THE JOURNAL OF DIGITAL FORENSICS, SECURITY AND LAW

Journal of Digital Forensics, Security and Law

Volume 14 | Number 2 Article 4

6-30-2019

Enhancing Forensic-Tool Security with Rust: Development of a String Extraction Utility

Jens Getreu
Taltech,Tallinn University of Technology

Olaf Maennel
Taltech, Tallinn University of Technology

ENHANCING FORENSIC-TOOL SECURITY WITH RUST: DEVELOPMENT OF A STRING EXTRACTION UTILITY

Jens Getreu, Olaf Maennel
Taltech, Tallinn University of Technology
Ehitajate tee 5
19086 Tallinn, Estonia

ABSTRACT

The paper evaluates the suitability of the *Rust* ecosystem for forensic tool development. As a case study, a forensic tool named *Stringsext* is developed. Starting from analyzing the specific requirements of forensic software in general and those of the present case study, all stages of the software development life-cycle are executed and evaluated.*Stringsext* is a re-implementation and enhancement of the *GNU-strings* tool, a widely used program in forensic investigations. *Stringsext* recognizes Cyrillic, CJKV East Asian 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.

Keywords: Forensic analysis, string search, multi-byte encoding, Rust language, `Stringsext`-tool

1. INTRODUCTION

Human interaction with electronic devices leaves traces in their electronic memory. In the age of cloud computing most human interaction triggers requests to distant servers leaving traces not only in their log files, but also in many intermediate network devices. Due to the cross-linked nature of computer systems the data that needs to be taken into consideration when investigating a crime is enormous. In this huge amount of information the investigator has to find those specific bits of information constituting digital evidences. In the domain of digital forensics an electronic trace (observation) with a well-known cause-effect-relationship between the observation and the human action causing it (activity), is a so called **artefact**. For the sake of simplicity we present here a rather simplistic view associating univocally one trace (artefact/observation) with only one possible cause/activity. In the physical world, one trace might have several possible causes which cannot be excluded a priori. For this reason, modern forensics favors the *Likelihood Ratio* approach, in which the degree of support of the observations for the hypotheses is considered, providing a strength of evidence (diagnostic value) that is relative to the hypotheses tested. It embodies any "item of interest that helps an investigation move forward." (Harichandran, Walnycky, Baggili, & Breitinger, 2016, p. 125)[1].

Forensic examiners, the law enforcement personnel who deal with digital evidence, face inter alia two challenges: to collect and to preserve the huge amount of data that may be related to a crime and to search and detect artifacts in the collected data. The latter aspect implies the so called *string search* which is useful when dealing with unknown binary data. Most binary data contain human readable character sequences called strings. A very commonly used program to extract strings from a binary data is the so called *GNU-strings* program. Although still widely used, *GNU-strings* has only a very rudimentary support for multi-byte encodings such as Unicode. Furthermore, it was subject several to memory safety vulnerabilities which exclude the handling of potentially harmful forensic data. The software tool *Stringsext*, developed in this present work, is made for same purpose. The new development is designed to overcome both shortcomings. Where possible, it maintains *GNU-strings'* user-interface.

In the following section we analyze general tool requirements in digital forensics. Together with the shortcomings of the original *GNU-strings* tool, it becomes obvious that the C++ programming language does not satisfy the indispensable security requirements in the field of forensic software. The remaining sections show how the Rust-language mitigates them and what should be observed during implementation.

[1]Cf. Harichandran (Harichandran et al., 2016, p. 131) who proposed a more formal definition: A *Curated (digital) Forensic Artefact (CuFA)* "must have evidentiary value in a legal proceeding, must be created by an external force/artificially, must have antecedent temporal relation/importance and must be exceptional (based on accident, rarity, or personal interest)[...]".

2. TOOL REQUIREMENTS IN DIGITAL FORENSICS

2.1 Multi-byte character encoding support

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

Validation is the confirmation by examination and the provision of objective evidence that a tool, technique or procedure functions correctly and as intended (Craiger, Swauger, Marberry, & Hendricks, 2006, p. 92).

One way of establishing a set of requirements for a new forensic tool is to analyze how similar existing tools are validated. Beckett and Slay (Beckett & Slay, 2007) propose a *functionality oriented validation* method called *Model of tool neutral testing.* Instead of testing if a software product meets all its requirements, an independent set of forensic functions is defined and later tested. The digital forensic discipline can be broadly defined in terms of the key functions *Identification, Data Preservation, Data Analysis and Presentation of Digital Evidence.* Each key function is further divided into subcategories. For example *Data Analysis* breaks down into: *Searching, File Rendering, Data Recovery, Decryption, Identification, Processing, Temporal Data* and *Process Automation*. The first item *searching* relates to finding and locating information of interest in digital memories. Form a functional point of view the *searching* falls into the searching domain (where), the searching mode (how) and the searching target (what). This breakdown of forensic functions into detailed categories allows to decouple the validation procedure from the implementation of the forensic tool itself. Based on this categorization, independent test environments are set up for each specific function. For example, a test may certify if the software is able to run a fuzzy search for strings in the unallocated disk space.

Beyond validation, the categorization of forensic functions is also helpful to define requirements for improvement existing tools — in our case *GNU strings*. For example, *Searching* can be further classified into groups as shown in the Figure 1 (Beckett & Slay, 2007, p. 17). The leaves in the diagram list typical capabilities a *Searching*-software can implement. For example the subcategory "Character encoding" illustrates the main deficit of *GNU-strings* as it supports only ASCII encoding. In a global cyberspace, forensic tools must identify a multitude of encodings. This leads us to the main motivation and requirement of *Stringsext*: *multibyte character encoding support.*

The *functionality oriented validation* can be classified as "black box testing" examining functionality without any knowledge of the internal implementation, even without having access to the source code. Its advantage is, that it allows to reuse the test environment thus reducing costs. Black box testing is sometimes also referred as "behavioral testing" as it feeds the tool with known test case data and observes if the tool outputs the expected results. In the context of this work black box testing is used to assert that the developed tool *Stringsext* deals correctly with big real-world input data: *Stringsext* has been designed to produce, as a special case, bit-identical output to that of *GNU-strings*. This way the comparison of the output data of both tools allowed to confirm that *stringsext* is working correctly when dealing with big data.

In addition to the above black box testing, Rust's build in test harness (unit test) is used in conjunction with the *test driven development* in order to guarantee maximum security and correctness of the developed tool.

2.2 Security

"Make it hard for them to find you and impossible for them to prove they found you". With this concise word Berinato (2007) characterizes the relation between the criminal and the forensic investigator. In digital forensics this "hide-and-seek" game might soon take a new dimension: Eggendorfer (2016) stresses with good reasons that forensic tools are software too and therefor vulnerable to attacks: A vulnerability found in 2017 in a common forensic tool *EnCase Forensic Imager* demonstrates exemplary the pertinence of the risk (Consult, 2017):

> *By writing a manipulated LVM2 partition (a hard disk format commonly used for Linux servers) on a storage device, an attacker could — if the device were ever to be analyzed using EnCase Forensic Imager — take over an investigator's machine. When the investigator tries to read the device, EnCase Forensic Imager crashes — unbeknownst to the investigator, however, a lot more is happening. Through a buffer overflow security*

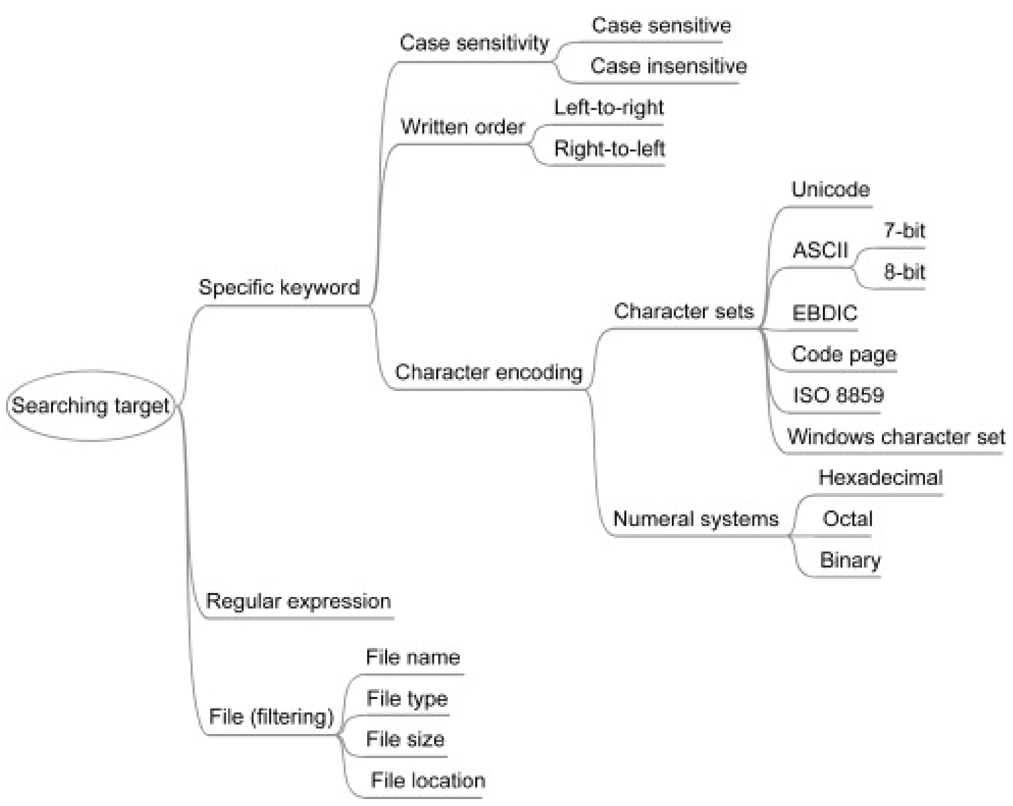


Figure 1. The search target mapping, cf. (Beckett & Slay, 2007, p. 8)

> *flaw, EnCase Forensic Imager can be tricked into executing data read from the storage device. Afterwards the code provided by the attacker has full control of the investigator's machine and can be used by the suspect to manipulate evidence (Consult, 2017).*

Confronted with this vulnerability report, the vendor downgraded the issue: "The exploit SEC Consult claims to have found is an extreme edge case, much like the theoretical alerts they tried to promote in November. [...] We do not consider this claim to be serious and it will not impact the performance of our products (Consult, 2017)." For the user it remains unclear if and when the vulnerability gets fixed. While such a reaction would have been inconceivable in other software domains, the risk of forensic tool exploitation is still largely neglected despite its potential impact:

- The adversary may be warned about an ongoing investigation.
- The adversary may gain control of the investigator's machine and alter evidences.

The*Stringsext*-project addresses this risk by choosing the programming language Rust. Rust offers some outstanding security guarantees which are presented in Section 4.

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 also requires carefully chosen abstractions, efficient algorithms avoiding unnecessary data-copies and program-loops. Concerning the choice of the programming language we define the following requirements: The programming language should:

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

The above motivates the choice of developing *Stringsext* in Rust. Chapter 4 shows how Rust meets the above requirements by its *memory safety* guarantees and *zero cost* design goal.

3. *GNU-STRINGS'* SHORTCOMINGS IN FORENSICS

This section first analyzes *GNU-strings'* limitations concerning multi-byte-encodings and international scripts. Based upon this we derive a set of additional requirements for *Stringsext*. Many forensic practitioners use the GNU program *strings*, hereafter referred as *GNU strings*, to get a sense of the functionality of an unknown program by extracting human readable strings from binary data. Of special interest are for example strings containing URLs to malicious sites, often an indicator of malware activity. But also, user prompts, error messages, and status messages can give valuable hints.

3.1 International character encodings

As discussed above the main motivation for developing *Stringsext* is the missing multi-byte character encoding semantics in *GNU-strings*. *GNU-strings* encoding support consists of a rudimentary filter accessed with the option `--encoding`. Please consult the manual page for more details. How well does *GNU-strings* detect Unicode? The Figure 2 shows the content of a text file chosen as test case. To find out how well *GNU*

```
Arabic:     حبل الكذب قصير
Chinese:    師傅領進門，修行在個人
French:     Les pâtes
Greek:      Ιστορία
German:     Viele Grüße
Russian:    Поздравляю
Symbols:    €, 𝄞
```

Figure 2. Test case international character encodings

Strings deals with different Unicode encodings, the text-file is then is converted into `UTF-8`, `UTF-16LE`, `UTF-16BE`, `UTF-32LE` and `UTF-32BE`, each encoding in one file. In order to observe *GNU-strings* Unicode detection capabilities, all the test-files are then searched for valid graphic strings with the command `strings` using all possible variation of its encoding filter. The Figure 3 shows exemplary *GNU-strings* output for a UTF-16 little endian encoded input.

```
  2 Arabic:
 38 Chinese:
 62 French:        Les p
 8e Greek:
 b4 German:        Viele Gr
 e4 Russian:
10c Symbols:
```

Figure 3. GNU-strings with UTF-16LE encoded input

Results: `UTF-8` is the only encoding in which *GNU strings* is able to find international characters. The Figure 3, chosen as an example, shows that with `UTF-16` input, *GNU strings* fails to recognize all non-ASCII characters. The same holds true for `UTF-32` and most other encodings: This limitation is of particular 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* is not designed to analyze multi-byte encodings in general. This is why other very common encodings e.g. *big5* or *koi8-r* are not supported at all even though they are widely used. The above-outlined limitations leads to *Stringsext*'s main requirement: *character encoding support.*

3.2 Secure coding

In the narrow sense, "secure coding" is rather a design goal than a functional requirement. Secure coding denotes the practice of developing computer software by reducing the accidental introduction of security vulnerabilities by preventing coding errors or discovering and eliminating security flaws during implementation and testing. From the secure coding point of view the requirement *character encoding support* is the most critical: The NIST *National Vulnerability Database* lists under the heading "character encoding" 22 vulnerabilities. Not only new complex forensic software is affected by vulnerabilities. It also concerns other well-established products: The tool *GNU-strings*, part of the *GNU binutils* collection, became publicly available in 1999 (Cygnus-Solutions, 1999). 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 (2014) discovered a serious security vulnerability *CVE-2014-8485* :

> *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.*

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.

Of particular importance is that the above bugs are part of a vulnerability class related to memory safety problems. *GNU strings* is written in C, a language whose abstractions can not guarantee memory safety. In order to exclude potential vulnerabilities of the same kind from the outset, *Stringsext* was developed in the *Rust* programming language.

4. THE RUST PROGRAMMING LANGUAGE

In the Section 2 we showed that the requirements *code efficiency* and *security* are of paramount importance. This section illustrates how Rust supports these requirements with its *zero cost abstractions* and its *guaranteed memory safety* (The-Rust-Team, 2019) motivating the choice of implementing *Stringsext* in Rust.

4.1 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 1 shows a selection of most common memory safety related vulnerabilities (Corporation, 2016). Memory safe languages like *Java* do not give programmers direct and uncontrolled access to pointers. For example, in *Java* memory safety is guaranteed through 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.

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 1 are already detected at compile time. Moreover, the *Rust* compiler 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*. Put in simplified terms, they translate to the following three rules which Rust's type system enforces at compile time:

1. All resource (everything that can be bound to a variable, e.g. integers, vectors, structures) has a unique owner.
2. Others can borrow from the owner (technically borrowing means that another scope sets a pointer to the owner's resource).
3. The owner cannot free or mutate the resource while it is borrowed.

By enforcing the above rules *Rust* organizes how resources are shared among different scopes. Memory problems occur for instance when a resource is referenced by multiple pointers (aliasing) and when more than one pointer writes to the same memory at the same time (mutation). 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 (cf. Table 2). 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.

The following code samples (The-Rust-Project-Developers, 2017, Sec. 3.2) illustrate how the Rust compiler detects non-obvious hidden memory safety issues. The function `as_str` 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`.

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

Table 1. Common weaknesses in C/C++ that affect memory

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

Table 2. Resource sharing in Rust

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

Here the corrected memory safe code performing a byte-copy at the end of the function.

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

The `push()` method in line 3 of the next example causes the backing storage of `data` to be reallocated. As a result we have a dangling pointer vulnerability! Again, the Rust compiler detects the error and code does not compile.

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

Here the corrected memory safe version that compiles:

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

4.2 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 3 (Corporation, 2016)).

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

Table 3. Common weaknesses in C/C++ affecting memory avoidable with iterators (Corporation, 2016)

In addition to traditional imperative loop control structures, *Rust* offers efficient iteration with functional style iterators. Like in Haskell iterators are lazy and avoid allocating memory for intermediate structures (you only allocate just when you call `.collect()`). Iterators considerably enhance the robustness and safety of programs. They enable the programmer to iterate through vectors without explicitly naming neither the index nor its bounds, thus avoiding common mistakes. The Figure 4 shows an example.

```
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);
    }
```

Figure 4. Vigenère cipher in Rust

4.3 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 shows a vector in *Rust*. Its memory layout is very similar is to a vector in *C++*. In contrast, the

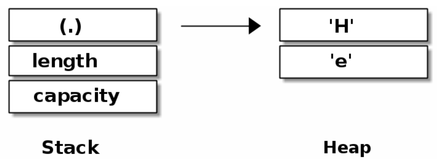


Figure 5. Memory layout of a Rust vector

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. Fig. 6). As the data could be represented in a more efficient way in memory, we see that *Java* does not prioritize *Zero-Cost-Abstraction* as primary objective.

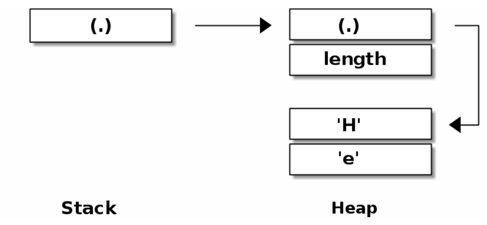


Figure 6. Memory layout of a Java vector

5. USE CASE: DEVELOPMENT OF THE *STRINGSEXT*-TOOL

In the Section 2 we have analyzed the special requirements for tools in digital forensics: *code efficiency* and *security*. The Section 4 showed that Rust's core properties *zero cost abstractions* and *memory safety* in theory meet well our requirements.

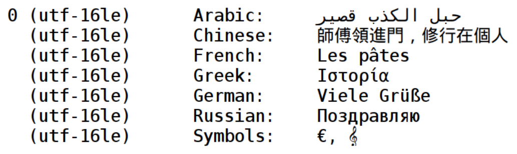
But how well is the Rust ecosystem suited for forensic tool development? In order to evaluate also the practical aspects of Rust, the tool *Stringsext* (Getreu, 2018)(Getreu, 2017) was developed. The technical challenges such as concurrent batch processing of multi-byte character streams revealed to be sufficiently complex, thus allowing us to deduce general recommendations for forensic-tool development.

5.1 Encoding support

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?

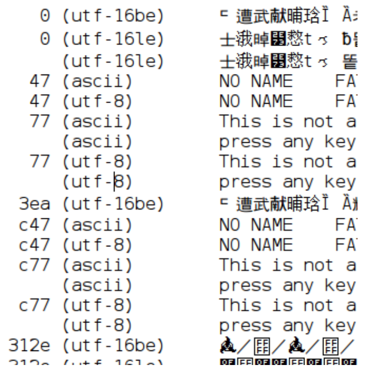
To evaluate *Stringsext*'s capabilities to handle international scripts with Unicode, we choose the same input text file (cf. Figure 2) we used with *GNU-strings* in the Section, 3.1. *Stringsext*'s output (cf. Figure 7) confirms that all international characters are found correctly. Furthermore, Figure 8 illustrates how *Stringsext*'s formats its output when it operates in simultaneous multi-encoding scanning mode.

Stringsext's unique Unicode range restriction feature has shown itself to be of use in practice: The Microsoft-Windows operating system handles Unicode characters in memory as 2 byte UTF-16 words. Thus, searching for UTF-16 encoded strings in memory images is common practice in forensics. This is more difficult as expected, as almost every random byte combination is assigned to a valid Unicode code point. Without further measures such a search leads to many false positives and unusable results. The offered solution by *Stringsext* allows the user to restrict the Unicode-range: For example, the Unicode-range `U+400-U+7FF` captures only strings in Cyrillic, Armenian, Hebrew, Arabic and Syriac. This feature is particularly useful when dealing with UTF-16 in memory images.



```
0 (utf-16le)      Arabic:      حبل الكذب قصير
  (utf-16le)      Chinese:     師傅領進門，修行在個人
  (utf-16le)      French:      Les pâtes
  (utf-16le)      Greek:       Ιστορία
  (utf-16le)      German:      Viele Grüße
  (utf-16le)      Russian:     Поздравляю
  (utf-16le)      Symbols:     €, 𝄞
```

Figure 7. Stringsext's output with `UTF-16LE` encoded input



```
   0 (utf-16be)      ᄃ 遭武献晡琀Ï Ä
   0 (utf-16le)      土涐啅瞷慹t ㄘ Ђ
     (utf-16le)      土涐啅瞷慹t ㄘ 
  47 (ascii)         NO NAME    FA
  47 (utf-8)         NO NAME    FA
  77 (ascii)         This is not a
     (ascii)         press any key
  77 (utf-8)         This is not a
     (utf-8)         press any key
 3ea (utf-16be)      ᄃ 遭武献晡琀Ï Ä
 c47 (ascii)         NO NAME    FA
 c47 (utf-8)         NO NAME    FA
 c77 (ascii)         This is not a
     (ascii)         press any key
 c77 (utf-8)         This is not a
     (utf-8)         press any key
312e (utf-16be)      ⽆/⾕/⽆/⾕/
```

Figure 8. Stringsext's output in multi-encoding scanning mode

5.2 Concurrency

The Figure 9 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. Rust supports by default two models of inter-thread communication: shared memory and message channels.

Stringsext first cuts the input stream into overlapping slices of shared read-only memory pages, which are then fed into the different *scanner* threads. Each scanner operates independently in batch. It runs through the input-slice, searches for valid string sequences, collects them in a list and finally sends the list to a *merger-printer*-thread though a dedicated *message channel*. This thread collects all lists from the different scanners and merges them into the final output stream.

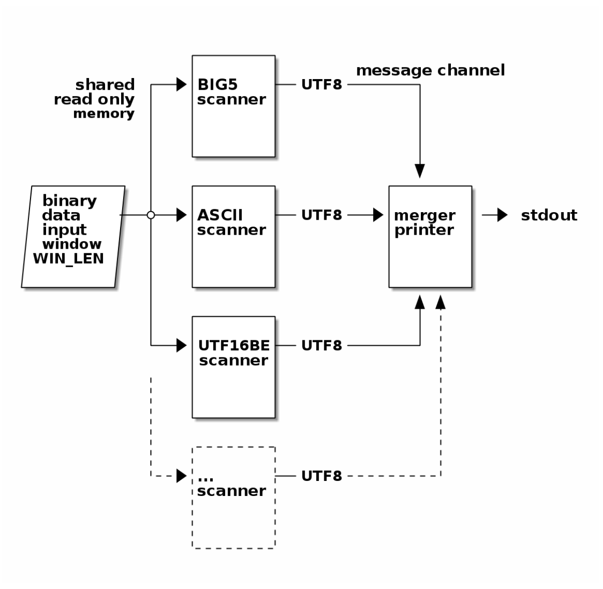


Figure 9. Data processing and threads

5.3 Algorithm

Batch processing of multibyte character streams turned out to be more difficult than expected. As we want to keep the scanning process as stateless as possible, we introduced overlapping windows: This allows reading beyond the memory page's edge in case a found string terminates in the next memory page. However, strings can be so long that you can't but cut them somewhere. In this case we need to make sure not to cut in the middle of a multi-byte character which can be up to 6 bytes long. As a result a scanner is not completely stateless: between each scanner run the position where the last run stopped and a flag indicating the forced cut of a string is passed.

The *scanner* decodes the input stream in two phases: first it uses the *Encoding*-crate library to identify valid code sequences and transcodes them into valid UTF-8 strings. Such a valid string may contain non-graphical and graphical characters. As we are only interested in the latter, a second filter extracts graphical substrings while meeting additional criteria, e.g. minimum length or Unicode-range restriction rules.

6. DEVELOPMENT PROCESS EVALUATION

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 fulfill 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). Furthermore, the data to be analyzed is potentially dangerous: it may contain malicious payload targeting common vulnerabilities (cf. Section 2.2). Finally, in order to fulfill legal requirements forensic tools must be extensively tested.

The present case study confirms our initial hypothesis that Rust meets these requirements: Rust, as system programming language, is designed for code efficiency. In addition Rust guarantees 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.

Memory safety is checked at compile time by 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.

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 bit-identical output. The field experiment yielded that even though *Stringsext's* 2.4 times slower, the speed is within the same order of magnitude. 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" (Fulgham & Gouy, 2019) Rust and *C/C++* have similar benchmark results, which confirms our above measurements.

Forensic tools have to operate on many architectures. Here enters Rust's cross-compiling feature on scene. As *Rust* uses the LLVM framework as back-end, it is available for most platforms. `rustc --print target-list` lists 80 compiler targets (`rustc` version 1.27).

As discussed 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. However, memory safety does not necessarily mean there is no bug. Beside the security aspects discussed above, the correctness of forensic software is crucial (cf. Section 2.1). It is clear that the overall correctness of a program also depend 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: for example one of the first versions 0.4.16 of the `kmerge` function, part of the `itertools` library used in *Stringsext*, reversed under rare conditions the first and second finding. Unit testing revealed a bug in `itertools`. The programmer responded quickly. It took only some days after its appearance that the bug was fixed with pull request #135. So far this was the only time we encountered a bug in the used libraries. One conclusion we 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 testing helps to find most bugs at early state. 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. Taking into account these benefits and drawbacks we 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++*.

Finally, we recognize the benefits of *unit testing* throughout this work. For this reason we chose for this project the *test driven development* method where *unit testing* is the key element. Contrary to other methods, in *test driven development* unit tests and the to be tested code is always programmed by the same person, which fitted well the setting of this project. However, other methods may be as suitable depending on the organisational structure of the programmer team.

7. CONCLUSION

We have shown that forensic tool development is subject to special requirements in terms of memory safety and code efficiency. Both are inherent properties of the used programming language and its ecosystem. Even though the language design of C and C++ allows generating very efficient code, there is no guaranty that the compiled code is memory safe. Not only that the lack of memory safety is one of the principal causes for most software vulnerabilities, forensic software is particularly exposed to such risks as it processes binary data of unknown origin containing all kinds of malware. As an alternative to C and C++, the relatively new programming language Rust offers guarantied memory safety by design, while being as fast as C. The match-making of general forensic tool requirements and the theoretical properties of the Rust programming language makes it an ideal alternative to the hitherto dominant programming language C++. But a general recommendation for a shift in programming practises can not be deduced from theoretical considerations alone: this is why we developed the *Stringsext*-Software as use case.

The use case "development of the *Stringsext*-tool" shows that Rust was a 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 hurdle at the beginning. Fortunately, the friendly Rust community helped to overcome occasional technical obstacles quickly. 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 applies to common settings when a programmer has to refactor existing code. Rust clears away doubts like "Do I free the memory at the right moment? Is this pointer still valid?" 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 currently in production state and can be used in forensic investigations as a *GNU strings* replacement. It is especially useful where *GNU-strings* fails: it allows finding UTF-16 multi-byte characters in memory images and supports other multi-byte encodings like *big5* or *koi8-r*.

REFERENCES

Beckett, J., & Slay, J. (2007). Digital forensics: Validation and verification in a dynamic work environment. In *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference* (p. 266a-266a). IEEE.

Berinato, S. (2007, June). *The Rise of Anti Forensics.* http://www.csoonline.com/ article/ 2122329.

Consult, S. (2017, May). *Chainsaw of Custody: Manipulating forensic evidence the easy way.*

Corporation, M. (2016). *CWE - Common Weakness Enumeration, a Community-Developed Dictionary of Software Weakness Types.* https://cwe.mitre.org/.

Craiger, P., Swauger, J., Marberry, C., & Hendricks, C. (2006). Validation of digital forensics tools. *Digital crime and forensic science in cyberspace. Hershey, PA: Idea Group Inc*, 91-105.

Cygnus-Solutions. (1999, May). *Log message: Sourceware import.* https://sourceware.org/ ml/ binutils-cvs/ 1999-q2/ msg00000.html.

Eggendorfer, T. (2016, July). *IT forensics. Why post-mortem is dead. Cyber Security Summer School 2016: Digital Forensics, Technology and Law.* Tallinn University of Technology.

Fulgham, B., & Gouy, I. (2019, February). *Computer Language Benchmarks Game: C++ versus Rust.* https://benchmarksgame-team.pages.debian.net/ benchmarksgame/ faster/ rust.html.

Getreu, J. (2017). *Forensic-Tool Development with Rust* (Unpublished doctoral dissertation). Tallinn University of Technology, Tallinn.

Getreu, J. (2018). *Stringsext, a GNU Strings Alternative with Multi-Byte-Encoding Support.* Tallinn.

Harichandran, V. S., Walnycky, D., Baggili, I., & Breitinger, F. (2016). CuFA: A more formal definition for digital forensic artifacts. *Digital Investigation*, *18*, S125-S137.

The-Rust-Project-Developers. (2017). *The Rustonomicon.*

The-Rust-Team. (2019, January). *Rust Documentation.* https://doc.rust-lang.org/.

Zalewski, M. (2014, October). *PSA: Don't run 'strings' on untrusted files (CVE-2014-8485).*