-16le, UTF-32be and UTF-32le encodings the Unix tool `iconv` is used.

The second part of the script feeds the generated copies one by one into *Stringsext*. The options `-e ascii -e utf-8 -e utf-16be -e utf-16le`

instruct *Stringsext* to search for the following encodings: ASCII, UTF-8, UTF-16be, UTF-16le. Please refer to Table 8.3, “Manual page - stringsext - version 1.0” for details on *stingsext*'s command-line options.

Encoding test script

```
#!/bin/sh
cp orig.txt encoded-utf8.txt
iconv -f utf8 -t utf16le orig.txt >encoded-utf16le.txt
iconv -f utf8 -t utf32le orig.txt >encoded-utf32le.txt
iconv -f utf8 -t utf16be orig.txt >encoded-utf16be.txt
iconv -f utf8 -t utf32be orig.txt >encoded-utf32be.txt

echo "Test stringsext" > report.txt

find . -name "encoded*" -exec echo -e "\n\nScanning file {}:\n" \; \
  -exec ./stringsext -n 8 -e ascii -e utf-8 -e utf-16be -e utf-16le \
                     -c i -t x {} \;  >> report.txt
```

The following figures show *stringsext*'s output, case by case.

8.1.1. UTF-8 encoded input

Stringsext’s UTF-8 encoded input

```
0000000: efbb bf41 7261 6269 633a 2041 206c 6965  ...Arabic: A lie
0000010: 2068 6173 2073 686f 7274 206c 6567 732e   has short legs.
0000020: 2028 4c69 743a 2054 6865 2072 6f70 6520   (Lit: The rope
0000030: 6f66 206c 7969 6e67 2069 7320 7368 6f72  of lying is shor
0000040: 742e 290a d8ad d8a8 d984 20d8 a7d9 84d9  t.)....... .....
0000050: 83d8 b0d8 a820 d982 d8b5 d98a d8b1 200a  ..... ........ .
0000060: 0a43 6869 6e65 7365 3a20 5465 6163 6865  .Chinese: Teache
0000070: 7273 206f 7065 6e20 7468 6520 646f 6f72  rs open the door
0000080: 2e20 596f 7520 656e 7465 7220 6279 2079  . You enter by y
0000090: 6f75 7273 656c 662e 0ae5 b8ab e582 85e9  ourself.........
00000a0: a098 e980 b2e9 9680 efbc 8ce4 bfae e8a1  ................
00000b0: 8ce5 9ca8 e580 8be4 baba 0a0a 4672 656e  ............Fren
00000c0: 6368 3a20 7061 7374 610a 4c65 7320 70c3  ch: pasta.Les p.
00000d0: a274 6573 0a0a 4772 6565 6b3a 2048 6973  .tes..Greek: His
00000e0: 746f 7279 0ace 99cf 83cf 84ce bfcf 81ce  tory............
00000f0: afce b10a 0a47 6572 6d61 6e3a 2047 7265  .....German: Gre
0000100: 6574 696e 6773 0a56 6965 6c65 2047 72c3  etings.Viele Gr.
0000110: bcc3 9f65 0a0a 5275 7373 6961 6e3a 2043  ...e..Russian: C
0000120: 6f6e 6772 6174 756c 6174 696f 6e73 0ad0  ongratulations..
0000130: 9fd0 bed0 b7d0 b4d1 80d0 b0d0 b2d0 bbd1  ................
```

```
0000140: 8fd1 8e0a 0a45 7572 6f20 7369 676e 0ae2  .....Euro sign..
0000150: 82ac 2028 552b 3230 4143 290a 0a54 7265  .. (U+20AC)..Tre
0000160: 626c 6520 636c 6566 0af0 9d84 9e20 2028  ble clef.....  (
0000170: 552b 3144 3131 4529 0a0a 0a              U+1D11E)...
```

```
Scanning file ./encoded-utf8.txt:

      0 (utf-16be)
      0 (utf-16le)
      0 (utf-8)       Arabic: A lie has short legs. (Lit: The rope of l
        (utf-8)       حبل الكذب قصير
        (utf-8)       Chinese: Teachers open the door. You enter by you
        (utf-8)       師傅領進門，修行在個人
        (utf-8)       French: pasta
        (utf-8)       Les pâtes
        (utf-8)       Greek: History
        (utf-8)       Ιστορία
        (utf-8)       German: Greetings
        (utf-8)       Viele Grüße
        (utf-8)       Russian: Congratulations
        (utf-8)       Поздравляю
        (utf-8)       Euro sign
        (utf-8)       € (U+20AC)
        (utf-8)       Treble clef
        (utf-8)       𝄞 (U+1D11E)
      3 (ascii)       Arabic: A lie has short legs. (Lit: The rope of l
     4a (utf-16be)
     54 (utf-16le)
     5e (utf-16be)
     61 (ascii)       Chinese: Teachers open the door. You enter by you
     bc (ascii)       French: pasta
     d6 (ascii)       Greek: History
     f5 (ascii)       German: Greetings
    107 (ascii)       Viele Gr
    116 (ascii)       Russian: Congratulations
    145 (ascii)       Euro sign
    152 (ascii)         (U+20AC)
    15d (ascii)       Treble clef
    16d (ascii)         (U+1D11E)
```

Figure 8.2. Stringsext’s output with UTF-8 encoded input

Observations

The UTF-8-scanner recognize all characters correctly starting with offset `0x0`. Even though the input starts with word “Arabic”, the ASCII scanner identifies the first ASCII character with offset `0x3`! The reason is the preceding byte-Sequence `ef bb bf` which is a Unicode byte-order-mark (BOM, cf. Table 8.1, “Unicode byte order mark”) indicating the used encoding . For the UTF-8 scanner the BOM is a valid byte-sequence, for the ASCII scanner it is not. This is why the ASCII-scanner reports the position of the first valid byte at position `0x3`.

Table 8.1. Unicode byte order mark

BOM bytes	Encoding
EF BB BF	UTF-8
FE FF	UTF-16, big-endian

BOM bytes	Encoding
FF FE	UTF-16, little-endian
00 00 FE FF	UTF-32, big-endian
FF FE 00 00	UTF-32, little-endian

Knowing that the ASCII-encoding is a subset of UTF-8, we are not surprised that that most ASCII characters are recognized. But there are some exceptions to this rule. For example, we can see that the ASCII character “e” of the word “Grüße” at position `0x113` is not printed! It may initially be surprising, but we should keep in mind that for the ASCII-scanner the characters “üß” are invalid byte sequences. When the scanner encounters the letter “e” at the end of the line, it is discarded because one letter alone does not meet the minimum string length requirement.

The lines 34-35 and 53-55 are showing the strings found by the UTF-16BE and UTF-16LE scanners. Surprisingly these scanners found Chinese characters in our text! Because we designed the test case ourself, we know that *Stringsext*'s input data is definitely encoded in UTF-8 with very little Chinese symbols. This means any other encodings found in there are false positives!

8.1.2. UTF-16 encoded input

UTF-16 exists in two variants: UTF-16BE (big-endian) and UTF16LE (little-endian). The following figures show the sample in- and output for each of these variants.

Stringsext’s UTF-16be encoded input

```
0000000: feff 0041 0072 0061 0062 0069 0063 003a  ...A.r.a.b.i.c.:
0000010: 0020 0041 0020 006c 0069 0065 0020 0068  . .A. .l.i.e. .h
0000020: 0061 0073 0020 0073 0068 006f 0072 0074  .a.s. .s.h.o.r.t
...
00001b0: 0067 0073 000a 0056 0069 0065 006c 0065  .g.s...V.i.e.l.e
00001c0: 0020 0047 0072 00fc 00df 0065 000a 000a  . .G.r.....e....
00001d0: 0052 0075 0073 0073 0069 0061 006e 003a  .R.u.s.s.i.a.n.:
00001e0: 0020 0043 006f 006e 0067 0072 0061 0074  . .C.o.n.g.r.a.t
00001f0: 0075 006c 0061 0074 0069 006f 006e 0073  .u.l.a.t.i.o.n.s
0000200: 000a 041f 043e 0437 0434 0440 0430 0432  .....>.7.4.@.0.2
0000210: 043b 044f 044e 000a 000a 0045 0075 0072  .;.O.N.....E.u.r
0000220: 006f 0020 0073 0069 0067 006e 000a 20ac  .o. .s.i.g.n.. .
```

```
0000230: 0020 0028 0055 002b 0032 0030 0041 0043  . .(.U.+.2.0.A.C
0000240: 0029 000a 000a 0054 0072 0065 0062 006c  .).....T.r.e.b.l
0000250: 0065 0020 0063 006c 0065 0066 000a d834  .e. .c.l.e.f...4
0000260: dd1e 0020 0020 0028 0055 002b 0031 0044  ... . .(.U.+.1.D
0000270: 0031 0031 0045 0029 000a 000a 000a       .1.1.E.).......
```

```
Scanning file ./encoded-utf16be.txt:

     0 (utf-16be)     Arabic: A lie has short legs. (Lit: The rope of l
       (utf-16be)     حبل الكذب قصير
       (utf-16be)     Chinese: Teachers open the door. You enter by you
       (utf-16be)     師傅領進門，修行在個人
       (utf-16be)     French: pasta
       (utf-16be)     Les pâtes
       (utf-16be)     Greek: History
       (utf-16be)     Ιστορία
       (utf-16be)     German: Greetings
       (utf-16be)     Viele Grüße
       (utf-16be)     Russian: Congratulations
       (utf-16be)     Поздравляю
       (utf-16be)     Euro sign
       (utf-16be)     € (U+20AC)
       (utf-16be)     Treble clef
       (utf-16be)     𝄞 (U+1D11E)
     0 (utf-16le)     xEFxBFxBF
 1ca (utf-16le)
```

Figure 8.3. Stringsext’s output with UTF-16be encoded input

Stringsext’s UTF-16le encoded input

```
0000000: fffe 4100 7200 6100 6200 6900 6300 3a00  ..A.r.a.b.i.c.:.
0000010: 2000 4100 2000 6c00 6900 6500 2000 6800   .A. .l.i.e. .h.
0000020: 6100 7300 2000 7300 6800 6f00 7200 7400  a.s. .s.h.o.r.t.
...
00001b0: 6700 7300 0a00 5600 6900 6500 6c00 6500  g.s...V.i.e.l.e.
00001c0: 2000 4700 7200 fc00 df00 6500 0a00 0a00   .G.r.....e.....
00001d0: 5200 7500 7300 7300 6900 6100 6e00 3a00  R.u.s.s.i.a.n.:.
00001e0: 2000 4300 6f00 6e00 6700 7200 6100 7400   .C.o.n.g.r.a.t.
00001f0: 7500 6c00 6100 7400 6900 6f00 6e00 7300  u.l.a.t.i.o.n.s.
0000200: 0a00 1f04 3e04 3704 3404 4004 3004 3204  ....>.7.4.@.0.2.
0000210: 3b04 4f04 4e04 0a00 0a00 4500 7500 7200  ;.O.N.....E.u.r.
0000220: 6f00 2000 7300 6900 6700 6e00 0a00 ac20  o. .s.i.g.n.... 
0000230: 2000 2800 5500 2b00 3200 3000 4100 4300   .(.U.+.2.0.A.C.
0000240: 2900 0a00 0a00 5400 7200 6500 6200 6c00  ).....T.r.e.b.l.
0000250: 6500 2000 6300 6c00 6500 6600 0a00 34d8  e. .c.l.e.f...4.
0000260: 1edd 2000 2000 2800 5500 2b00 3100 4400  .. . .(.U.+.1.D.
0000270: 3100 3100 4500 2900 0a00 0a00 0a00       1.1.E.).......
```

```
10 Scanning file ./encoded-utf16le.txt:
11
12        0 (utf-16be)      xEFxBFxBE祓爀慬慴椀瀀牥 祓 敳椀攀 桴慬渀 渀桴敳爀瑬 敳
13        0 (utf-16le)      Arabic: A lie has short legs. (Lit: The rope of l
14          (utf-16le)      حبل الكذب قصير
15          (utf-16le)      Chinese: Teachers open the door. You enter by you
16          (utf-16le)      師傅領進門，修行在個人
17          (utf-16le)      French: pasta
18          (utf-16le)      Les pâtes
19          (utf-16le)      Greek: History
20          (utf-16le)      Ιστορία
21          (utf-16le)      German: Greetings
22          (utf-16le)      Viele Grüße
23          (utf-16le)      Russian: Congratulations
24          (utf-16le)      Поздравляю
25          (utf-16le)      Euro sign
26          (utf-16le)      € (U+20AC)
27          (utf-16le)      Treble clef
28          (utf-16le)      𝄞 (U+1D11E)
29      1ca (utf-16be)      攀▯▯刀攀渀渀椀慴湥搀 縀敳湥最爀慴瑬攀敳慴瑬椀敳湥渀▯α爀
```

Figure 8.4. Stringsext's output with UTF-16le encoded input

In the Figure 8.4, "Stringsext's output with UTF-16le encoded input" is interesting to notice, that the UTF-16BE scanner in line 29 restarts at offset `0xca`. The above hex-dump of *Stringsext*'s input data explains why: the preceding bytes `df00 6500` at position `0xc6` are one of the rare invalid code unit combinations in UTF-16BE (cf. Table 8.2, "UTF-16 Bit distribution"). The same phenomena can be observed in the Figure 8.3, "Stringsext's output with UTF-16be encoded input".

The Figure 8.3, "Stringsext's output with UTF-16be encoded input" and the Figure 8.4, "Stringsext's output with UTF-16le encoded input" show that, when the right decoder (big-endian or little-endian) is chosen, all Unicode-characters are recognized and printed correctly. This is huge improvement compared to *GNU-strings* which failed to recognize any non-ASCII characters in UTF-16 (cf. Section 3.1, "Test case 1 - International character encodings").

When the wrong scanner was chosen, we see Chinese and Japanese characters. These false positives are very common when scanning for UTF-16 characters. The reason is not the scanner, but an inherent property of the UTF-16 encoding: Almost every possible byte combination maps to a valid UTF-16 character! Only some very few byte sequences are invalid: "Because surrogate code points are not Unicode scalar values, isolated UTF-16 code units in the range `0xD800..0xDFFF` are ill-formed [18 p. 160]". Nevertheless, even code units in this invalid range can appear as *surrogate pairs* as shown in the last line of the following table:

Table 8.2. UTF-16 Bit distribution

Unicode scalar value (code point)	**UTF-16-BE code units**
`xxxxxxxx xxxxxxxx` (no code points in `110111000 00000000` … `110111111 11111111`)	`xxxxxxxx xxxxxxxx` (all except `110111000 00000000` … `110111111 11111111`)
`000uuuuu xxxxxxxx xxxxxxxx`	surrogate pairs: `110110ww wwxxxxxx 110111xx xxxxxxxx` (with `wwww = uuuuu - 1`)

As we can see from the Table 8.2, "UTF-16 Bit distribution", almost every possible byte sequence, interpreted as UTF-16 code unit, relates to a Unicode code point. 96% of the UTF-16 code units map directly to Unicode plane 0 (Basic Multilingual Plane BMP) code points. This explains the big number of false positives. But why do we see so many Chinese and Japanese characters (CJK)? The reason is simple: there are just so many of them in plane 0! The range `0x2E80-0x33FF` is allocated to the "CJK Miscellaneous Area", and the range `0x3400-0x9FFF` to the "CJKV Unified Ideographs Area" [18 p. 85] covering 29055 code units out of 63488 possible code units. This means the probability of encountering CJKV symboles in a random byte stream, interpreted as UTF-16, is 44%. In a stream with ASCII text this probability is even much higher and can get close to 100% because alphabetical letters in ASCII are encoded as `0x41`-`0x7a`. When these bytes are interpreted as high bytes of UTF-16 code units, the result always points in the CJKV Unicode range.

In the context of forensic examination, false positives are highly undesirable. A practicable solution could be to restrict the output of scanners by setting up additional filter criteria: for example the user could limit his search to a certain Unicode code block. This solution is out of the scope of this work and considered as future potential extension.

As of *Stringsext* version 1.1 [1], the `--encoding` option interprets specifiers limiting the search scope to a range of Unicode blocks.

[1] This present document describes *Stringsext* 1.0. The new Unicode-range-filter feature released with *Stringsext* version 1.1 was published after the writing of this thesis.

> For example `--encoding utf-16le,8,U+0..U+3ff` searches for strings encoded in UTF-16 Little Endian being at least 8 bytes long and containing only Unicode codepoints in the range from `U+0` to `U+3ff`. Please consult the man-page for details.

8.2. User documentation

The following table shows the *man-page* user documentation. It is typeset as *reStructuredText* and compiled using the *Sphinx* tool [21].

Table 8.3. Manual page - stringsext - version 1.0

NAME

stringsext - search for valid strings, decode and print its graphic characters as UTF-8.

stringsext is a Unicode enhancement of the *GNU strings* tool with additional functionalities: **stringsext** recognizes Cyrillic, CJKV characters and other scripts in all supported multi-byte-encodings, while *GNU strings* fails in finding any of these scripts in UTF-16 and many other encodings.

SYNOPSIS

```
stringsext [options] [-e ENC...] [--] [FILE]
stringsext [options] [-e ENC...] [--] [-]
```

DESCRIPTION

stringsext prints all graphic character sequences in *FILE* or *stdin* that are at least *MIN* bytes long.

Unlike *GNU strings* **stringsext** can be configured to search for valid characters not only in ASCII but also in many other input encodings, e.g.: utf-8, utf-16be, utf-16le, big5-2003, euc-jp, koi8-r and many others. **--list-encodings** shows a list of valid encoding names based on the WHATWG Encoding Standard. When more than one encoding is specified, the scan is performed in different threads simultaneously.

stringsext reads its input data from **FILE**. With no **FILE**, or when **FILE** is `-`, it reads standard input *stdin*.

stringsext is mainly useful for determining the Unicode content of non-text files.

When invoked with `stringsext -e ascii -c i` **stringsext** can be used as *GNU strings* replacement.

OPTIONS

-c *MODE*, --control-chars=*MODE*
Determine if and how control characters are printed.

The search algorithm first scans for valid character sequences which are then are re-encoded into UTF-8 strings containing graphical (printable) and control (non-printable) characters.

When *MODE* is set to **p** all valid (control and graphic) characters are printed. Warning: Control characters may contain a harmful payload. An attacker may exploit a vulnerability of your terminal or post processing software. Use with caution.

MODE **r** will never print any control character but instead indicate their position: Control characters in valid strings are first grouped and then replaced with the Unicode replacement character '  ' (U +FFFD). This mode is most useful together with **--radix** because it keeps the whole valid character sequence in one line allowing post-processing the output with line oriented tools like `grep`. To ease post-processing, the output in MODE **r** is formatted slightly different from other modes: instead of indenting the byte-counter, the encoding name and the found string with *spaces* as separator, only one *tab* is inserted.

When *MODE* is **i** all control characters are silently ignored. They are first grouped and then replaced with a newline character.

See the output of **--help** for the default value of *MODE*.

-e *ENC*, --encoding=*ENC*
Set (multiple) input search encodings. Encoding names *ENC* are identified according to the WATHWG standard. **--list-encodings** prints a list of implemented encodings.

See the output of **--help** for the default value of *ENC*.

-h, --help
Print a synopsis of available options and default values.

-l, --list-encodings
List available encodings as WHATWG Encoding Standard names and exit.

-n *MIN*, --bytes=*MIN*
Print only strings at least *min* bytes long. The length is measured as UTF-8 byte-string. **--help** shows the default value.

-p *FILE*, --output=*FILE*
Print to *FILE* instead of *stdout*.

-t *RADIX*, --radix=*RADIX*
Print the offset within the file before each valid string. The single character argument specifies the radix of the offset: **o** for octal, **x** for hexadecimal, or **d** for decimal. When a valid string is split into several graphic character sequences the cut-off point is labelled according to **--control-chars** but no additional offset is printed for each graphic character sequence.

The exception to the above is **--encoding=ascii --control-chars=i** for which the offset is always printed before each graphic character sequence.

When the output of **stringsext** is piped to another filter you may consider **--control-chars=r** to keep multi-line strings in one line.

-V, --version
Print version info and exit.

EXIT STATUS

0
Success.

other values
Failure.

EXAMPLES

List available encodings:

```
stringsext -l
```

Search for UTF-8 strings and strings in UTF-16 Big Endian encoding:

```
stringsext -e utf-8  -e utf-16be somefile.bin
```

Or:

```
cat somefile.bin {vbar} stringsext -e utf-8  -e utf-16be -
```

The following settings are designed to produce bit-identical output with *GNU strings*:

```
stringsext -e ascii -c i         # equals `strings`
stringsext -e ascii -c i -t d    # equals `strings -t d`
stringsext -e ascii -c i -t x    # equals `strings -t x`
stringsext -e ascii -c i -t o    # equals `strings -t o`
```

When used with pipes `-c r` is required:

```
stringsext -e ascii -e iso-8859-7 -c r somefile.bin {vbar} grep "Ιστορία"
```

LIMITATIONS

It is guaranteed that all valid string sequences are detected and printed whatever their size is. However due to potential false positives when interpreting binary data as multi-byte-strings, it may happen that the first characters of a valid string may not be recognised immediately. In practice, this effect occurs very rarely and the scanner synchronises with the correct character boundaries quickly.

When the size of a valid string exceeds FLAG_BYTES_MAX bytes it may be split into two or more strings and then printed separately. Note that this limitation refers to the *valid* string size and not to the *graphic* string size which may be shorter. If a valid string is longer than WIN_LEN bytes then it is always split. To know the values of the constants please refer to the definition in the source code of your **stringsext** build. Original values are: FLAG_BYTES_MAX = 6144 bytes, WIN_LEN = 14342 bytes.

RESOURCES

Project website: https://github.com/getreu/stringsext

COPYING

Copyright (C) 2016 Jens Getreu

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

```
http://www.apache.org/licenses/LICENSE-2.0
```

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

8.3. Benchmarking and field experiment

Rust's build in benchmarking feature allows to clock the time of unit testing code. At the time of this writing this feature is only available with the "nightly" distribution of the Rust compiler. It is especially valuable when used together with the *test driven development method* (cf. Section 6.3, "Test Driven Development"): First the programmer implements the unit testing code for a new feature. The second step consists in finding alternative solutions to implement this new feature. Using Rust's benchmarking the programmer

can take performance consideration into account at very early state when he is still exploring alternative solutions for the to be tested unit.

A second approach to benchmark software is to monitor the system resource usage of the running binary. The Linux-tool `time` runs programs and summarize system resource usage. This way we can compare the performance of *GNU-strings* and *Stringsext*. For this purpose the following script runs a series of 6 benchmark tests. In benchmark test 2 *Stringsext* is launched with only one ASCII scanner producing the same output as *GNU-strings* in benchmark test 1.

> The "Field experiment 1" compares the output of *GNU-strings* with the output of *Stringsext* in ASCII-only mode on real-life data. Both are expected to be identical.

The benchmark tests 3 to 5 are designed to study how *Stringsext* scale with more than one ASCII-scanner. The last benchmark 6 is a more realistic test case with 4 different scanners: ASCII, UTF-8, UTF-16BE and UTF-16LE.

All test operate on the same input data: a partition image with a Linux kernel `dev-sda.raw`.

Benchmark script

```
#!/bin/sh
FILE=dev-sda.raw
BMARK="$1-benchmark.txt"

echo "$(./stringsext -V)" >>"$BMARK"
echo "Inputfile: $(ls -l $FILE)" >>"$BMARK"

echo "\n\nBenchmark 1" >>"$BMARK"
time -vao "$BMARK" strings -n 10 -t x $FILE \
    > "$1-input_$FILE-output_orig.txt"

echo "\n\nBenchmark 2" >>"$BMARK"
time -vao "$BMARK" ./stringsext -c i -n 10 -e ascii -t x $FILE \
    > "$1-input_$FILE-output_1scanner.txt"

echo "\n\nField experiment 1" >>"$BMARK"
cmp --silent "$1-input_$FILE-output_orig.txt" \
             "$1-input_$FILE-output_1scanner.txt"
if [ $? -eq 0 ] ; then
   echo "    Success: Output of benchmark 1 and 2 is identical." \
        >> "$BMARK"
```

```
else
   echo "    FAILED! strings' and stringsext's output is different!" \
        |tee -a "$BMARK"  &&  exit 1
fi

echo "\n\nBenchmark 3" >>"$BMARK"
time -vao "$BMARK"  ./stringsext -n 10 -e ascii -e ascii -t x $FILE \
    > "$1-input_$FILE-output_2ascii.txt"

echo "\n\nBenchmark 4" >>"$BMARK"
time -vao "$BMARK"  ./stringsext -n 10 -e ascii -e ascii -e ascii -t x \
    $FILE  > "$1-input_$FILE-output_3ascii.txt"

echo "\n\nBenchmark 5" >>"$BMARK"
time -vao "$BMARK"  ./stringsext -n 10 -e ascii -e ascii -e ascii \
    -e ascii -t x $FILE > "$1-input_$FILE-output_4ascii.txt"

echo "\n\nBenchmark 6" >>"$BMARK"
time -vao "$BMARK"  ./stringsext -n 10 -e ascii -e utf-8 -e utf-16be \
    -e utf-16le -t x $FILE > "$1-input_$FILE-output_4scanners.txt"

echo "\n\n\n" >>"$BMARK"
```

The script is executed on a laptop with an Intel Core i5-2540M, 2.60GHz CPU.

Benchmark results

```
Version 0.9.4, (c) Jens Getreu, 2016
Inputfile:-rw-rw---- 1 jens myworkers 536870912 Aug 18 09:12 dev-sda.raw

Benchmark 1
 Command being timed: "strings -n 10 -t x dev-sda.raw"
 User time (seconds): 4.65
 System time (seconds): 0.06
 Percent of CPU this job got: 99%
 Elapsed (wall clock) time (h:mm:ss or m:ss): 0:04.72
 Maximum resident set size (kbytes): 2616
 File system outputs: 8552

Benchmark 2
 Command being timed: "./stringsext -c i -n 10 -e ascii -t x dev-sda.raw"
 User time (seconds): 11.26
 System time (seconds): 1.01
 Percent of CPU this job got: 106%
 Elapsed (wall clock) time (h:mm:ss or m:ss): 0:11.49
```

```
 Maximum resident set size (kbytes): 13032
 File system outputs: 8552

Field experiment 1
    Success: Output of benchmark 1 and 2 is identical.

Benchmark 3
 Command being timed: "./stringsext -n 10 -e ascii -e ascii -t x dev-
sda.raw"
 User time (seconds): 31.56
 System time (seconds): 1.52
 Percent of CPU this job got: 195%
 Elapsed (wall clock) time (h:mm:ss or m:ss): 0:16.91
 Maximum resident set size (kbytes): 19604
 File system outputs: 23176

Benchmark 4
 Command being timed: "./stringsext -n 10 -e ascii -e ascii -e ascii -t x
 dev-sda.raw"
 User time (seconds): 49.86
 System time (seconds): 2.51
 Percent of CPU this job got: 248%
 Elapsed (wall clock) time (h:mm:ss or m:ss): 0:21.08
 Maximum resident set size (kbytes): 26388
 File system outputs: 34752

Benchmark 5
 Command being timed: "./stringsext -n 10 -e ascii -e ascii -e ascii -e
 ascii -t x dev-sda.raw"
 User time (seconds): 71.66
 System time (seconds): 3.09
 Percent of CPU this job got: 312%
 Elapsed (wall clock) time (h:mm:ss or m:ss): 0:23.89
 Maximum resident set size (kbytes): 32692
 File system outputs: 46336

Benchmark 6
 Command being timed: "./stringsext -n 10 -e ascii -e utf-8 -e utf-16be -e
 utf-16le -t x dev-sda.raw"
 User time (seconds): 53.00
 System time (seconds): 9.29
 Percent of CPU this job got: 225%
 Elapsed (wall clock) time (h:mm:ss or m:ss): 0:27.64
 Maximum resident set size (kbytes): 18896
 File system outputs: 1177360
```

Table 8.4. Benchmark result synopsis

Benchmark	**% of CPU**	**Clock**	**Threads**	**% CPU ideal**	**Clock adjusted**
no.	this job got	measured elapsed time	scanner + merger/printer	required for optimal speed	adjusted for throttling
1	99%	*00:04.72*	1	100%	00:04.67
2	106%	*00:11.49*	1+1	106%	**00:11.49**
3	195%	00:16.91	2+1	212%	**00:15.55**
4	248%	00:21.08	3+1	336%	**00:15.56**
5	312%	00:23.89	4+1	448%	**00:16.64**
6	225%	00:27.64	4+1	448%	**00:13.88**

Observations

1. When scanning only ASCII, *GNU-strings* is 2.4 times faster than *Stringsext*. (Compare "% of CPU" benchmark 1 and 2).
2. The *merger/printer* thread consumes approximately 6% of the processor resources of one ASCII scanner thread.
3. In benchmark 4-6 *Stringsext* is slowed down because of missing hardware resources. (Compare column "% of CPU` this job got" and "% CPU ideal, required for optimal speed"). The threads are also throttled down because the processor temperature exceeds 80°C.
4. The column "Clock adjusted" show the adjusted value for throttling slow down we expect for a system with better hardware resources. The benchmarks where run on a laptop with an Intel Core i5-2540M CPU at 2.60GHz. Although this processor can run four threads concurrently, all threads have to share only two cores.
5. In line with expectations, the "maximum resident set size" of *Stringsext* depends on the number of threads launched. Its highest value of 32,7MB was observed in benchmark 5.
6. The "Field experiment 1" succeeds: *GNU-strings*' output and *Stringsext*'s output in ASCII-only mode are identical.

Conclusion

When launched as pure ASCII scanner *Stringsext* produces the same output as *GNU-strings*, but 2.4 times slower. This result is very satisfactory:

Stringsext's ASCII-only mode is only one special usage scenario among many others requiring complex time costly computing. When scanning for other encodings or for more than one encoding in parallel *Stringsext* can play off its particular strengths. It is best run on modern hardware with four or more kernels.

8.4. Product evaluation

In the Section 8.3, “Benchmarking and field experiment” we could convince ourselves that *Stringsext* produces accurate results timely. But how do matters stand with the other requirements defined in the Chapter 4, *Specifications*? Specifically:

Section 4.1, “User interface”

The user interface of Stringsext should reproduce GNU-strings' user interface as close as possible.

The command-line-options: `--bytes`, `--radix`, `--help`, `--version`, `-n`, `-t` and `-V` have the same meaning and syntax. The syntax of `--encoding` takes into account *Stringsext’s* advanced encoding support. The option `-w` is replaced by `-c MODE` offering a better output control.

Section 4.2, “Character encoding support”

Besides ASCII, Stringsext should support common multi-byte encodings like UTF-8, UTF-16 big endian, UTF-16 little endian, KOI8-R, KOI8U, BIG5, EUC-JP and others.

All the listed encodings are covered (see details in the Section 7.5, “Integration with a decoder library”). The found strings in multiple encodings are merged and presented in chronological order. The user can specify more than one encoding at the same time.

Section 4.3, “Concurrent scanning”

Each search encoding specified by the user is assigned to a separate thread.

This design specification is meet and detailed in the Section 7.1, “Concurrency”.

Section 4.4, “Batch processing”

All scanners operate simultaneously on the same chunk of the search field.

To meet this requirement a proprietary input reader with a circular buffer is implemented (cf. Section 7.7, “Polymorphic IO”).

Section 4.5, "Merge findings"

When all threads' findings are collected, the merging algorithm brings them in chronological order.

Different alternatives had been explored (cf. Section 7.8, "Merging vectors"). The implemented solution uses the `kmerge()`-function of the *rust/itertools* library.

Section 4.6, "Facilitate post-treatment"

Stringsext should have at least one print mode allowing post-treatment with line-oriented tools like `grep` or `agrep`.

The command-line-options `--radix=x` `--control-chars=r` print the offset of the finding, a tab character, the encoding name, a tab character and the found string in one line. Control characters in the found string are replaced with '  ' (U+FFFD). This output format facilitates post-treatment with line-orientated tools and spreadsheet applications.

Section 4.7, "Automated test framework"

Automated unit tests guaranty correct results for the implemented test cases. Furthermore, the chosen methodology makes sure that the tests are working as intended.

Stringsext has 17 unit tests. The chosen *test driven development* method (cf. Section 6.3.2, "Development cycle") guarantees that the unit tests work as intended.

Section 4.8, "Functionality oriented validation"

The same hard-disk image of approximate 500MB is analysed twice: first with GNU-strings then with Stringsext. If both outputs are identical, the test is passed.

This test, hereinafter referred to as "Field experiment 1" is executed with success and discussed in the Section 8.3, "Benchmarking and field experiment".

Section 4.9, "Efficiency and speed"

To address this requirement *Stringsext* is developed in the system programming language Rust (cf. Chapter 5, *The Rust programming language*). The satisfactory results are described and discussed in the Section 8.3, "Benchmarking and field experiment".

Section 4.10, "Secure coding"

This matter is addressed e.g. by choosing the new system programming language *Rust* offering various compile-time security guarantees (cf. Chapter 5, *The Rust programming language*). See also the analysis and

the discussion in the Section 2.2, "Security" and the Section 4.10, "Secure coding".

Conclusion

Stringsext meets all requirements defined in the Chapter 4, *Specifications*. Because of the inherent properties of the UTF-16 encoding, the UTF-16 scanners produce many false positives when run over binary data. A possible solution is suggested at the end of the Section 8.1.2, "UTF-16 encoded input".

8.5. User feedback

Before publishing *Stringsext*, a beta-version had been tested by a small group of forensic practitioners. In addition, the participants were invited to report back about desirable extensions or missing features:

1. String decoding based https://tools.ietf.org/html/rfc4648 (Base64 and others)
2. Base58 decoding
3. It would be nice that the list option `-l` displayed the supported encodings in alphabetic order, this would make easier to find the option we are looking for.

— User feedback: feature requests

Regarding additional encodings: *Stringsext* is designed to be extensible. Adding further encodings other than the ones listed in the Section 7.5, "Integration with a decoder library" is beyond the scope of this project, but it is made easy: As working sample encoding extension `ASCII_GRAPHIC` can be found in the source code of *Stringsext* in `src/codec/ascii.rs`. The request "ordered list" was implemented in version 0.9.5.

So far *Stringsext's* search algorithm is based solely on finding *valid* byte sequences for a given encoding. *Stringsext* is a pure *data processing system* in the sense that there are no semantics weather the resulting graphical character sequences make any "sense". The following suggestion received by email [22] goes far beyond this limitation.

> For future development: it would be nice to have some form of automatic detection of what encodings are more likely to be present in a given file, or even go further and do automatic de-

> tection of language like in Google translator (maybe you could upload selected words) [22].
>
> — Professor Miguel Frade *Computer Science and Communication Research Centre - Polytechnic Institute of Leiria*

This above idea opens the very interesting research field of *Computational Linguistics*. Language detection in character sequences requires a linguistic model of "what is a word" in a given human language. Thus, with the suggested enhancement *Stringsext* would become a *language processing system*.

Jurafsky [23 p. 3] illustrates the conceptual difference between a *data processing system* and a *language processing system* as follows: "What distinguishes language processing applications from other data processing systems is their use of knowledge of language. Consider the Unix `wc` program, which counts the total number of bytes, words, and lines in a text fi le. When used to count bytes and lines, `wc` is an ordinary data processing application. However, when it is used to count the words in a file, it requires knowledge about what it means to be a word and thus becomes a language processing system." Applied to *Stringsext* "the knowledge about what it means to be a word" comprises a probabilistic model about the likelihood that a certain character sequence represent a word in a given human language. It is clear that the approach is beyond the scope of this project. Nevertheless, the exiting challenge could be tackled in future research projects.

8.6. Licence and distribution

Stringsext is licensed under the Apache Licence, Version 2.0; you may not use this program except in compliance with the Licence. You may obtain a copy of the Licence at `http://www.apache.org/licenses/LICENSE-2.0`. The copyright remains with the author Jens Getreu.

Unless required by applicable law or agreed to in writing, software distributed under the Licence is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the Licence for the specific language governing permissions and limitations under the Licence.

The source code including its inline source documentation is hosted on Github [1]: `https://github.com/getreu/stringsext`. The project's main

page has links to the developer documentation and to the compiled binaries for various architectures.

Chapter 9. Development process evaluation and conclusion

Besides the contribution of the new tool *Stringsext* to the forensic community a more general consideration is of scientific interest: Seeing that Rust is a very young programming language: how well is the Rust ecosystem suited for forensic tool development?

Forensic tools have to fulfil stringent requirements concerning their quality: In general, huge amount of data has to be processed which leads to most demanding requirements in terms code efficiency (cf. Section 2.3, “Code efficiency”). Furthermore, the data to be analysed is potentially dangerous: it may contain malicious payload targeting common vulnerabilities (cf. Section 2.2, “Security”). Finally, in order to fulfil legal requirements forensic tools must be extensively tested.

The present case study confirm my initial hypothesis that Rust addresses theses requirements (cf. Chapter 5, *The Rust programming language*): Rust, as system programming language, is designed for code efficiency. Rust’s security guaranties comprise memory safety, the cause for a common category of vulnerabilities. It’s build in unit testing feature supports *software verification* as defined in the Section 2.1, “Tool validation”.

Guaranteed memory safety is a core property of Rust’s borrow checker: When a Rust source code compiles, the resulting binary *is* guaranteed to be memory safe. In consequence, such a binary is immune to memory safety related attacks: e.g. out-of-bounds read, buffer over-read, heap-based buffer overflow, improper validation of array index, improper release of memory before removing last, double free, use after free. As *Stringsext* and all its used libraries are solely Rust components, *Stringsext* is memory safe.

In the Section 8.3, “Benchmarking and field experiment” we compared the code efficiency of *GNU-strings* implemented in C and *Stringsext* implemented in Rust. When *Stringsext* is run in ASCII-only mode, both produce the same output. The field experiment yielded the expected result, 2.4 times slower but still on the same scale. However, *Stringsext’s* design implies much more complex computations, hence the result is not surprising.

How about the efficiency of Rust's abstractions and its overall performance? A good estimation is to compare benchmarks of small and simple programs. Too complex programs should be avoided for this purpose because variations of the programmer's skills may bias the result. According to the "Computer Language Benchmark Game" [24] Rust and *C/C++* have similar benchmark results.

Forensic tools have to operate on many architectures. Here enters Rust's cross-compiling feature on scene:

> As *Rust* uses the LLVM framework as backend, it is available for most platforms. `rust-lang-nursery/rustup.rs` [25] is a Rust toolchain multiplexer. It installs and manages several toolchains in parallel and presents them all through a single set of tools installed. Thanks to the LLVM backend, it's always been possible in principle to cross-compile Rust code: just tell the backend to use a different target! And indeed, intrepid hackers have put Rust on embedded systems like the Raspberry Pi 3, bare metal ARM, MIPS routers running OpenWRT, and many others.

As described above, Rust's memory safety guarantee is a huge improvement in terms of security because a whole category of potential vulnerabilities can be ruled out from the outset. But memory safety does not mean bug freeness! Beside the security aspects discussed above, the correctness of forensic software is crucial (cf. Section 2.1, "Tool validation"). It is clear that the overall correctness of a program depend also on the correctness of every library used. Hence, the question arises whether the Rust ecosystem is mature enough to meet the ambitious requirements of forensic software. Indeed, compared to C, Rust's libraries are relatively young. Here again extensive unit testing revealed to be a helpful diagnostic method: version 0.4.16 of the brand new `kmerge` function, part of the `itertools` library used in *Stringsext*, reversed under rare conditions the first and second finding. This bug was actually fixed with pull request #135 (2. Aug. 2016) some days after its appearance. Although the bug-fix was already committed in Github, the package manager did not know about it, because no new version of `itertools` was released yet. On the whole, a little change in the package reference list `Cargo.toml` solved the problem immediately. Finally, it took another week for the corrected `itertools` version to be released. So far this was the only time I encountered a bug in any of the used libraries.

One conclusion we can draw from this experience, is that young libraries are more likely to have bugs than established ones. It cannot be emphasised enough that, diligent unit tests help to find most bugs at early state. Also those present in external libraries. However, unit testing do not help against memory safety related vulnerabilities, which are typical for C and C++ programs and which can persist in software for decades. It is incumbent on readers to form their own opinion, I largely prefer accepting the greater likelihood of manageable bugs related to young Rust libraries, than the uncertainty of hidden memory safety related vulnerabilities typical for *C* and *C++*.

Rust code has the reputation that it is easy to read and understand, but it is hard to write. I subscribe to this point of view. Rust's biggest strength is that unsafe code does not compile, can be also very frustrating. Especially when you do not understand the compiler's error messages. At some stage it even happened, that I run out of ideas how to fix a particular problem. Fortunately, the Rust Internet community is very supporting and helpful. In the meantime, also Rust's error messages improved with version 1.12 and Rust's documentation is steadily updated and enhanced.

The benefits of *unit testing* had been stressed throughout this work. The chosen software development method for this project was the *test driven development* method where *unit testing* is the key element. Contrary to other methods unit tests and the to be tested code is always programmed by the same person. The Section 6.3, "Test Driven Development" describes the method more in detail and shows why it was good choice under the given circumstances. However, other methods may be as suitable depending on the organisational structure of the programmer team.

Conclusion

Looking back, Rust was a very good choice for the present project, even though batch processing of multi-bytes character streams revealed to be far more complex than expected. Additionally, concurrent programming in Rust posed a formidable hurdle at the beginning. Fortunately, it did prove to be helpful to contact the Rust community for their friendly assistance. In addition, for a not so experienced Rust programmer it is reassuring to know that when a complex piece of code finally compiles, it is memory safe. The same reasoning applies when a programmer has to refactor existing code. I often had a queasy feeling when I had to work on other people C code. Do I free the memory at the right moment? Is this pointer still valid? Rust's

ownership paradigm resolves this uncertainty. When it compiles, then it is memory safe. Furthermore, Rust is especially suitable for bigger projects where several programmers contribute to the same code. And this is particularly true when developing forensic software with its high quality standards.

It has to be noted though that the Rust ecosystem is still very young and bugs in new libraries are nothing uncommon. Fortunately, the library maintainers are very responsive and a bug is usually fixed within days. Here again unit testing becomes handy. It does not only find bugs in our own code at early stage, it also helps to identity bugs in external libraries. Used together with the *test driven development* method, the test code and the to be tested code can be validated in one go.

Stringsext is especially useful where *GNU-strings* fails: For example recognizing multi-byte characters in UTF-16. In order to realise *Stringsext's* full potential an additional filter, limiting the Unicode output to a chosen set of scripts, would be desirable.

A major focus of future development will be aiming to reduce the number of false positives especially when scanning for UTF-16 in binary data. A practicable solution could be a parametrizable additional filter limiting the search to a range of Unicode blocks.

As of *Stringsext* version 1.1 [1], the `--encoding` option interprets specifiers limiting the search scope to a range of Unicode blocks.
For example `--encoding utf-16le,8,U+0..U+3ff` searches for strings encoded in UTF-16 Little Endian being at least 8 bytes long and containing only Unicode codepoints in the range from `U+0` to `U+3ff`. Please consult the man-page for details.

[1] This present document describes *Stringsext* 1.0. The new Unicode-range-filter feature released with *Stringsext* version 1.1 was published after the writing of this thesis.

References

1. J. Getreu, “Stringsext, a GNU Strings Alternative with Multi-Byte-Encoding Support.” Tallinn, Jan-2016.
2. D. Meuwly, “Case Assessment and Interpretation in Digital Forensic Casework. Cyber Security Summer School 2016: Digital Forensics, Technology and Law.” Tallinn, May-2016.
3. Y. Guo, J. Slay, and J. Beckett, “Validation and Verification of Computer Forensic Software tools—Searching Function,” *Digital Investigation*, vol. 6, pp. S12–S22, Sep. 2009.
4. V. S. Harichandran, D. Walnycky, I. Baggili, and F. Breitinger, “CuFA: A More Formal Definition for Digital Forensic Artifacts,” *Digital Investigation*, vol. 18, pp. S125–S137, 2016.
5. J. Beckett and J. Slay, “Digital Forensics: Validation and Verification in a Dynamic Work Environment,” 2007, pp. 266a–266a.
6. P. Craiger, J. Swauger, C. Marberry, and C. Hendricks, “Validation of Digital Forensics Tools,” *Digital crime and forensic science in cyberspace. Hershey, PA: Idea Group Inc*, pp. 91–105, 2006.
7. S. Berinato, “The Rise of Anti Forensics.,” *CSO Online*. http://www.csoonline.com/article/2122329/investigations-forensics/the-rise-of-anti-forensics.html , Aug-2007.
8. T. Eggendorfer, “IT Forensics. Why Post-Mortem Is Dead. Cyber Security Summer School 2016: Digital Forensics, Technology and Law.” Tallinn University of Technology, Jul-2016.
9. “Log Message: Sourceware Import,” *Mail archive of the binutils-cvs @sourceware.cygnus.com mailing list for the binutils project*. https://sourceware.org/ml/binutils-cvs/1999-q2/msg00000.html , Mar-1999.
10. M. Zalewski, “PSA: Don’t Run ’strings’ on Untrusted Files (CVE-2014-8485),” *lcamtuf’s blog*. Oct-2014.
11. US-CERT/NIST, “Vulnerability Summery for CVE-2016-3861,” *National Vulnerability Database*. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-3861 , Nov-2016.
12. M. I. T. R. E. Corporation, “CWE - Common Weakness Enumeration, a Community-Developed Dictionary of Software Weakness Types.” https://cwe.mitre.org/ , 2016.

13. The-Rust-Project-Developers, *The Rustonomicon*. 2016.

14. A. Liao, “Rust Borrow and Lifetimes.” http://arthurtw.github.io/2014/11/30/rust-borrow-lifetimes.html , Nov-2014.

15. K. Beck, *Test-Driven Development: By Example*. Addison-Wesley Professional, 2003.

16. The-Rust-Project-Developers, *The Rust Programming Language*. 2016.

17. D. Bargen, “How Does Rust Handle Concurrency? - Quora.” Dec-2016.

18. *The Unicode Standard, Version 9.0.0 Core Specification*, vol. 9. Mountain View,: Unicode Consortium, 2016.

19. K. Seonghoon, “Character Encoding Support for Rust: Rust-Encoding.” Aug-2016.

20. J. Goulding, “Rust Implementing Merge-Sorted Iterator,” *Stack Overflow*. http://stackoverflow.com/questions/23039130/rust-implementing-merge-sorted-iterator , Aug-2015.

21. R. Lehmann, “The Sphinx Project,” Universität Potsdam, Project Documentation, 2011.

22. M. Frade, “E-Mail: GNU Strings Reimplementation.” Nov-2016.

23. D. Jurafsky and J. H. Martin, *Speech and Language Processing*. Pearson, 2014.

24. B. Fulgham and I. Gouy, “C G vs Rust (64-Bit Ubuntu Quad Core) | Computer Language Benchmarks Game.” http://benchmarksgame.alioth.debian.org/u64q/compare.php?lang=gpp&lang2=rust , Oct-2016.

25. B. Anderson, “Taking Rust Everywhere with Rustup - The Rust Programming Language Blog,” *The Rust Programming Language Blog*. https://blog.rust-lang.org/2016/05/13/rustup.html , May-2016.