ST. PÖLTEN UNIVERSITY
OF APPLIED SCIENCES

Informatik & Security

fh st.pölten

Parsing YAML is a minefield

An experimental research to determine differences in YAML 1.2 parser libraries

Master thesis

for attainment of the academic degree of

Diplom-Ingenieur

submitted by

Benjamin Oberdorfer BSc.
is231519

in the
University Course Information Security at St. Pölten University of Applied Sciences

Supervision
Advisor: Michael Kirchner, MSc.

ST. PÖLTEN UNIVERSITY
OF APPLIED SCIENCES

Informatik & Security

fh st.pölten

Declaration of Honour

First name, Surname: Benjamin Oberdorfer BSc.
Matriculation number: is231519
Type of Thesis: Master thesis
Title of the thesis: Parsing YAML is a minefield

I hereby affirm that

- I have written the work at hand on my own without help from others and I have used no other resources and tools than the ones acknowledged.
- I have complied with the Standards of good scientific practice in accordance with the St. Pölten UAS' Statute Part 05 when writing this work.
- I have neither published nor submitted the work at hand to another higher education institution for assessment or in any other form as examination work.

Regarding the use of generative artificial intelligence tools such as chatbots, image generators, programming applications, paraphrasing and translation tools, I declare that

☐ no generative artificial intelligence tools were used in the course of this work.

☒ I have used generative artificial intelligence tools to proof-read this work.

☒ I have used generative artificial intelligence tools to create parts of the content of this work. I certify that I have cited the original source of any generated content. The generative artificial intelligence tools that I used are acknowledged at the respective positions in the text.

Having read and understood the Statute 05 — Good Scientific Practice, I am aware of the consequences of a dishonest declaration.

Kurzfassung

Das benutzerfreundliche Datenserialisierungsformat YAML Ain't Markup Language (YAML) wird unter anderem in Cloud-Infrastrukturen, der Maschine-zu-Maschine-Kommunikation und Konfigurationsdateien häufig verwendet. Die Homogenität von YAML-Parser-Bibliotheken ist von essenzieller Bedeutung für die Gewährleistung von Sicherheit, Zuverlässigkeit und Interoperabilität. Die YAML 1.2.2 Spezifikation ist jedoch komplex, sodass eine vollständige Konformität schwer zu erreichen ist.

Um festzustellen, ob sich YAML 1.2-Parser-Bibliotheken voneinander unterscheiden, wurde eine YAML Test Suite entwickelt. Die YAML Test Suite umfasst 16 YAML 1.2 Parser-Bibliotheken und 245 YAML Testfalldateien, die durch Lesen der YAML 1.2.2 Spezifikation erstellt wurden. Die Ergebnisse werden dann automatisch ausgewertet, um festzustellen, ob die YAML Parser-Bibliotheken eine Datei korrekt parsen oder nicht. Zusätzlich werden Timeouts und Abstürze verfolgt.

Die Ergebnisse der YAML-Testsuite zeigen, dass zwar alle Parsing-Bibliotheken mehr als 50% der YAML Testfalldateien korrekt parsen können, aber keine YAML-Parsing-Bibliothek vollständig mit der YAML 1.2.2 Spezifikation kompatibel ist. Manchmal werden gültige YAML Testfalldateien abgelehnt, und in anderen Fällen werden ungültige YAML Testfalldateien akzeptiert. Es wurde auch festgestellt, dass YAML Parser-Bibliotheken bestimmte Funktionen, die in der YAML 1.2.2 Spezifikation festgelegt sind, einheitlich ignorieren. Ferner liefern keine zwei YAML Parser-Bibliotheken die gleichen Ergebnisse, wenn sie mit dem gleichen Satz von YAML Testfalldateien versorgt werden.

Außerdem wurden potenzielle Denial-of-Service-Schwachstellen (DoS) gefunden. Es wurde festgestellt, dass 14 von 16 YAML Parser-Bibliotheken während der Ausführung der YAML Test Suite mindestens einmal einen Timeout verursachen. Die Bibliotheken *HsYAML* und *YamlReference* wurden weiter analysiert und es wurde festgestellt, dass sie während dem Parsing einer YAML Testfalldatei, die eine immense Menge an Verschachtelungen enthält, einen nicht linearen Anstieg der Ausführungszeit aufweisen. Des Weiteren verursachen die YAML Parser-Bibliotheken *yaml-rust* und *YamlDotNet* einen Stapelüberlauf-Fehler. Obwohl beide Bibliotheken bei unterschiedlichen YAML Testfalldateien abstürzen, ist das zugrunde liegende Problem dasselbe. Bei einer tief verschachtelten Sequenz tritt ein Stapelüberlauf-Fehler auf. Anwendungen, die diese Bibliotheken zum Parsen von benutzerdefinierten YAML-Eingaben verwenden, können anfällig für DoS-Angriffe sein.

Zusammenfassend kann gesagt werden, dass die YAML 1.2.2 Spezifikation von den YAML Parser-Bibliotheken nur teilweise und uneinheitlich eingehalten wird. Dies untergräbt die Wirksamkeit der YAML 1.2.2 Spezifikation. Daher ist das Verhalten von YAML Parsing-Bibliotheken unvorhersehbar und inkompatibel, insbesondere weil keine zwei YAML Parsing-Bibliotheken

identische Ergebnisse liefern.[1]

[1]Deutsche Version der Kurzfassung übersetzt unter Zuhilfenahme von DeepL: https://deepl.com

Abstract

The user-friendly data serialization format known as YAML Ain't Markup Language (YAML) is widely used in cloud infrastructure, machine-to-machine communication, and configuration files, among other applications. Consistency among YAML parser libraries is essential for security, reliability, and interoperability. However, the YAML 1.2.2 specification is complex, making full compliance difficult to achieve.

To determine whether YAML 1.2 parser libraries differ from each other, a YAML Test Suite is developed. The YAML Test Suite includes 16 YAML 1.2 parser libraries and 245 YAML test case files that are created by reading the YAML 1.2.2 specification. The results are then automatically evaluated to determine whether the YAML parser libraries correctly parse a file or not. Additionally, timeouts and crashes are also tracked.

The results of the YAML Test Suite present that, while all parsing libraries can parse more than 50% of YAML test case files correctly, no YAML parsing library is completely compliant with the YAML 1.2.2 specification. Sometimes valid YAML test case files are rejected, and on other occasions invalid YAML test case files are accepted. It is also found that YAML parser libraries uniformly ignore certain features that are specified in the YAML 1.2.2 specification. In addition, no two YAML parser libraries produce the same results when supplied with the same set of YAML test case files.

In addition, potential Denial of Service (DoS) vulnerabilities are found. It is found that 14 out of 16 YAML parser libraries run into a timeout at least once during the run of the YAML Test Suite. The libraries *HsYAML* and *YamlReference* are further analyzed and are found to have a nonlinear increase in execution time during the parsing of a YAML test case file that contains an immense amount of nesting. Moreover, the YAML parser libraries *yaml-rust* and *YamlDotNet* produce a stack overflow error. While both libraries crash on different YAML test case files, the underlying issue is the same. When a sequence is deeply nested, a stack overflow error happens. Applications that use these libraries for parsing user-supplied YAML input can be susceptible to DoS attacks.

In conclusion, compliance with the YAML 1.2.2 specification is given only partially and inconsistently across YAML parser libraries. This undermines the effectiveness of the YAML 1.2.2 specification. Therefore, the behavior of YAML parsing libraries is unpredictable and incompatible, particularly because no two YAML parsing libraries generate identical results.

Acknowledgements

First and foremost, I would like to express my gratitude to Michael Kirchner, MSc, my supervisor. His involvement as a supervisor was instrumental in my success on numerous occasions, as it provided me with a clear understanding of the subject. This understanding has enabled me to remain motivated and perform at my highest level.

Additionally, I am grateful to my brother Lukas Oberdorfer, BA, and my colleague DI(FH) Bernhard Lorenz for reading my master's thesis and providing insightful feedback.

I also want to express my sincere gratitude to Manuel Elmer, who sold me his old laptop when mine was about to break. I would also like to extend my heartfelt gratitude to my sister, Miriam Oberdorfer, for her instrumental role in facilitating this.

Finally, I want to express my gratitude to my friends, family, and girlfriend for their patience with me, for supporting me on this journey, and for believing in me.

Contents

1. Introduction

Yaml Ain't Markup Language (YAML) is a user-friendly data serialization format that has various use cases.[1] YAML is employed in various real-world applications as a result of its readability and simplicity, such as the management of cloud infrastructures and DevOps.[2] Since it is used in multiple critical appliances, portability, reliability, and robustness must be a given in YAML parsing libraries. However, the YAML 1.2.2 specification specifies numerous intricate features that may prove difficult to implement in their entirety.[1] Therefore, it is difficult to achieve complete compliance in practice. Nevertheless, differences in the behavior of YAML parser libraries can negatively affect the interoperability, reproducibility, and security of applications that use YAML documents in machine-to-machine communication.

In terms of robustness, S. Rasheed, J. Dietrich, and A. Tahir find in their study multiple Denial of Service (DoS) vulnerabilities across various YAML parsing libraries.[3] Besides that, there exist Common Vulnerabilities and Exposures (CVE) for multiple YAML parsing libraries that are affecting the availability of programs using these parsing libraries such as *SnakeYAML*[2], *go-yaml*[3], and *yaml-cpp*[4] among others.[4], [5], [6] However, it is not possible to determine whether these vulnerabilities result from a feature defined by the YAML specification or from a mistake in the parser library's implementation.

Presently, there are numerous YAML parsing libraries available for various programming languages. Because the YAML 1.2 specification is almost 17 years old, many parsers now support features from the YAML 1.2 specification.[7] The YAML 1.2 specification has been revised twice in the past as a result of errors. The newest revision from 2021, the YAML 1.2.2 specification, intends to add clarity and correct errors. This research aims to determine whether YAML 1.2 parsing libraries are compliant with the YAML 1.2.2 specification. The YAML 1.2.2 specification is taken as a normative reference point, because earlier revisions contain errors.[1]

There exists research for different data serialization formats such as JavaScript Object Notation (JSON),[8], [9], [10] and Extensible Markup Language (XML).[11], [12] In contrast, YAML has received little scholarly attention outside the analysis of DoS vulnerabilities in YAML parsing libraries. Therefore, this research focuses on finding differences between YAML 1.2 libraries regarding the YAML 1.2.2 specification. This research takes the YAML 1.2.2 specification as a normative reference point and analyzes 16 YAML parser libraries for acceptance, rejection, and failure behavior. Specifically, the subsequent research question is intended to be addressed:

[2]CVE-2022-41854: https://nvd.nist.gov/vuln/detail/CVE-2022-41854
[3]CVE-2022-3064: https://nvd.nist.gov/vuln/detail/CVE-2022-3064
[4]CVE-2019-6292: https://nvd.nist.gov/vuln/detail/CVE-2019-6292

- How do YAML 1.2 parsers compare to the YAML 1.2.2 specification in terms of robustness, specification compliance, and parsing behavior?

This research question is further split into three sub-research questions:

- In accordance with the YAML 1.2.2 specification, how reliably do YAML 1.2 parsers accept or reject valid and invalid YAML inputs?
- What language rules from the YAML 1.2.2 specification are either inconsistently implemented or disregarded by YAML 1.2 parsers in practice?
- What are the security-relevant implications and robustness of the performance characteristics and failure modes of the parsing libraries?

1.1. Contribution

The contributions of this research are as follows:

- Python is used to construct a YAML Test Suite that enables the automatic evaluation of 16 distinct YAML parser libraries from various programming languages against a collection of YAML test case files.
- To systematically test for valid or invalid parsing behavior, 245 YAML test case files are generated by reading the YAML 1.2.2 specification.
- Evaluation of the YAML parser behavior to determine whether the parser library is in compliance with the YAML 1.2.2 specification and to identify inconsistencies.
- A discussion and analysis of the underlying causes of security-relevant failure modes, such as crashes, timeouts, and excessive execution time, are conducted.

1.2. Thesis outline

The thesis is structured as follows:

- **Introduction**: Section 1 introduces the topic, emphasizes the motivation, and sets the scope of this research.
- **Prerequisites**: Section 2 explains what data serialization formats are, gives an overview of what YAML is, and describes the history of YAML specifications.
- **Related Work**: Section 3 presents existing literature on parsing differences in JSON and XML and the current state of research in YAML parsing research.
- **Methodology**: Section 4 describes the methodology used in this research on a conceptual level.
- **Approach**: Section 5 elaborates on the experimental setup that is used in this research. Additionally, the limitations of this work are presented.
- **Results**: Section 6 presents the results that are acquired by the YAML Test Suite, highlights specific YAML test case files where YAML parsing libraries reveal non-compliance with the YAML 1.2.2 specification, and analyzes the causes of timeouts and crashes.
- **Discussion**: Section 7 discusses and interprets the results and answers the research questions.
- **Conclusion**: Section 8 summarizes the key findings of this research, final thoughts are expressed, and possibilities for future work are described.

2. Prerequisites

2.1. Data serialization formats

With data serialization formats, it is possible to transform data structures into an interoperable format that can be read by different systems, applications, or programming languages. Data structures can be serialized either into binary format or in a text-based format. While binary formats are more efficient, it is not user-friendly to interact with such formats. Simpler syntax is implemented in text-based formats, which enhances readability and facilitates user interaction. In practice, text-based data serialization formats can be used for configuration files (e.g., Docker), data exchange (e.g., web APIs), logging, data storage, etc.[13]

Nowadays, there are multiple text-based data serialization formats. Examples are JSON, XML, Comma-separated values (CSV), YAML, etc. Each of those has its own unique feature sets and syntax rules.[13]

2.2. Overview of YAML

YAML aims to be a user-friendly data serialization format. The syntax of a YAML file is designed to be simple and easy to read, and thus, a YAML file consists of Unicode printable characters. The structure of the data can be indicated via indentation, a list can be created via dashes, and to differentiate between key and value in a key/value pair, a colon is used.[1] While YAML implements multiple data structures, there are three main components:

- Scalars: Data such as strings, numbers, or booleans;
- Sequences: This component serves as an ordered list;
- Mappings: This component serves as a dictionary with key/value pairs.[1]

An example of a valid YAML file can be seen in Listing 1.

```yaml
- Test Scalar
- TestSequence:
    - Hello
    - World
- TestMapping: Hello
```

Listing 1: Example of a simple YAML file using scalars, sequences, and mappings.

YAML can be used for multiple purposes, for example, configuration files (e.g., Docker), data sharing across programming languages, interprocess messaging, logging, etc.[1] Since most of the YAML files are automatically parsed, consistency and accuracy to the specification are prerequisites for the YAML parser library to ensure reliability and interoperability.

2.2.1. Interpretation of the YAML specification

This Subsection describes core concepts that are required to understand the YAML 1.2.2 specification.

2.2.1.1. RFC 2119 key words

The YAML 1.2.2 specification uses the in RFC 2119 defined keywords “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY” and “OPTIONAL” throughout the specification.[1], [14]

When the keywords “MUST”, “REQUIRED”, and “SHALL” are used in the specification, it means that this feature is an absolute requirement and must be implemented. In contrast, the keywords “MUST NOT” and “SHALL NOT” indicate that something should not be implemented. “SHOULD” and “RECOMMENDED” communicate that a feature can be disregarded if there are good reasons to not implement it. But in not implementing the feature, the consequences must be understood. The keywords “SHOULD NOT” and “NOT RECOMMENDED” are the counterparts and indicate that something can be implemented if there are good reasons to do so. Lastly, the “MAY” and “OPTIONAL” keywords describe features that are optional to implement. However, interoperability needs to be a given between different parsing libraries, even if it means that certain functionalities are ignored.[14]

2.2.1.2. Syntax error reporting

The YAML 1.2.2 specification specifies that malformed syntax should be rejected by YAML parsing libraries. Nevertheless, the YAML parser developers are responsible for determining whether a syntax error results in the parsing process being aborted or if the error is disregarded and the remaining contents of the YAML file are parsed. If a YAML parsing library can recover from syntax errors, it must be reporting it. However, it is not specified how the reporting feature must be implemented.[1]

2.3. Evolution of the YAML specification

There are various versions of the YAML specification.[7] The YAML 1.0 specification was released in January 2004.[15] In January 2005, the YAML 1.1 specification was released, which added new functionalities, for example, allowing the keywords `yes` and `no` to be interpreted as booleans.[16] Four years later, in July 2009, the YAML 1.2.0 specification was introduced, and the goal was to make YAML a superset of JSON. Additionally, there were minor changes, such as removing the feature that `yes` and `no` can be interpreted as booleans and interpreting empty values as `null` values. The YAML 1.2.0 specification still intends to be compatible with YAML 1.1. The YAML 1.2.1 specification was released later that year, in October 2009, with the intent to fix errors from the YAML 1.2.0 specification.[17] The YAML 1.2.2 specification was released in October 2021, twelve years after the release of the YAML 1.2.1 specification. Instead of adding any new lexical or syntactical features, this specification focused on revising mistakes to make things clearer.[1]

Most parser libraries claim to be compliant with YAML 1.2, but the revision is not indicated. [7] Due to errors in the YAML 1.2.0 specification, inconsistencies, and ambiguities can arise, which can lead to differences between parsing libraries.[1] Thus, the YAML 1.2.2 specification is the reference point for this research.

3. Related Work

Compared to other serialization formats, YAML has received relatively limited academic attention regarding benchmarking different YAML parser libraries. This section presents research on related topics in various data serialization formats and offers a glimpse into the research conducted for YAML.

3.1. Research on differences in JSON parsing

Similar research exists for the JSON format. In 2016 the paper “Parsing JSON is a Minefield” by N. Seriot analyzes different JSON parser libraries for multiple programming languages. The author proceeds to write JSON test case files and decides based upon RFC 8259, whether a JSON test case file should be accepted or rejected by the JSON parsing library. To automatically test all parsers, N. Seriot developed the JSON Test Suite. JSON libraries return the exit code zero on successful parsing and one if parsing failed. Additionally, a five-second timeout is configured for parsing. N. Seriot highlights in his research that no two JSON parser libraries can produce the same results when parsing the same set of JSON test case files. Furthermore, bugs, crashes, and potential DoS attack vectors are found during this research.[8]

H. K. Dhalla assesses the performance by examining resource consumption and parsing speed of five different JSON parsers. The native JSON parsers of the programming languages Java, Python, MS.NET Core, JavaScript, and PHP are being put under test. The JSON test case files are automatically generated and have a fixed nesting depth of 20 nesting layers. With each JSON test case file, the number of key/value pairs in every nesting layer increases by ten. Ultimately, ten JSON test case files are generated. H. K. Dhalla concludes that the native JSON parser of JavaScript is the most efficient of the other native parsers under test.[18]

The study “The Behavioral Diversity of Java JSON Libraries”[9] from N. Harrand, T. Durieux, D. Broman, and B. Baudry is another example of a JSON Test Suite that tests 20 JSON parsing libraries and includes 473 JSON test case files. The difference between the work of 2016 and this study is that this study additionally observes the input and output of JSON parsing libraries.[9]

In “Cross-Language Differential Testing of JSON Parsers”[10] by J. Möller, F. Weißberg, L. Pirch, T. Eisenhofer, and K. Rieck, a framework is developed to automate the process of determining semantic discrepancies in JSON parsing libraries. In total, 22 JSON parsing libraries are tested, and the approach is different from the one used in “Parsing JSON is a Minefield”. While N. Seriot focused on finding discrepancies between parsing libraries by reading specifications and crafting JSON test case files manually,[8] whereas the work of J. Möller, F. Weißberg, L. Pirch, T. Eisenhofer, and K. Rieck rather focuses on finding differences in JSON parsing libraries regardless of what the specification states.[10]

3.2. Research on differences in XML parsing

Additionally, comparable research exists for XML. Back in 2007, a comparative study and benchmarking of XML parser libraries was released by S. C. Haw and G. S. V. R. K. Rao in the study “A Comparative Study and Benchmarking on XML Parsers”.[11] The study focuses on benchmarking three different XML parser libraries by supplying XML files with different sizes and measuring the performance and execution time of each parsing library. The parser libraries xParse, Xerces, and .NET parser are being tested in this research. According to this study, Xerces performs the best, but xParse is more effective at parsing large data sets.[11]

The study “A comparative study on performance of XML parser APIs (DOM and SAX) in parsing efficiency”[12] by D. Wu, K. T. Chau, J. Wang, and C. Pan from 2019 focuses on benchmarking the performance of the Document Object Model (DOM) and the Simple API for XML (SAX) parsing Application Programming Interfaces (APIs) for XML. The study’s main emphasis is to determine whether DOM parsing is faster than SAX in terms of execution time, memory consumption, and adjustability. The result is that parsing with the DOM API is consuming more memory, has a higher execution time, but offers a higher adjustability.[12]

3.3. Current state of YAML parsing research

While no academic research is found to examine multiple parser libraries and map their compliance to the YAML 1.2.2 specification, a GitHub repository exists from The YAML Project called “YAML Test Suite”[19] which is a collection of YAML test case files that is actively maintained. The primary goal of this repository is to ensure interoperability and correctness of YAML parsing libraries.[19] Furthermore, the “YAML Test Matrix”[20] repository by T. Müller uses the YAML test case files from the “YAML Test Suite” repository, runs the parser libraries in docker containers, and visualizes the results in a matrix.[20] The currently supported YAML parsing libraries for the “YAML Test Matrix” can be seen in Table 1.)

YAML parser library	Programming language	URL
libfyaml	C	https://github.com/pantoniou/libfyaml
libyaml	C	https://github.com/yaml/libyaml
rapidyaml	C++	https://github.com/biojppm/rapidyaml
yaml-cpp	C++	https://github.com/jbeder/yaml-cpp
YamlDotNet	C#	https://github.com/aaubry/YamlDotNet
yaml	JavaScript	https://github.com/eemeli/yaml
js-yaml	JavaScript	https://github.com/nodeca/js-yaml
NimYAML	Nim	https://github.com/flyx/NimYAML
yaml-rust	Rust	https://github.com/chyh1990/yaml-rust
go-yaml	Go	https://github.com/go-yaml/yaml
HsYAML	Haskell	https://github.com/haskell-hvr/HsYAMLs
YamlReference	Haskell	https://github.com/orenbenkiki/yamlreference
lyaml	Lua	https://github.com/gvvaughan/lyaml

YAML::PP	Perl	https://github.com/perlpunk/YAML-PP-p5
YAML::Parser	Perl	https://metacpan.org/release/YAML-Parser
YAML::Syck	Perl	https://metacpan.org/release/YAML-Syck
YAML::Tiny	Perl	https://metacpan.org/release/YAML-Tiny
YAML::XS	Perl	https://metacpan.org/release/YAML-LibYAML
YAML.pm	Perl	https://metacpan.org/release/YAML
PyYAML	Python	https://github.com/yaml/pyyaml
ruamel.yaml	Python	https://sourceforge.net/projects/ruamel-yaml/
Psych	Ruby	https://github.com/ruby/psych

Table 1: YAML parser libraries used in the “YAML Test Matrix”.[21]

However, the most recent commit of the “YAML Test Matrix”[20] repository dates back four years,[20] and the latest available matrix dates from 17 January 2022.[21] Therefore, the results do not reflect the current state of YAML parsing libraries, especially because some YAML 1.2 parsing libraries are not included in the matrix (e.g., *SnakeYAML Engine*, *eo-yaml*, *ocaml-yaml*, etc.).[20], [21] Furthermore, the “YAML Test Suite”[19] and “YAML Test Matrix”[21] primarily focus on functionality testing and do not explicitly analyze potential security risks that arise from differences in YAML parsing libraries.[19], [20]

Nevertheless, there exists research on security risks associated with YAML parsing libraries. The study “Laughter in the Wild: A Study Into DoS Vulnerabilities in YAML Libraries”[3] by S. Rasheed, J. Dietrich, and A. Tahir tests for DoS vulnerabilities in 14 different YAML parsing libraries by using features defined .[3] Table 2 shows the parsing libraries that are analyzed in this study.

YAML parser library	**Programming language**
js-yaml	JavaScript
php-yaml	PHP
Symfony Yaml	PHP
PyYAML	Python
ruamel.yaml	Python
yaml	Dart
YamlDotNet	C#
YAML::Syck	Perl
YAML::XS	Perl
YAML	Perl
Psych	Ruby

SnakeYAML	Java
yaml-rust	Rust
Swift	YamlSwift

Table 2: YAML parser libraries analyzed in “Laughter in the Wild: A Study Into DoS Vulnerabilities in YAML Libraries”.[3]

The researchers found memory exhaustion flaws that can lead to DoS in multiple libraries, such as *js-yaml* (JavaScript), *php-yaml* (PHP), *yaml* (Dart), *YamlDotNet* (C#), *Psych* (Ruby), *SnakeYAML* (Java), and *yaml-rust* (Rust). The authors identified the main cause for these vulnerabilities to be the anchor and alias feature.[3] The anchor and alias feature makes it possible to first assign a value with an anchor that can later be reused with an alias.[1]

Nevertheless, this work is published in 2019. Since then, many YAML parser libraries examined in that research in this research have been updated with bug fixes and other changes. In addition, the YAML parsing libraries are not evaluated for compliance with the YAML specification. Instead, this work explicitly focuses on finding DoS vulnerabilities in YAML parsing libraries.[3]

S. Rasheed, J. Dietrich, and A. Tahir further analyze multiple parsers in their study “Caught in the Web: DoS Vulnerabilities in Parsers for Structured Data”.[22] Parsers for formats like JSON, Scalable Vector Graphics (SVG), Portable Document Format (PDF), YAML, etc. are being investigated for vulnerabilities in this study. During the experiment, the authors find a vulnerability in the YAML parsing library *SnakeYAML*.[22] The YAML input that leads to this vulnerability consists of merging keys, which are an optional YAML feature.[22], [23] Furthermore, nested lists and aliases are utilized, which ultimately leads to a stack overflow in the *SnakeYAML* parsing library.[22]

There is extensive academic research on JSON and XML parsing library differences and security analysis on YAML. Nevertheless, no scholarly study has been conducted that assesses YAML parsing libraries for conformance to the YAML 1.2.2 specification and examines any possible security risks brought on by differences in YAML parsing libraries.

4. Methodology

Experimental research is being conducted to ascertain the distinctions between YAML parser libraries to address the research questions posed in Section 1. A Python script is developed to evaluate a collection of YAML test case files against a list of YAML 1.2 parser libraries.

First, YAML parser libraries that are specified on the official YAML website[5] and are listed to support YAML 1.2 are selected for this research (see Table 3).

YAML parser library	Version	Programming language	URL
yaml-rust	0.4.5	Rust	https://github.com/chyh1990/yaml-rust
ruamel.yaml	0.18.15	Python	https://sourceforge.net/projects/ruamel-yaml/
libfyaml	0.5.7.34	C	https://github.com/pantoniou/libfyaml
yaml	2.8.0	JavaScript	https://github.com/eemeli/yaml
js-yaml	4.1.0	JavaScript	https://github.com/nodeca/js-yaml
go-yaml	1.18.0	Golang	https://github.com/goccy/go-yaml
The Yaml Component	7.3.1	PHP	https://github.com/symfony/symfony/blob/7.3/src/Symfony/Component/Yaml/Yaml.php
YAML::PP	0.39.0	Perl	https://github.com/perlpunk/YAML-PP-p5
NimYAML	2.2.0	Nim	https://github.com/flyx/NimYAML
ocaml-yaml	3.2.0	OCaml	https://github.com/avsm/ocaml-yaml
yaml-cpp	0.8.0	C++	https://github.com/jbeder/yaml-cpp
YamlDotNet	16.3.0	C#	https://github.com/aaubry/YamlDotNet

[5]Official YAML website: https://yaml.org

HsYAML	0.2.1.5	Haskell	https://github.com/haskell-hvr/HsYAMLs
YamlReference	0.10.0	Haskell	https://github.com/orenbenkiki/yamlreference
SnakeYAML Engine	2.10	Java	https://bitbucket.org/snakeyaml/snakeyaml-engine
eo-yaml	8.0.6	Java	https://github.com/decorators-squad/eo-yaml

Table 3: Parser libraries that are chosen to determine differences in YAML parsing.

A standalone application is created for each YAML parser library. Each standalone application accepts the absolute file path to a YAML file as a command line argument that is supposed to be parsed with the corresponding YAML parser library. The call to the parsing libraries is encapsulated in a try-catch statement in most YAML parsing libraries. In programming languages that do not support try-catch statements, an if-condition checks whether parsing is successful or not. On successful parsing of the YAML file, the standalone applications return the exit code zero; on failure, the exit code one is returned. Different exit codes imply that the YAML parsing library crashes during parsing.

Moreover, 245 YAML test case files are created that check for specified functionality but also for potential edge cases. The YAML test case files are created by reading the YAML 1.2.2 specification.[1] Each test case file's filename begins with `valid` or `invalid`, indicating whether the library being tested should successfully parse the file in compliance with the YAML 1.2.2 specification or whether an error should be raised.

The Python script `run_yaml_parser_tests.py` is the core of the YAML Test Suite. It contains a predefined list with the absolute filepath of the standalone applications and the specific execution instructions. During the runtime of the script, each YAML test case file is passed to these applications and then parsed with the corresponding YAML 1.2 library. The Python script captures the exit codes that are returned by the standalone applications. Using exit codes, the Python script can then determine whether parsing the file is successful, parsing fails, or the parser crashes. Additionally, a timeout of three seconds for each execution of a standalone application is set. Since the YAML 1.2.2 specification allows libraries to ignore syntax errors by reporting the error to the user,[1] the Python script stores the contents of the Standard output (stdout) and Standard error (stderr) of the standalone applications in a logfile. The log is then used to further analyze whether the library recognized the syntax error or if the input is parsed.

The results are then stored in a Hypertext Markup Language (HTML) file for visualization and are automatically converted to a Typst table with the help of another Python script called `typst_tablemaker.py`. After a library processes a YAML test case file, the Python script then determines if:

- The result is expected according to the YAML 1.2.2 specification;
- The parser should have been able to parse the test case file but fails;
- The parser should not have been able to parse the test case file but succeeds;
- The application times out during testing;
- The application crashes during testing.

An approximate visual representation of the YAML Test Suite is depicted in Figure 1. The numbers indicate the order in which the Python script will execute these tasks. Further details are specified in Section 5.

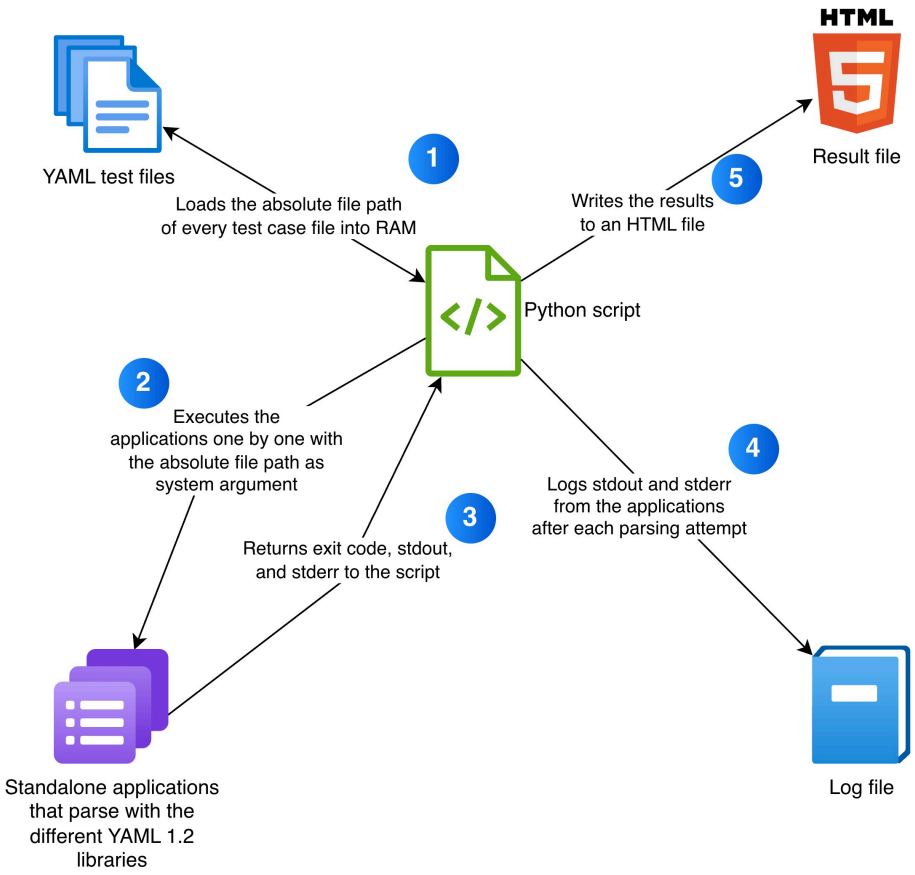

Figure 1: A simplified outline of the YAML Test Suite.

5. Approach

This Section provides an overview of the test setup used during this research. First, the Virtual Machine (VM) setup gets explained, which explains the VM configuration used and what is needed to develop the YAML Test Suite.

Next, the experimental prototype is elaborated. This Subsection focuses on the workflow of the YAML Test Suite and describes all the parts that it consists of to give an understanding of how it works.

The last Subsection describes the limitations of this academic research. It is divided into three more Subsections to give an explanation of the methodological limitations, scope limitations, and specification-related limitations.

5.1. Setup

The experiment is run inside an Ubuntu 24.04.1 VM with the Linux kernel version 6.14.0-32-generic. This VM has two processor cores, 16 Gigabyte (GB) of Random-Access Memory (RAM), and 60 GB of hard disk space. The codebase is developed using Visual Studio Code by remotely connecting via Secure Shell (SSH) to the VM.

5.1.1. Development setup

Because YAML 1.2 libraries are written in various programming languages, using them requires the use of different compilers, build automation tools, software development kits, and interpreters. A listing of all compilers, build automation tools, software development kits, and interpreters is depicted in Table 4.

Library	**Compiler/Build automation tool/Software development kit/Interpreter**	**Version**
yaml-rust	rustc	1.86.0
ruamel.yaml	Python	3.12.3
libfyaml	gcc	13.3.0
yaml	Node	18.19.1
js-yaml	Node	18.19.1
go-yaml	gc	1.24.5
The Yaml Component	PHP	8.3.6
YAML::PP	Perl	5.38.2
NimYAML	Nim Compiler	2.2.4
ocaml-yaml	dune	3.19.1

yaml-cpp	g++	13.3.0
YamlDotNet	dotnet	8.0.121
HsYAML	stack	3.7.1
YamlReference	stack	3.7.1
SnakeYAML Engine	Maven	3.8.7
eo-yaml	Maven	3.8.7

Table 4: List of compilers, build automation tools, software development kits, and interpreters that are used.

5.2. The YAML Test Suite

The YAML Test Suite is a framework for experimenting with different YAML 1.2 parsers. The core is a Python script that can be extended with YAML test case files to test parser libraries for specified functionality.

The YAML Test Suite can be divided into 3 components:

- The Python script. This script is the central unit of this research project and handles the execution of standalone applications and stores the results in logs and in an HTML file.
- The YAML test case files, which are YAML files that contain specific YAML syntax.
- The standalone applications. Each application uses one of the YAML libraries and tries to parse the file that gets passed over the command line argument.

A sequence diagram shows the relationships between the different parts of the YAML Test Suite and the correct order in which it gets executed. The sequence diagram is displayed in Figure 2.

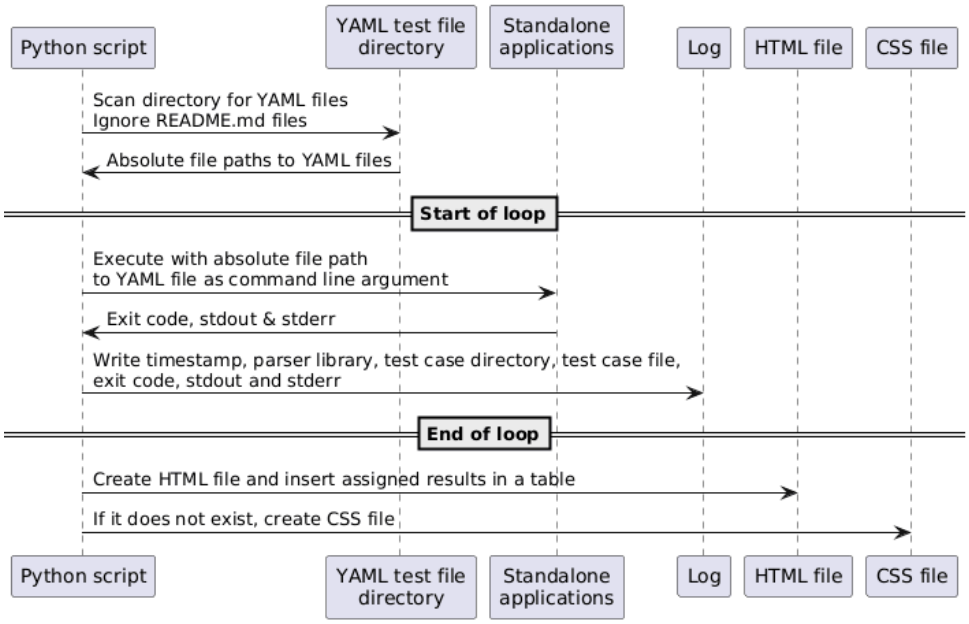

Figure 2: Relationships between the different components.

The technical details of each component are elaborated in this Section.

5.2.1. Python script

A Python script with the name `run_yaml_parser_tests.py` acts as the core. Listing the files inside the test directory, executing standalone applications, storing stdout and stderr of each application, interpreting the exit codes, and visualizing the results in an HTML file are the responsibilities of the Python script.

5.2.1.1. Scanning the test case directory

Before the YAML parsing tests start, the `yaml_testcases` directory gets recursively scanned to find YAML test files. Each file (except for `README.md` files) is then stored inside the dictionary `FILE_STRUCTURE` with the following format: `<absolute_file_path_to_directory>: [<test_file_1>, <test_file2>]`. An example can be seen in Listing 2.

```Python
{
    '/mnt/disk/YAMLTestSuite/yaml_testcases/ParsingYAMLIsAMinefield':
    ['invalid_:_indicator_after_1024_unicode_characters.yml',
    'invalid_:_indicator_after_1024_unicode_characters_inside_flow_mapping.yml',
    'invalid_:_indicator_after_1024_unicode_characters_inside_flow_sequence.yml',
    '...']
}
```

Listing 2: Example of how file paths are stored inside the `FILE_STRUCTURE` dictionary.

Since the `yaml_testcases` directory is being scanned recursively, it is possible to extend test cases and organize them in different subdirectories. The Python script also allows adding README files. This can be used to, e.g., describe what is tested with the current subset of YAML files.

5.2.1.2. Running YAML parser tests

After `FILE_STRUCTURE` is populated, the script jumps into the `run_tests` function. A triple-nested for loop is used that runs tests using all standalone applications and all test case files in the `yaml_testcases` directory. For this function, another global variable with the name `ALL_PARSERS` is used that stores information about each standalone application (the name of the parser utilized and source URL) and how to execute the standalone application that wraps the library. Listing 3 provides one example.

```Python
{
    "yaml-rust": {
        "source": "https://github.com/chyh1990/yaml-rust",
        "command": [
            os.path.join(
                PARSER_DIR,
                "yaml_rust_test",
                "target",
                "debug",
                "yaml_rust_test"
            )],
    },
    "..."
```

```
14 }
```

Listing 3: Excerpt from the Python script where an example entry of the `ALL_PARSERS` variable is shown. The variable `PARSER_DIR` stores the absolute path to the directory where all standalone applications reside.

Before the commencement of the nested loops, the log file is initiated by recording the initial information that a new test run has commenced. During the iteration of the loops, a text output is written, storing the stdout and stderr of the standalone applications after each execution. The main purpose of this file is to aid the understanding of each result. Hence, this log file is verbose and captures every execution, except for executions leading to timeouts. Listing 4 presents the implementation of the logging functionality.

```python
def run_tests(parsers=None):
[...]
    # Log file gets opened before the nested loops start
    with open(LOGFILE, "a", encoding="utf-8") as log_file:
        log_file.write(
            f"New Test Run started at {datetime.now()}\n"
            f"-------------------------------\n"
        )
        [...]
        for parser_name, parser_info in parsers.items():
            [...]
            for directory, testcases in FILE_STRUCTURE.items():
            [...]
                    for testcase in testcases:
                        [...]
                        # Log the exact timestamp of execution, which parser library was used, which directory the YAML test case is in, the test case file, the exit code and stdout and stderr of the application
                        log_file.write(
                            f"Timestamp: {datetime.now()}, Parser: {parser_name}, Directory: {directory}, File: {testcase}, Error Code: {ret_code}\n"
                            f"Output to stderr: {cmd_output.stderr}\n"
                            f"Output to stdout: {cmd_output.stdout}\n"
                            f"-------------------------------\n"
                        )
                        [...]
```

Listing 4: Capturing stdout and stderr to store it inside a logfile.

Next, each entry of `ALL_PARSERS` is being accessed via a for loop, storing the command in a temporary variable and printing to stdout which parser library is currently under test. Thereafter, another for loop starts that iterates the items of the `FILE_STRUCTURE` dictionary. Lastly, since the test case filenames are stored inside a list in the `FILE_STRUCTURE` dictionary, another for loop iterating this list is used. The corresponding Python code can be seen in Listing 5.

```Python
def run_tests(parsers=None):
    [...]
    if not parsers:
            parsers = ALL_PARSERS
    [...]
        # Iterate over each parser with the exact command on how to execute the
        standalone application wrapping this parser
        for parser_name, parser_info in parsers.items():
            command = parser_info["command"]
            print(f"Running tests with {parser_name}...")
            # Iterate over each test case file
            for directory, testcases in FILE_STRUCTURE.items():
                tmp_result_dict = {}
                # Iterate the list of YAML test cases
                for testcase in testcases:
                [...]
```

Listing 5: Python code of the triple-nested for loop. First, the `ALL_PARSERS` dictionary is iterated, then the `FILE_STRUCTURE` dictionary, and lastly the list containing the filenames of the YAML test case files.

With all the necessary information gathered, the standalone applications can be executed. Using the `subprocess` library, the standalone applications are executed with the absolute file path to the YAML test file as a command line argument. Additionally, a timeout of three seconds is specified for execution. In other terms, if a standalone application needs longer than three seconds to finish execution, the execution is interrupted. The exit codes then get interpreted by the function `assign_results` (see Section 5.2.1.3). Additionally, the interpreted results are stored inside the `ALL_PARSERS` dictionary after all test case files are parsed from a directory. The execution of standalone applications and the storage of results are illustrated in Listing 6.

```Python
def run_tests(parsers=None):
[...]
        for parser_name, parser_info in parsers.items():
            [...]
            for directory, testcases in FILE_STRUCTURE.items():
                [...]
                for testcase in testcases:
                    full_path_to_yaml = os.path.join(
                        directory, testcase)
                    # Prepare the command for the standalone application
                    full_command = command + [full_path_to_yaml]
                    # Try executing the application
                    try:
                        cmd_output = subprocess.run(
                            full_command, capture_output=True, text=True,
                            timeout=3, check=False)
                    # In case three second timeout is reached
                    except subprocess.TimeoutExpired:
```

```
                        tmp_result_dict.update({
                            testcase: "TIMEOUT"
                        })
                        [...]
                        continue
                    # Store the return code of the application
                    ret_code = cmd_output.returncode
                    [...]
                    # Interpret the exit code and temporarily store the result
                    tmp_result_dict.update(
                        assign_results(testcase, ret_code))
                # Add results to the ALL_PARSERS dictionary
                parsers[parser_name].update({
                    directory: tmp_result_dict
                })
```

Listing 6: Executing standalone applications and storing the evaluated results in the `ALL_PARSER` dictionary.

5.2.1.3. Interpreting results

In accordance with the YAML 1.2.2 specification, the filename of all YAML test case files contains a prefix that denotes whether the YAML parsing library should successfully parse the file or fail to parse it. These filename prefixes are either `valid_` or `invalid_`. Using exit codes and those prefixes, the `assign_results` function interprets the results and returns a dictionary. The logic of this function can be viewed in Listing 7.

```python
def assign_results(yaml_file, code):
    [...]
    # Test case file is valid and is parsed succesfully
    if code == 0 and yaml_file.startswith("valid_"):
        return ({
            yaml_file: "PASSED"
        })
    # Test case file is invalid and is not parsed successfully
    if code == 1 and yaml_file.startswith("invalid_"):
        return ({
            yaml_file: "PASSED"
        })
    # Test case file is invalid, but parsed successfully
    if code == 0 and yaml_file.startswith("invalid_"):
        return ({
            yaml_file: "SHOULD_HAVE_FAILED"
        })
    # Test case file is valid, but not parsed successfully
    if code == 1 and yaml_file.startswith("valid_"):
        return ({
            yaml_file: "SHOULD_HAVE_PASSED"
```

```
22          })
23      # File name prefix is not set
24      if not yaml_file.startswith("valid_") and not
        yaml_file.startswith("invalid_"):
25          return ({
26              yaml_file: "UNKNOWN"
27          })
28      # If the standalone application returns a different exit code than zero and
        one, the application crashes
29      return ({
30          yaml_file: "CRASHED"
31      })
```

Listing 7: The `assign_results` function.

5.2.1.4. Visualizing the results with HTML

HTML is used to visualize the results. The `write_to_html` function inside the Python script builds an HTML file that takes the results of the `ALL_PARSERS` dictionary and puts them into a HTML table. Additionally, a Cascading Style Sheets (CSS) file is created if it does not exist. The characters <, >, &, ' and " in the YAML test case file names are escaped with HTML entities. An example of the HTML result file can be seen in Figure 3.

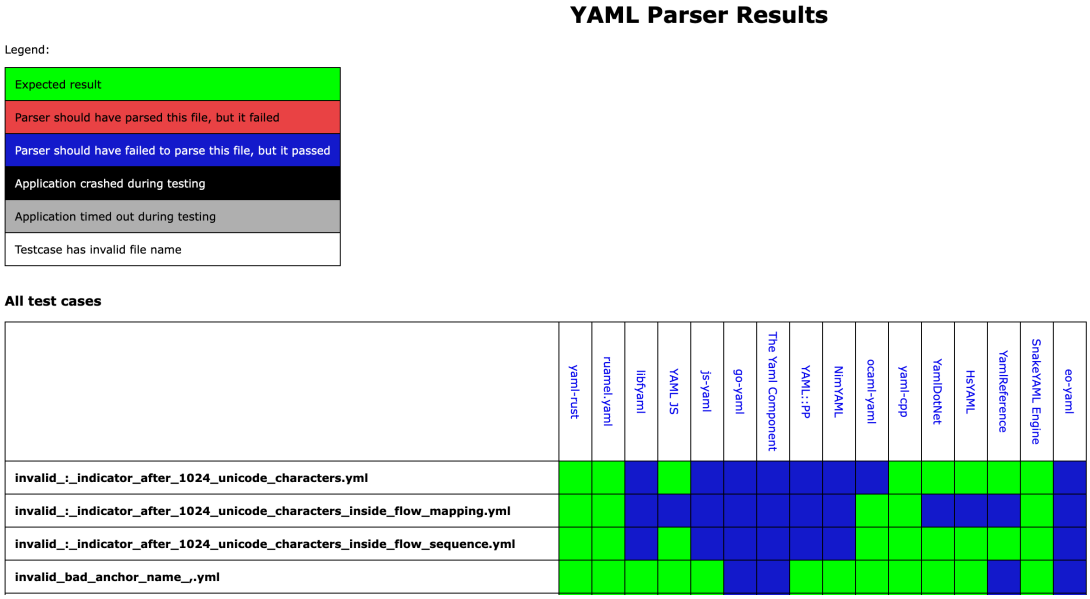

Figure 3: An HTML result file is generated by the Python script.

5.2.1.5. Optional command-line parameters

Optional command-line parameters can be used to gain more control over the Python script. All currently supported command-line parameters are listed in Table 5.

Long form	Short	Description
`--single`	`-s`	Runs tests only for one standalone application by specifying the parser library that should be tested.

`--directory`	`-dir`	Specify which YAML test case directory should be used. Defaults to the `yaml_testcase` directory.
`--output`	`-o`	Specify the filename of the result HTML.
`--dump-to-json`	Not Available (N/A)	Additionally creates a JSON dump with the results.

Table 5: List of supported optional command-line parameters.

5.2.2. YAML test case files

YAML test case files are YAML files that test for specific syntax that is or is not specified in YAML 1.2.2. By reading the YAML 1.2.2 specification, these test case files are crafted.

Every test case file has a prefix in its filename. The prefix `valid_` stands for test case files that should be parsed successfully, while the prefix `invalid_` stands for test case files that should make parsers fail.

After the prefix, the filenames describe what specific syntax is being tested with this test case file. An example of a test case filename can be seen in Figure 4.

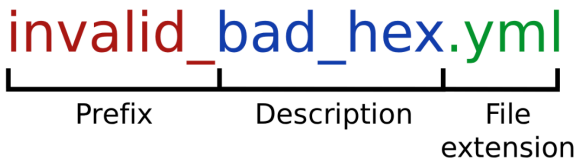

Figure 4: Nomenclature of the YAML test case files.

For example, YAML 1.2.2 specifies the following: “If the “?” indicator is omitted, parsing needs to see past the implicit key to recognize it as such. To limit the amount of lookahead required, the “:” indicator must appear at most 1024 Unicode characters beyond the start of the key.”[1] Based on this information, a YAML test case file is created with the name `invalid_:_indicator_after_1024_unicode_characters_inside_flow_mapping.yml`, which contains 1028 characters before the colon appears. A shortened version can be seen in Listing 8.

```yaml
{
  AAAAAAAAAAAAA[...]: Test
}
```

Listing 8: YAML test case file with a 1028 character long key value that is invalid as per the YAML 1.2.2 specification.

A YAML test case file that includes valid syntax as per the YAML 1.2.2 specification is `valid_yaml_1_2_directive.yml`. YAML 1.2.2 specifies the following: “A version 1.2 YAML processor must accept documents with an explicit “%YAML 1.2” directive, as well as documents lacking a “YAML” directive.”[1] A directive in a YAML file can give instructions to the YAML processor, which in this case means to assert the YAML version in which the YAML file shall be parsed.[1] With that information, the YAML test case file `valid_yaml_1_2_directive.yml` is constructed (see Listing 9).

```yaml
%YAML 1.2
---
```

Listing 9: YAML test case file `valid_yaml_1_2_directive.yml_showcasing` containing the YAML directive to specify the version in which the YAML file shall be parsed.

In addition, the YAML Test Suite can be extended and grouped by adding new directories to the `yaml_testcases` directory. Currently there are the `ParsingYAMLIsAMinefield` and `functionality_testing` directories. The `ParsingYAMLIsAMinefield` contains the handcrafted YAML test cases that are created during this research, and the `functionality_testing` directory contains a YAML file that includes 211 grammar rules from the YAML 1.2.2 specification[24] and a YAML file that is supposed to be rejected by every YAML parsing library.[1] This directory can optionally be used to extend the YAML Test Suite.

5.2.3. Standalone Applications

A standalone application is a program that wraps a YAML parser library. Its main functionality is to parse the file that it receives via command line argument. The parsing call is mostly wrapped in a try-catch statement, which, on successful parsing, returns the exit code zero, and if an error occurs during parsing, returns the exit code one. For instance, Listing 10 illustrates this implementation by displaying the standalone application for *ruamel.yaml*.

```python
#!/usr/bin/env python3

from ruamel.yaml import YAML
from ruamel.yaml.parser import ParserError
import sys


def parse(file):
    yaml = YAML()

    with open(file, 'r') as f:
        # Tries parsing the given YAML file
        try:
            yaml.load(f)
        except ParserError as pe:
            # Sends error message to stderr, which will be shown in the logs
            print(f"ParserError: {pe}", file=sys.stderr)
            # On failure return exit code 1
            sys.exit(1)
        except Exception as e:
            # Sends error message to stderr, which will be shown in the logs
            print(f"Error: {e}", file=sys.stderr)
            # On failure return exit code 1
            sys.exit(1)


if __name__ == "__main__":
    # Parses the YAML file that is received via command line argument
    parse(sys.argv[1])
```

```
    # On success return exit code 0
    sys.exit(0)
```

Listing 10: The standalone application for *ruamel.yaml*.

Some standalone applications are written in programming languages that do not support try-catch statements (e.g., standalone applications written in Golang). This specific problem is solved by using if-conditions. The standalone application for *go-yaml* can be seen in Listing 11, which implements such a method.

```go
package main

import (
    "os"

    "github.com/goccy/go-yaml"
)

func main() {
    yaml_content, err := os.ReadFile(os.Args[1])
    if err != nil {
        os.Exit(2)
    }

    var iface interface{}
    // On error during parsing, return exit code 1
    if err := yaml.Unmarshal(yaml_content, &iface); err != nil {
        os.Exit(1)
    }
    // On success during parsing, return exit code 0
    os.Exit(0)
}
```

Listing 11: The standalone application for *go-yaml* using if-conditions.

It should be further noted that some standalone applications are developed with the help of Artificial Intelligence (AI). For some standalone applications, ChatGPT 4o is chosen, while for others, ChatGPT 5. The following template is used to generate standalone applications: “Please write me a <insert programming language name> program that gets a filepath passed over arguments. The file should then be parsed with <insert parser library>. I dont need the result printed to stdout. If the program was able to parse it, it should exit with error code 0. if not, it should exit with error code 1.”

5.2.3.1. Verifying functionality of standalone applications

The YAML Project offers a repository with a YAML grammar file.[7] This grammar file, which is called `yaml-spec-1.2.yaml`, tries to encapsulate each grammar rule from the YAML 1.2.2 specification inside one YAML file.[24] This grammar file is used to test whether the standalone applications can parse and recognize valid YAML patterns.

Additionally, a YAML file is used that should cause every standalone application to fail parsing. For this test, a YAML file is created by reading the YAML 1.2.2 specification. The error, which the file tries to trigger, is that the Flow Node is not closed correctly.[1] Its name is `expected_failure.yml` because it is meant to fail when it is parsed. The YAML file can be seen in Listing 12.

```yaml
1 [hello, [, world]
```

Listing 12: YAML file that should cause standalone applications to fail parsing.

A new directory is being created in the `yaml_testcases` directory with the name `verify_standalone_application_functionality`. The `yaml-spec-1.2.yaml` and `expected-failure.yml` files are being moved into this directory. Furthermore, the filename of the `yaml-spec-1.2.yaml` file is prefixed with `valid_` because each grammar rule tested inside this file is valid according to the YAML 1.2 specification.[24] The `expected-failure.yml` file is prefixed with `invalid_`, since this YAML file contains invalid YAML syntax.[1] The `run_yaml_parser_tests.py` Python script can then be started with the `--dir` option, which specifies the `verify_standalone_application_functionality` directory to be tested. The results of this initial test run can be viewed in Table 6.

- Expected result.
- Parsing failed, but should have succeeded.
- Parsing succeeded, but should have failed.
- Standalone application timed out.
- Standalone application crashed

	yaml-rust	ruamel.yaml	libfyaml	yaml	js-yaml	go-yaml	The Yaml Component	YAML::PP	NimYAML	ocaml-yaml	yaml-cpp	YamlDotNet	HsYAML	YamlReference	SnakeYAML Engine	eo-yaml
valid_yaml-spec-1.2.yaml																
invalid_expected-failure.yml																

Table 6: Verifying standalone application parser functionality with the “yaml-grammar”[24] repository and a YAML test case file that is syntactically incorrect as per the YAML 1.2.2 specification.[1]

Table 6 shows that almost all standalone applications can parse the `yaml-spec-1.2.yaml` file, except for *YamlDotNet*. The error message that gets written to the log contains the following: `Output to stderr: Error: While parsing a block mapping, did not find expected key. at line 297, column 2`. Even though *YamlDotNet* cannot parse the YAML file, it indicates that the parsing functionality of the standalone application works.

Moreover, Table 6 also depicts that every standalone application is unable to parse the `invalid_expected-failure.yml` file, which further highlights that the parsing functionality of each standalone application is working as expected.

5.3. Limitations

This Subsection emphasizes the limitations of this academic research. The limitations are separated into 3 categories: methodological limitations, scope limitations, and specification-related limitations.

5.3.1. Methodological limitations

Because the academic study focuses on the YAML 1.2.2 specification, test cases are created by reading the specification carefully. However, the parser libraries are not reverse engineered to find bugs in parsing behavior. This research is thus lacking internal analysis, which can help interpret results and find further bugs in YAML parser libraries.

Furthermore, since the test cases are handcrafted while reading the YAML 1.2.2 specification, they may be prone to confirmation bias. The main focus is to see whether YAML parser libraries differ from each other, and thus edge cases might have been selected that do not have real-world implications.

5.3.2. Scope limitations

The YAML 1.2.2 specification defines many features, but not all features are covered with the YAML test case files. For example, YAML 1.2.2 specifies that multiple YAML documents can exist in one YAML file.[1] However, some parser libraries, like *The Yaml Component*, do not support parsing multiple YAML documents at once.[25] Some parser libraries also require a different API call to read multiple documents (e.g., *ruamel.yaml* defines the `load_all` API for reading multiple documents).[26] Thus, the support for parsing multiple documents has not been included in the YAML test case files.

Moreover, the YAML specification supports different Unicode Transformation Format (UTF) encodings than UTF–8. UTF–16 and UTF–32 are also supported as per the YAML 1.2.2 specification. This research focuses on UTF–8 characters. Hence, Byte Order Marks (BOMs) are also not tested.

Furthermore, the output of YAML parsing libraries is not analyzed. A YAML parsing library might be able to parse a YAML test case file; however, it is not evaluated whether the YAML file is parsed semantically correct regarding the YAML 1.2.2 specification. Additionally, YAML parser libraries might differ from each other in terms of semantics.

Lastly, only YAML libraries have been selected that are listed on the official YAML web page[6]. [1] There might be different parser libraries implementing YAML 1.2, but they are not included in this research.

5.3.3. Specification-related limitations

Another limitation of this academic research is that YAML 1.2.2 specifies that parsers can optionally recover from syntax errors. It is not specified how the YAML parsing libraries are supposed to continue after finding a syntax error; it is only proposed that YAML parsing libraries can ignore the incorrect input.[1] This can become problematic because syntax errors do not get visualized correctly in the resulting HTML file. As a workaround, since YAML 1.2.2 specifies to report such errors, the output of stdout and stderr are captured in a log. However,

[6] Official YAML website: https://yaml.org

it is not specified how syntax errors must be reported, so there might be another mechanism than reporting to stdout and stderr that is not captured.[1] In conclusion, this could lead to results where parser libraries are unable to parse a specific YAML test case file but still finish parsing without raising an error.

Furthermore, due to the specifications complexity, it is almost impossible to create a testing suite that tests for every feature in its whole depth.[1] Some tested features might work in a certain YAML test case file but might fail if it is, e.g., nested in a specific way. Each YAML test case file contains a different feature from the YAML 1.2.2. This allows for tracing which specific feature from the YAML 1.2.2 specification is not implemented in the YAML parsing library.

6. Results

This Section highlights the results that are being produced by the YAML Test Suite. The YAML Test Suite runs every standalone application, which calls the corresponding YAML 1.2 parsing library, and stores whether the library can parse the YAML file or not.

In Section 6.1, a statistical overview is being displayed that shows how many libraries can interpret YAML syntax correctly according to the YAML 1.2.2 specification. The unadjusted result table (see Table 7) lacks the context from the logs, while the adjusted result table (see Table 8) shows the changes from the unadjusted results by adding context from the logs.

Section 6.2 interprets whether the results are conforming with the YAML 1.2.2 specification. Two examples that are being tested with the YAML Test Suite are compared to the definition in the YAML specification.

Lastly, further analysis of security-relevant observations is being shown in Section 6.3. This Section focuses on the timeouts and crashes that are being found during the run of the YAML Test Suite, which could lead to security issues in applications that use these libraries. In addition to the data collected by the YAML Test Suite, additional information is being obtained by manually debugging the most pertinent libraries that result in timeouts and crashes.

6.1. Overview of the results from the YAML Test Suite

This Section presents an overview of the results that are being collected from the YAML Test Suite. First, the unadjusted results are being shown and described. These are the raw results without adding context from the logs, solely relying on the exit codes of the standalone applications but ignoring stdout and stderr. Next, the adjusted results are presented with the context of the logs that are produced during the execution of the YAML Test Suite.

6.1.1. Unadjusted results

Table 7 displays the *pass*, *fail*, *timeout*, and *crash rate* of each YAML library that is being tested. They are listed as percentages rounded up to two decimal places. The *pass rate* shows how often the library can interpret YAML test case files correctly according to the YAML 1.2.2 specification. The *fail rate (should pass but fail)* column shows how often the library fails to parse YAML test case files with correct YAML syntax, whereas the *fail rate (should fail but succeed)* column shows the opposite. This metric captures how often the library can successfully parse the YAML test case file, even though the syntax is not correct, according to the YAML 1.2.2 specification. The last two columns, *timeout rate* and *crash rate*, show how often the standalone applications run into the three-second timeout or crash during parsing of YAML test case files.

This statistic represents the unadjusted percentages without considering the logs. Since the YAML 1.2.2 specification allows libraries to recover from syntax errors by reporting them, it may seem in these raw results that some libraries can parse a file when they should fail. However, some libraries might report on the error and continue parsing. This means that some libraries may see an increase in the *pass rate* column while others may see a decrease in the *fail rate (should fail but succeed)* column by adding context from the logs (see Section 6.1.2).

Library	**Pass rate**	**Fail rate (should pass but fail)**	**Fail rate (should fail but succeed)**	**Timeout rate**	**Crash rate**
yaml-rust	80.00%	2.04%	16.73%	0.82%	0.41%
ruamel.yaml	82.45%	2.45%	13.88%	1.22%	0.00%
libfyaml	80.00%	2.86%	17.14%	0.00%	0.00%
yaml	82.04%	2.04%	15.51%	0.41%	0.00%
js-yaml	77.55%	8.98%	13.47%	0.00%	0.00%
go-yaml	71.43%	6.53%	21.63%	0.41%	0.00%
The Yaml Component	57.14%	21.22%	21.22%	0.41%	0.00%
YAML::PP	82.45%	2.04%	15.10%	0.41%	0.00%
NimYAML	81.22%	0.41%	17.96%	0.41%	0.00%
ocaml-yaml	77.55%	3.67%	18.37%	0.41%	0.00%
yaml-cpp	79.18%	2.04%	18.37%	0.41%	0.00%
YamlDotNet	71.02%	14.69%	13.47%	0.41%	0.41%
HsYAML	77.96%	6.12%	13.47%	2.45%	0.00%
YamlReference	70.61%	5.71%	21.22%	2.45%	0.00%
SnakeYAML Engine	75.10%	11.43%	12.65%	0.82%	0.00%
eo-yaml	60.00%	4.08%	35.51%	0.41%	0.00%

Table 7: List of *pass*, *fail*, *timeout*, and *crash rate* of each library under test without adding context from the logs.

6.1.1.1. Analysis of the unadjusted results

According to Table 7, it becomes visible that each library has different results in parsing. In fact, there is no library that shares the same results as another library.

Moreover, there is no library that has interpreted all YAML test cases correctly according to the YAML 1.2.2 specification. The libraries *ruamel.yaml* and *YAML::PP* have the highest pass rate with 82.45%, whereas *The Yaml Component* has the lowest pass rate with 57.14%, closely followed by *eo-yaml* with a 60% pass rate.

In the *fail rate (should pass but fail)* column, *NimYAML* has the lowest rate with 0.41%, while *The Yaml Component* has the highest rate with 21.22%. However, *eo-yaml* ignored the highest amount of syntax errors, which can be seen in the *fail rate (should fail but succeed)* column, with 35.51%. The *SnakeYAML Engine* ignores the least amount of syntax errors, with 12.65%.

The libraries *HSYaml* and *YamlReference* record the highest amount of timeouts during parsing, with a timeout rate of 2.45%. It is further to note that the *libfyaml* and *js-yaml* libraries do not run into a timeout or crash during parsing, while other libraries record at least one timeout during the run of the YAML Test Suite.

Another observation is that there are two parser libraries, *yaml-rust* and *YamlDotNet*, that crash during one YAML test case file, even though the parsing call is being encapsulated in a try-catch statement.

6.1.2. Adjusted results

The adjusted results are taking the logs into consideration for further context. As YAML 1.2.2 specifies that YAML libraries can ignore syntax errors by reporting them, the strategy is to capture stdout and stderr after each standalone application execution. This data is acquired by automatically writing a Python script that filters out any log, where stdout, stderr, or both are empty. By manually analyzing the logs, it can be seen whether a YAML parsing library correctly reports on a syntax error and continues parsing.

By adding context to the unadjusted results, it is possible to determine if a library correctly processes a YAML file by ignoring the syntax error, which adjusts the *pass rate* and *fail rate (should fail but succeed)* column from Table 7. The adjusted results additionally provide insight into whether any library uses the optional reporting feature or not. Table 8 shows the difference between the adjusted and unadjusted results. Only *YAML::PP* is shown in this table, because *YAML::PP* was the only library that reported a syntax error to stdout or stderr during the runtime of the YAML Test Suite but continued parsing. By adjusting the results, *YAML::PP* now has the highest *pass rate* of all libraries that are under test.

Library	Unadjusted pass rate	Adjusted pass rate	Unadjusted fail rate (should fail but succeed)	Adjusted fail rate (should fail but succeed)
YAML::PP	82.45%	82.86%	15.10%	14.96%

Table 8: Comparison between unadjusted and adjusted results.

6.1.2.1. Analysis of adjusted results

A manual review of the logs shows that *YAML::PP* is the only library that reports an error to stderr, even though parsing succeeds (returns exit code zero). However, it has occurred on only one YAML test case file. The contents of the file `invalid_yaml_directive_specifying_higher_major_version.yml` can be viewed in Listing 13.

```yaml
%YAML 2.0
---
```

Listing 13: YAML file specifying with the YAML directive to use a higher major version that does not exist.

YAML 1.2.2 specifies a so-called YAML directive.[1] These directives start with a `%` character and give instructions to the parser,[27] such as defining which YAML version a YAML file complies with. The correct usage of a YAML directive can be seen in Listing 14.[1]

```yaml
%YAML 1.2
```

```
2 ---
```

Listing 14: YAML file specifying a valid YAML version it complies with.

The correct usage of YAML directives is also defined in the YAML 1.2.2 specification. For example, if a higher minor version is declared in the YAML directive, the library should give a warning but still process it. YAML directives that declare a higher major version should not be accepted and should throw an error. The library *YAML::PP* throws an appropriate error message to stderr and still parses the file without aborting the parsing process, which is allowed as per the YAML 1.2.2 specification.[1] The error message can be viewed in Listing 15.

```
1 Unsupported YAML version '%YAML 2.0' at /home/pok3/perl5/lib/perl5/YAML/PP/
  Lexer.pm line 825, <$fh> line 1.
```

Listing 15: *YAML::PP* is throwing a correct error message to stderr, even though parsing succeeded.

6.2. Conformity to the YAML 1.2.2 specification

It can be concluded that the parser libraries under test parse achieve a minimum pass rate of 57.14%, which is more than half of YAML test case files (see Table 7 and Table 8). Six out of 16 parser libraries even achieve a pass rate over 80%. However, there are still multiple examples of YAML parser libraries where the YAML 1.2.2 specification is not fully implemented.

One example is mentioned in Section 6.1.2.1. The YAML test case file `invalid_yaml_directive_specifying_higher_major_version.yml` should not be parsed by YAML parser libraries as per the YAML 1.2.2 specification.[1] However, five out of the 16 parsers can parse this YAML test case file without reporting the user via stdout or stderr, as can be seen in Table 9.

- Expected result.
- Parsing failed, but should have succeeded.
- Parsing succeeded, but should have failed.
- Standalone application timed out.
- Standalone application crashed

	yaml-rust	ruamel.yaml	libfyaml	yaml	js-yaml	go-yaml	The Yaml Component	YAML::PP	NimYAML	ocaml-yaml	yaml-cpp	YamlDotNet	HsYAML	YamlReference	SnakeYAML Engine	eo-yaml
invalid_yaml_directive_specifying_higher_major_version.yml																

Table 9: Five out of 16 parser libraries under test can parse `invalid_yaml_directive_specifying_higher_major_version.yml` without reporting it to the user.

Another example is the parsing of the YAML test case file `invalid_plain_scalar_implicit_key_containing_{}.yml`. In Section 7.3.3 of the YAML 1.2.2 specification, the following is defined: “In addition, inside flow collections, or when used as implicit keys, plain scalars must not contain the “[”, “]”, “{”, “}” and “,” characters. These characters would cause ambiguity with flow collection structures.”[1] Plain scalars are unquoted strings, and an implicit key is a key in a key-value structure that is not explicitly marked as a key.[27] Examples for plain scalars, quoted scalars, and implicit and explicit keys can be found in Listing 16.

```yaml
# Plain scalar
Plain scalar
# Quoted scalar
"Quoted scalar"
# Implicit key-value structure with plain scalars
key: value
# Explicit key-value structure with plain scalars
? key
: value
```

Listing 16: Examples of plain scalars, quoted scalars, and implicit and explicit key-value structures.

As per this definition, YAML libraries should not accept, or at least report it to the user, if the plain scalar implicit key contains one of the following characters: `[`, `]`, `{`, `}` or `,`. However, all libraries tested accept such inputs, as can be seen in Table 10.

- Expected result.
- Parsing failed, but should have succeeded.
- Parsing succeeded, but should have failed.
- Standalone application timed out.
- Standalone application crashed

	yaml-rust	ruamel.yaml	libfyaml	yaml	js-yaml	go-yaml	The Yaml Component	YAML::PP	NimYAML	ocaml-yaml	yaml-cpp	YamlDotNet	HsYAML	YamlReference	SnakeYAML Engine	eo-yaml
invalid_plain_scalar_implicit_key_containing_,.yml																
invalid_plain_scalar_implicit_key_containing_[.yml																
invalid_plain_scalar_implicit_key_containing_[].yml																
invalid_plain_scalar_implicit_key_containing_].yml																

	yaml-rust	ruamel.yaml	libfyaml	yaml	js-yaml	go-yaml	The Yaml Component	YAML::PP	NimYAML	ocaml-yaml	yaml-cpp	YamlDotNet	HsYAML	YamlReference	SnakeYAML Engine	eo-yaml
invalid_plain_scalar_implicit_key_containing_{.yml																
invalid_plain_scalar_implicit_key_containing_{}.yml																
invalid_plain_scalar_implicit_key_containing_}.yml																

Table 10: All parser libraries succeed during parsing of plain scalar implicit keys with prohibited characters.[1]

As a reference, the YAML test case file `invalid_plain_scalar_implicit_key_containing_,.yml` can be seen in Listing 17, which shows how the YAML test case file is structured. All YAML test case files listed in Table 10 are structurally equivalent to `invalid_plain_scalar_implicit_key_containing_,.yml`. The sole variation between them is the substitution of the comma in the reference example with the specific character being evaluated in each individual test case.

```yaml
te,st: test
```

Listing 17: Contents of the `invalid_plain_scalar_implicit_key_containing_,.yml` YAML test case file.

It can be seen that every parser can parse these files, even though the characters are not allowed to be used in plain scalars that are implicit keys. Additionally, none of the libraries under test report the syntax error to stdout or stderr.

Another example, where most YAML parser libraries deviate from the YAML 1.2.2 specification, is the parsing of the `valid_tab_separation_spaces.yml` YAML test case file. Ten out of 16 parser libraries fail to parse this YAML test case file. The YAML 1.2.2 specification defines in Section 6.2: “Outside indentation and scalar content, YAML uses white space characters for separation between tokens within a line. Note that such white space may safely include tab characters.”[1] This means that, to separate between what is YAML syntax and content information, a white space character may be inserted, which can also be a tab character. YAML 1.2.2 uses the “MAY” key word,[1] which per RFC 2119 means that the feature is optional to implement; however, implementations that do not use it must still be able to interoperate with an implementation that uses this optional feature.[14] To test this behavior, the YAML test case file `valid_tab_separation_spaces.yml` is used, where a tab character is inserted between the dash character and the content. The results of parsing this file can be seen in Table 11.

- Expected result.
- Parsing failed, but should have succeeded.
- Parsing succeeded, but should have failed.

- Standalone application timed out.
- Standalone application crashed

	yaml-rust	ruamel.yaml	libfyaml	yaml	js-yaml	go-yaml	The Yaml Component	YAML::PP	NimYAML	ocaml-yaml	yaml-cpp	YamlDotNet	HsYAML	YamlReference	SnakeYAML Engine	eo-yaml
valid_tab_ separation_spaces.yml																

Table 11: Ten out of 16 parser libraries reject tab characters between the token and the content.

Table 11 shows that 37.5% of all parsers under test can parse this file, even though it is allowed as per the YAML 1.2.2 specification. It can be concluded that, while all libraries can parse most YAML test case files correctly, no library under test is fully compliant with the YAML 1.2.2 specification.

6.3. Security-relevant observations

Table 7 shows that 14 out of 16 YAML parsing libraries hit the three-second timeout threshold, except for *libfyaml* and *js-yaml*, at least once. Furthermore, two parsers crashed once during parsing, even though standalone applications call the parsing functionality of the YAML library in a try-catch statement.

6.3.1. Analysis of timeouts during parsing

Except for *libfyaml* and *js-yaml*, 14 out of 16 parsing libraries reach the three-second timeout threshold at least once during the run of the YAML Test Suite. This means that it is possible to write YAML files that impact the parsing time for most libraries. Most parsers hit the timeout on the YAML file `valid_1000000_sequences.yml`, see Table 12.

- Expected result.
- Parsing failed, but should have succeeded.
- Parsing succeeded, but should have failed.
- Standalone application timed out.
- Standalone application crashed

	yaml-rust	ruamel.yaml	libfyaml	yaml	js-yaml	go-yaml	The Yaml Component	YAML::PP	NimYAML	ocaml-yaml	yaml-cpp	YamlDotNet	HsYAML	YamlReference	SnakeYAML Engine	eo-yaml
valid_1000000_sequences.yml																

Table 12: Most timeouts are being recorded on the `valid_1000000_sequences.yml` YAML test case file.

The `valid_1000000_sequences.yml` YAML test case file is a 23 Megabyte (MB) file, which holds 1,000,000 sequences. A sequence in YAML is an ordered list of zero or more nodes.[1], [27] The first and last five lines of this YAML file can be viewed in Listing 18.

```yaml
- 0: 'Test 0'
- 1: 'Test 1'
- 2: 'Test 2'
- 3: 'Test 3'
- 4: 'Test 4'
[...]
- 999995: 'Test 999995'
- 999996: 'Test 999996'
- 999997: 'Test 999997'
- 999998: 'Test 999998'
- 999999: 'Test 999999'
```

Listing 18: First and last five lines of the `valid_1000000_sequences.yml` YAML test case file.

Table 12 shows that 81.25% of libraries under test cannot parse the file in under three seconds. Additionally, the *SnakeYAML Engine* fails to parse this file. The error message hints that there might be a limit as to how much data can be handled in the *SnakeYAML Engine*, and the limit is three MB. However, this only affects the default configuration, since the maximum size can be configured in *SnakeYAML Engine* with the `LoadSettingsBuilder` class in the `org.snakeyaml.engine.v2.api` package.[28] The error message can be seen in Listing 19.

```
org.snakeyaml.engine.v2.exceptions.YamlEngineException: The incoming YAML
document exceeds the limit: 3145728 code points.
[...]
```

Listing 19: *SnakeYAML Engine* throws an error message during the parsing of `valid_1000000_sequences.yml`.

Apart from that, the two parsing libraries, *libfyaml* and *js-yaml*, were still able to parse this YAML test case file in under three seconds, even though the file has 23 MB of data to process. Furthermore, the *yaml* library, which is written in JavaScript, records a timeout during the parsing of this test case file, while *js-yaml*, which is also written in JavaScript, does not.

The execution times of each standalone application are being measured with the `time` command, and the output is listed in Table 13. The *real* measurement shows how long the standalone application takes from start to finish. On the other hand, the *user* metric captures how many Control Processing Unit (CPU) seconds the process takes in user mode. The *sys* metric measures the CPU seconds that are being spent in kernel mode.[29]

YAML parser library	*Real* execution time	*User* execution time	*Sys* execution time
yaml-rust	0m9.384s	0m8.555s	0m0.679s
ruamel.yaml	4m24.353s	3m57.142s	0m23.395s
libfyaml	0m2.678s	0m1.672s	0m0.951s
yaml	0m30.526s	0m30.402s	0m2.870s
js-yaml	0m3.209s	0m3.157s	0m0.466s
go-yaml	0m11.333s	0m11.678s	0m3.660s
The Yaml Component	0m18.205s	0m17.411s	0m0.489s
YAML::PP	3m10.070s	2m52.243s	0m15.371s
NimYAML	0m12.593s	0m3.339s	0m8.635s
ocaml-yaml	0m5.571s	0m4.938s	0m0.549s
yaml-cpp	0m39.729s	0m37.504s	0m1.823s
YamlDotNet	0m7.815s	0m7.564s	0m0.513s
HsYAML	2m51.439s	2m47.810s	0m2.403s
YamlReference	3m46.352s	3m40.405s	0m3.591s
SnakeYAML Engine	0m1.468s	0m2.298s	0m0.341s
eo-yaml	0m6.461s	0m6.548s	0m1.837s

Table 13: Parser execution times for the `valid_1000000_sequences.yml` test case file.

It can be observed that the *user* execution time is sometimes longer than the *real* execution time. That is because the VM has two processor cores, and parsing may be split between these cores for optimization reasons (e.g., by using multithreading). By adding the CPU seconds of both processor cores, it is possible to see a longer *user* execution time than the *real* execution time.[29]

From Table 13 it can be inferred that the libraries *ruamel.yaml*, *YAML::PP*, *HsYAML*, and *YamlReference* take the longest time for parsing. The fastest library to parse this file is *libfyaml*, with 2.678 seconds. The library *NimYAML* takes 12.593 seconds to parse this file; however, 8.635 seconds are being used in kernel mode, while the standalone application only took 3.339 seconds in user mode. This can be an indicator that the reading of the YAML test case file takes longer than the parsing of the file. The `readFile` function from the standard syncio library is used to read the file before parsing it with *NimYAML*,[30] as can be seen in Listing 20.

```nim
[...]
        let content = readFile(filePath)
        var doc: YamlNode
        load(content, doc)
[...]
```

Listing 20: An excerpt from the standalone application that uses *NimYAML* to parse the file.

The reason most libraries breach the threshold on this specific YAML test case file is that there is much data to process. Except for *go-yaml* and *SnakeYAML Engine*, every parser library under test is using up 100% of one processor core. There is significant processing overhead when loading a YAML document with 1,000,000 sequences because a correspondingly large number of parser events and in-memory objects must be created and handled.

6.3.1.1. Analysis of timeouts by *HsYAML* and *YamlReference*

Further investigation shows that *HsYAML* and *YamlReference* record timeouts on the same YAML test case files. In Table 14 this behavior can be observed.

- Expected result.
- Parsing failed, but should have succeeded.
- Parsing succeeded, but should have failed.
- Standalone application timed out.
- Standalone application crashed

	yaml-rust	ruamel.yaml	libfyaml	yaml	js-yaml	go-yaml	The Yaml Component	YAML::PP	NimYAML	ocaml-yaml	yaml-cpp	YamlDotNet	HsYAML	YamlReference	SnakeYAML Engine	eo-yaml
valid_10000000_spaces_indentation.yml																
valid_1000000_sequences.yml																
valid_5000_json_brackets.yml																
valid_block_collection_huge_compact_in_line_notation.yml																
valid_integer_overflow_larger.yml																
valid_large_image.yml																

Table 14: The libraries *HsYAML* and *YamlReference* timeouts during the same YAML test case files.

These two libraries still differ from each other when it comes to parsing. There are multiple examples when these libraries yield different results with different YAML test case files. The differences between the two parsers can be viewed in Table 15.

- Expected result.
- Parsing failed, but should have succeeded.
- Parsing succeeded, but should have failed.
- Standalone application timed out.
- Standalone application crashed

	HsYAML	YamlReference
invalid_bad_anchor_name_,.yml		
invalid_bad_anchor_name_[.yml		
invalid_bad_anchor_name_].yml		
invalid_bad_anchor_name_{.yml		
invalid_bad_anchor_name_}.yml		
invalid_default_secondary_handle_str_as_int.yml		
invalid_duplicate_keys.yml		
invalid_generic_mapping_with_duplicate_keys.yml		
invalid_multiple_yaml_directives.yml		
invalid_non_specific_tag_mixed_with_predefined_tag.yml		
invalid_plain_scalar_starting_with_indicator_*.yml		
invalid_reference_to_unknown_anchor.yml		
invalid_reserved_indicators_@.yml		
invalid_reserved_indicators_`.yml		
invalid_shorthand_tag_empty_suffix.yml		
invalid_shorthand_tag_never_referenced.yml		
invalid_shorthand_tag_uri_encoded_at_the_start.yml		
invalid_tag_shorthand_suffix_!.yml		
invalid_yaml_directive_specifying_higher_major_version.yml		
valid_infinite_recursion.yml		

Table 15: Differences between *HsYAML* and *YamlReference* during parsing.

These differences show that the parsers are interpreting the results differently on multiple occasions but still run into a timeout on the same test case files. Moreover, both libraries never finish parsing when supplied with the YAML test case files `valid_5000_json_brackets.yml` and `valid_large_image.yml`. This behavior is tested by running both standalone applications over one night and supplying the path to the YAML test case files per command line argument. The result is that both libraries use 100% of one processor core during the process and do not finish parsing.

First, the `valid_5000_json_brackets.yml` is being analyzed. This YAML test case file consists of 5000 opening square brackets and 5000 closing square brackets. The YAML 1.2.2 specification calls structures that consist of an opening and a closing square bracket flow sequences. To test this behavior from a dynamic perspective, the number of opening and closing square brackets is reduced to 14 opening and 14 closing square brackets. To see a pattern, one opening and closing bracket is being added after measuring the time of the standalone application. Table 16 displays these measurements. The column *Amount of opening and closing*

square brackets describes how many opening and closing brackets are being used; hence, if the column entry is, e.g., 15, it means there are 15 opening and 5 closing square brackets. Furthermore, the *real* time from the time command is being displayed.[29]

Amount of opening and closing square brackets	**Time for parsing with HsYAML (*real*)**	**Time for parsing with YamLReference (*real*)**
15	0m1.910s	0m2.730s
16	0m3.716s	0m5.411s
17	0m7.454s	0m10.786s
18	0m14.926s	0m21.536s
19	0m30.649s	0m43.502s
20	1m0.489s	1m29.263s
21	2m3.140s	3m0.286s
22	4m1.767s	5m55.066s
23	8m6.520s	11m59.570s
24	16m47.086s	24m4.767s
25	33m47.098s	48m11.458s

Table 16: List of time measurements by reducing the amount of square brackets of the `valid_5000_json_brackets.yml` YAML test case file.

It can be seen that the more square brackets are getting added to the YAML file, the execution time doubles in both libraries. By visualizing the data, a nonlinear pattern is being observed in the execution time (see Figure 5).

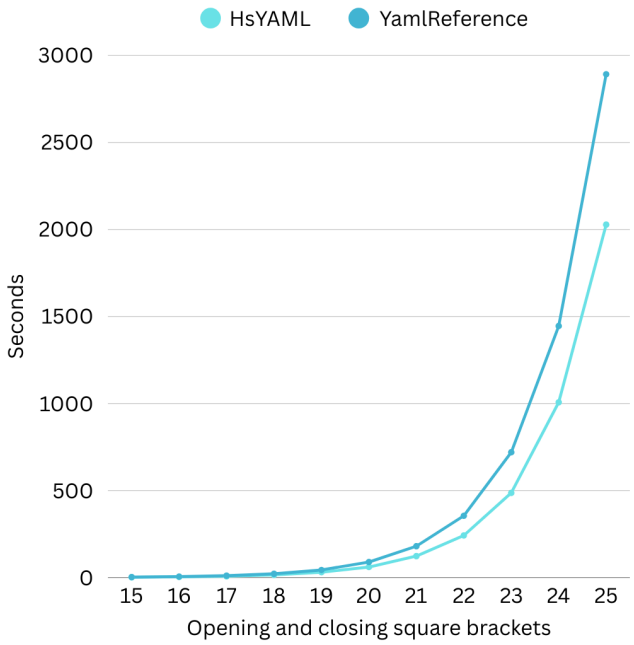

Figure 5: Visualization of the execution times that are being observed by adding more opening and closing square brackets.

The problem can be identified in the source code of *HsYAML* and *YamlReference*. In *HsYAML* the issue lies in lines 597-609 of the `Data.YAML.Token` module. The code snippet of the highlighted lines can be seen in Listing 21.

```Haskell
[...]
-- | @first <|> second@ tries to parse /first/, and failing that parses
-- /second/, unless /first/ has committed in which case is fails immediately.
instance Alternative Parser where
  empty = pfail "empty"

  left <|> right = Parser $ \state -> decideParser state D.empty left right
  state
    where
      decideParser point tokens left right state =
        let reply = applyParser left state
            tokens' = D.append tokens $ reply^.rTokens
        in case (reply^.rResult, reply^.rCommit) of
                (Failed _,    _)      -> Reply { rState  = point,
                                                 rTokens = D.empty,
                                                 rResult = More right,
                                                 rCommit = Nothing }
                (Result _,   _)       -> reply { rTokens = tokens' }
                (More _, Just _)      -> reply { rTokens = tokens' }
                (More left', Nothing) -> decideParser point tokens' left' right
                (reply^.rState)
[...]
```

Listing 21: Function `decideParser` in the *HsYAML* library that leads to an exponential growth in execution time.[31]

The code snippet shows the function `decideParser` in the library *HsYAML*, which is a backtracking implementation. By supplying the YAML test case file `valid_5000_json_brackets.yml`, at some point it receives the opening square bracket and assumes it to be a flow sequence. The function tries to prove this assumption by calling itself recursively while the parser reports the result to be incomplete.[31] However, `valid_5000_json_brackets.yml` consists of 5000 opening square brackets, which leads to a high amount of recursive parsing before the corresponding closing square bracket allows it to complete. This results in increasing execution times and high memory consumption.

A similar code snippet can be observed in the *YamlReference* library (see Listing 22), which is observed to lead to the same problem as in the *HsYAML* library after manually debugging the standalone application.

```Haskell
[...]
-- | @decide first second@ tries to parse /first/, and failing that parses
-- /second/, unless /first/ has committed in which case is fails immediately.
decide :: Parser result -> Parser result -> Parser result
decide left right = Parser $ \ state ->
  let Parser parser = decideParser state D.empty left right
```

```
7     in parser state
8     where decideParser point tokens (Parser left) right = Parser $ \state ->
9             let reply = left state
10                tokens' reply = D.append tokens $ reply|>rTokens
11            in case (reply|>rResult, reply|>rCommit) of
12                    (Failed _,    _)      -> Reply { rState  = point,
13                                                     rTokens = D.empty,
14                                                     rResult = More right,
15                                                     rCommit = Nothing }
16                    (Result _,    _)      -> reply { rTokens = tokens' reply }
17                    (More left', Just _)  -> reply { rTokens = tokens' reply,
18                                                     rResult = More left' }
19                    (More left', Nothing) -> let Parser parser = decideParser
                      point (tokens' reply) left' right
20                                              in parser $ reply|>rState
21  [...]
```

Listing 22: Function `decide` in the *YamlReference* library that leads to an exponential growth in execution time.[32]

Next, the high parsing time by parsing `valid_large_image.yml` is being inspected. The `valid_large_image.yml` YAML test case file consists of a base64 encoded picture. The YAML test case file has a size of four MB and 52,635 lines. A snippet of the mentioned `valid_large_image.yml` can be seen in Listing 23.

```yaml
1  picture: !!binary |
2    Qk32xi0AAAAADYAAAAoAAAA6AMAAOgDAAABABgAAAAAAMDGLQDEDgAAxA4AAAAAAAAAAAAAAAAA
3    AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
4  [...]
```

Listing 23: Snippet of the `valid_large_image.yml` YAML test case file.

To observe this behavior, the YAML test case file is first minimized to 6,350 lines with a file size of 492 Kilobyte (KB) so that the execution can be profiled within reasonable time and memory limits. By running the standalone applications with the built-in profiler of the Glasgow Haskell Compiler (GHC) and supplying the minimized YAML test case file, differences between the *HsYAML* and *YamlReference* libraries can be observed.

In the profiling report of *HsYAML*, it can be seen that 92.1% of all allocations occur in the `Data.YAML.Event` module on lines 368-418. Furthermore, 84.6% of the total execution time is being spent in this code region. Lastly, a total of approximately 140 GB of memory is allocated during the runtime of the standalone application.

The profiling report for *YamlReference* differs significantly from *HsYAML*. Allocations are distributed across multiple functions in the `Text.Yaml.Reference` module. The highest allocation share (18.1%) is recorded in lines 706-710 of this module, with 31.4% of the execution time spent in this code region. In contrast to *HsYAML*, *YamlReference* allocates only a fraction of the total memory, approximately 9 GB.

Increasing the number of lines in the minimized version of `valid_large_image.yml` leads to increased memory allocation in both libraries. Under sufficiently large inputs, this behavior can exhaust available memory, effectively preventing successful parsing.

6.3.2. Analysis of stack overflow errors

In the results of the YAML Test Suite, two libraries crash during parsing. The library *YamlDotNet* records a crash during the parsing of the `valid_5000_json_brackets.yml` file, and *yaml-rust* crashes when parsing `valid_block_collection_huge_compact_in_line_notation.yml`. In Table 17 it can be seen that not all libraries can parse that file successfully; however, they do not lead to a crash and exit with the predetermined exit codes.

- Expected result.
- Parsing failed, but should have succeeded.
- Parsing succeeded, but should have failed.
- Standalone application timed out.
- Standalone application crashed

	yaml-rust	ruamel.yaml	libfyaml	yaml	js-yaml	go-yaml	The Yaml Component	YAML::PP	NimYAML	ocaml-yaml	yaml-cpp	YamlDotNet	HsYAML	YamlReference	SnakeYAML Engine	eo-yaml
valid_5000_json_brackets.yml																
valid_block_collection_huge_compact_in_line_notation.yml																

Table 17: Showcase of all libraries under test during the parsing of the YAML test case files `valid_5000_json_brackets.yml` and `valid_block_collection_huge_compact_in_line_notation.yml`.

The parsing calls from the libraries are being encapsulated in a try-catch statement in the standalone applications to prevent crashes during parsing. Both libraries crash on different YAML test case files; however, the error that leads to this crash is the same: a stack overflow error. A memory protection fault is initiated, or adjacent memory regions are overwritten by stack frames when a program exceeds the allocated stack memory, resulting in a stack overflow error.[33]

6.3.2.1. Analysis of the stack overflow error in *YamlDotNet*

A stack overflow in *YamlDotNet* is recorded during parsing of the YAML test case file `valid_5000_json_brackets.yml`. This YAML test case file also leads to a nonlinear growth in the libraries *HsYAML* and *YamlReference* (see Section 6.3.1). The library *YamlDotNet* prints the stack trace to stderr, which also displays the functions that are responsible for this behavior. An excerpt with the most relevant data can be seen in Listing 24.

```
1 [...]
  at YamlDotNet.Serialization.NodeDeserializers.CollectionNodeDeserializer.
  DeserializeHelper(System.Type, YamlDotNet.Core.IParser,
2 System.Func`3<YamlDotNet.Core.IParser,System.Type,System.Object>,
  System.Collections.IList, Boolean, YamlDotNet.Serialization.INamingConvention,
  YamlDotNet.Serialization.ITypeInspector, System.Action`2<Int32,System.Object>)
  at YamlDotNet.Serialization.NodeDeserializers.CollectionNodeDeserializer.
3 Deserialize(YamlDotNet.Core.IParser, System.Type,
  System.Func`3<YamlDotNet.Core.IParser,System.Type,System.Object>, System.Object
  ByRef, YamlDotNet.Serialization.ObjectDeserializer)
  at YamlDotNet.Serialization.ValueDeserializers.NodeValueDeserializer.
4 DeserializeValue(YamlDotNet.Core.IParser, System.Type,
  YamlDotNet.Serialization.Utilities.SerializerState,
  YamlDotNet.Serialization.IValueDeserializer)
  at YamlDotNet.Serialization.ValueDeserializers.AliasValueDeserializer.
5 DeserializeValue(YamlDotNet.Core.IParser, System.Type,
  YamlDotNet.Serialization.Utilities.SerializerState,
  YamlDotNet.Serialization.IValueDeserializer)
  at YamlDotNet.Serialization.ValueDeserializers.NodeValueDeserializer+<>c__
6 DisplayClass6_0.<DeserializeValue>b__1(YamlDotNet.Core.IParser, System.Type)
7 [...]
```

Listing 24: Excerpt from the stack trace that *YamlDotNet* prints to stderr shows functions that are responsible for the stack overflow.

The following classes are perpetually being called during the parsing of this YAML test case file:

1. `Serialization.ValueDeserializers.AliasValueDeserializer.DeserializeValue` (lines 99-147)
2. `Serialization.ValueDeserializers.NodeValueDeserializer.DeserializeValue` (lines 52-103)
3. `Serialization.NodeDeserializers.CollectionNodeDeserializer.Deserialize` (lines 45-82)
4. `Serialization.NodeDeserializers.CollectionNodeDeserializer.DeserializeHelper` (lines 84-130)

From the source code of *YamlDotNet*, it can be observed that the library tries to deserialize the YAML input. These four functions are called continuously, without returning, until the deepest nesting layer is deserialized.[34] For each additional nesting layer, four new stack frames are stored on the call stack. Due to the huge nesting in `valid_5000_json_brackets.yml`, this causes the stack to overflow at some point during parsing.

6.3.2.2. Analysis of the stack overflow error in *yaml-rust*

The stack overflow in *yaml-rust* is triggered by the YAML test case file `valid_block_collection_huge_compact_in_line_notation.yml`. A shortened version of this file can be viewed in Listing 25. The file consists of a nested sequence that has 4000 nesting layers.

```yaml
1 - - - [...] - - - test
```

YAML

Listing 25: Shortened version of the `valid_block_collection_huge_compact_in_line_notation.yml` YAML test case file.

The cause can be seen in the source code of *yaml-rust* in lines 225-239 and 270-273 in the `parser.rs` source file, which are the functions `load_node` and `load_sequence`. Whenever a new node is detected, the `load_node` function is called. The `load_node` function sees that a sequence starts, which calls the function `load_sequence` (see Listing 26).

```Rust
[...]
    fn load_node<R: MarkedEventReceiver>(
        &mut self,
        first_ev: Event,
        mark: Marker,
        recv: &mut R,
    ) -> Result<(), ScanError> {
        match first_ev {
            Event::Alias(..) | Event::Scalar(..) => {
                recv.on_event(first_ev, mark);
                Ok(())
            }
            Event::SequenceStart(_) => {
                recv.on_event(first_ev, mark);
                self.load_sequence(recv)
            }
            Event::MappingStart(_) => {
                recv.on_event(first_ev, mark);
                self.load_mapping(recv)
            }
            _ => {
                println!("UNREACHABLE EVENT: {:?}", first_ev);
                unreachable!();
            }
        }
    }
[...]
```

Listing 26: Function `load_node` from the yaml-rust GitHub repository.[35]

However, in `load_sequence`, the function `load_node` is called again. And this cycle happens as long as there is nesting (see Listing 27). Because `valid_block_collection_huge_compact_in_line_notation.yml` has 4000 nesting levels, these two functions call themselves perpetually, by which the call stack grows since both functions cannot return until the end of the nesting is reached. Eventually, the call stack grows too large, resulting in a stack overflow error.

```Rust
[...]
    fn load_sequence<R: MarkedEventReceiver>(&mut self, recv: &mut R) ->
    Result<(), ScanError> {
        let (mut ev, mut mark) = self.next()?;
        while ev != Event::SequenceEnd {
            self.load_node(ev, mark, recv)?;

```

```
7               // next event
8               let (next_ev, next_mark) = self.next()?;
9               ev = next_ev;
10              mark = next_mark;
11          }
12          recv.on_event(ev, mark);
13          Ok(())
14      }
15  [...]
```

Listing 27: Function `load_sequence` from the yaml-rust GitHub repository.[35]

7. Discussion

This Section discusses the results from Section 6. In Section 7.1 the differences between YAML 1.2 parser libraries are highlighted, and the potential implications on applications that use these libraries are addressed. Section 7.2 questions the effectiveness of the YAML 1.2.2 specification, because no parser library is observed to be fully compliant. The security risks concerning applications that use these libraries are further elaborated in Section 7.3. Lastly, the research questions from Section 1 are answered in Section 7.4 with the context of the results of this research.

7.1. Parsing differences across YAML 1.2 parsers

No two parser libraries produce the same results in the YAML Test Suite. Thus, YAML 1.2 lacks portability in the current parser libraries. A YAML file that works on one library, might fail to be parsed with another library. Hence, the exchange of a YAML parser library may pose an issue. The behavior that is observed essentially leads to a library lock-in, which under certain circumstances makes it difficult to migrate to another library. The impact can range from simple inconveniences to severe issues in production systems, where a YAML file, that e.g., stores configuration data for an application, needs to be refactored or even rewritten to be compatible with the other YAML parser library.

Furthermore, if applications use YAML for machine-to-machine communication, it must be ensured that both libraries use the same YAML parser libraries, even though YAML is supposed to support data sharing across programming languages.[1] For example, the YAML test case file `invalid_yaml_directive_specifying_higher_major_version.yml` is accepted by the libraries *yaml-rust*, *yaml*, *NimYAML*, *YamlReference* and *eo-yaml*, but others do not accept such input. *YAML::PP* goes a different route by reporting the error to stderr, ignoring the wrong syntax, and continuing parsing. This can lead to inconsistencies in machine-to-machine communication, especially since the YAML directive specifies the version in which the YAML file is supposed to be interpreted. Especially, since it is not specified how much of the malformed syntax can be ignored, some libraries might keep certain parts, while others skip to the next line. A YAML file would have to be tested against the parser libraries that are in place, to figure out, which one works on all libraries.

7.2. Non-compliance with the YAML 1.2.2 specification

The results of this research demonstrate that no parser is fully compliant with the YAML 1.2.2 specification. The problem is that many complex features are specified, which can be difficult to implement consistently across libraries and programming languages.[1] In the yaml-grammar repository, which is maintained by The YAML Project, the authors state in

the README file: “Fully comprehending the YAML grammar is quite an undertaking for most mortals. Creating a fully compliant parser has proven almost impossible.”[24]

However, The YAML Project offers some tools that can help create a YAML 1.2.2 compliant parser. Repositories such as yaml-grammar[7], yaml-test-suite[8], or one of the YAML reference parsers[9][10] can help in understanding the YAML syntax, which in turn supports the development of a fully compliant YAML 1.2.2 parser library.[1]

Nonetheless, even *YAML::PP*, which records the highest pass rate of all parser libraries under test, interprets approximately 41 YAML test case files incorrectly. An example of that is the `valid_tab_separation_spaces.yml` test case file (see Table 11) which is explicitly stated in the YAML 1.2.2 specification to be valid; however, *YAML::PP* still interprets the file as a syntax error. Such cases make YAML files less portable between libraries, impact machine-to-machine communication where multiple programming languages and different YAML libraries are used, and undermine the effectiveness of a specification.

Solutions to this problem might be to introduce conformance levels, which every YAML parser needs to explicitly state. Currently, when using a YAML parser, some parser libraries make it challenging to determine which features are implemented from the YAML 1.2.2 specification. An example is the *yaml-rust* library, which describes itself as fully compliant with the YAML 1.2 specification,[35] even though it fails to interpret 49 YAML test case files correctly. From a user’s perspective, this makes it difficult to determine which parsers support what functionality. The library *eo-yaml*, on the other hand, describes all supported functionalities in the README file of the repository.[36] For transparency reasons, the YAML specification could introduce conformance levels, that have to be explicitly stated by every parser library. These would function as a label as to how compliant the YAML parser library is to the YAML specification, and help users determine the correct YAML parser libraries for their project.

Another idea is to specify which syntax errors have to be ignored and reported to the user, and which should cause a failure in parsing. At present, it is up to the parser library to decide, which syntax errors can be recovered from and which cause a hard failure.[1] This can be implemented by adding the expected behavior on success and syntax error to each described feature in the YAML specification.

7.3. Security risks

Multiple potential security vulnerabilities are identified that impact the availability of the YAML parsing libraries. In total, 14 out of 16 parser libraries under test breached the three-second execution threshold of the YAML Test Suite at least once. Most of them run into the three-second timeout, when parsing the YAML test case file `valid_1000000_sequences.yml`. While most parser libraries still take under a minute to finish parsing this file, the parser libraries *ruamel.yaml*, *YAML::PP*, *HsYAML*, and *YamlReference* take a few minutes to complete. Furthermore, during parsing, they use up 100% of the CPU on one processor core. In certain scenarios this can enable DoS attacks, especially if used on resource-constrained devices. The developer behind *ruamel.yaml*, Anthon van der Neut, addressed this issue in the latest

[7] yaml-grammar repository: https://github.com/yaml/yaml-grammar
[8] yaml-test-suite repository: https://github.com/yaml/yaml-test-suite
[9] yaml-reference-parser repository: https://github.com/yaml/yaml-reference-parser
[10] Interactive YAML reference parser: http://ben-kiki.org/ypaste/

patch. Starting with version 0.19.0, the documentation includes a warning regarding the processing of unchecked input and recommends that the file size be verified before processing with *ruamel.yaml*.[37] However, the vulnerability persists. A potential solution to this problem is the *SnakeYAML Engine*, which defaults to only accepting YAML files with a size up to three MB. Users can still to adapt the maximum allowed file size if they wish to.

The libraries *HsYAML* and *YamlReference* exhibit even more severe performance issues when parsing the YAML test case files `valid_5000_json_brackets.yml` and `valid_large_image.yml`. Both libraries need a significant amount of time to finish parsing these two YAML test case files. First, `valid_5000_json_brackets.yml` needs a long time to parse in *HsYAML* and *YamlReference* because the execution time doubles the more opening and closing square brackets exist in the YAML file. This can enable DoS attacks, even on files with small file sizes. By adding 25 opening and closing square brackets to a YAML file, *HsYAML* needs approximately 33 minutes, and *YamlReference* approximately 48 minutes to finish parsing. Additionally, the parsing uses 100% of a processor core, ultimately blocking it until the parsing process finishes. A YAML file with 50 bytes can cause these two parsers to consume many resources and block a processor core. In contexts, where user-supplied YAML input is allowed, this can become a DoS attack vector, since the file size is not the decisive point. The reason behind this security issue is a backtracking implementation, which consumes many resources in its process. To mitigate this vulnerability, a possible solution can be to set a default maximum recursion depth and throw an error if this depth is breached. This would make the library safe to be used by default.

Additionally, the YAML test case file `valid_large_image.yml` fails on both libraries due to different reasons. The input is so large, that many allocations happen in the process of parsing this file. While *HsYAML* and *YamlReference* both run into this timeout, there is a difference in the point of failure. *HsYAML* allocates approximately 140 GB memory in a single function, whereas *YamlReference* allocates only approximately 9 GB memory, which are distributed across multiple functions. Even though the point of failure differs in these two libraries, the problem with the excessive amount of parsing time with `valid_large_image.yml` is caused by huge memory consumption. With more lines in `valid_large_image.yml` the more memory gets allocated, which slows down the parsing process tremendously.

The results further highlight two examples of parser libraries that crash during the run of the YAML Test Suite. The library *YamlDotNet* crashes when parsing the YAML test case file `valid_5000_json_brackets.yml`. The reason is that, until the deepest nesting layer is reached, the library calls four functions continously in a recursive manner without returning. Thus, the more opening and closing square brackets exist in the YAML file, the more stack frames are being created in the process of parsing. After adding too many stack frames, the library ultimately runs into a stack overflow error, which cannot be handled by a try-catch statement. This means, an application that uses this library can crash when parsing a similar YAML file without the possibility to handle the error. The developer behind *YamlDotNet*, Antoine Aubry, wants to address this issue by adding the method `WithMaximumRecursion` to the `DeserializerBuilder`, which allows users to optionally limit the maximum allowed nesting depth. However, it is not planned to add a default maximum allowed nesting depth.[38]

The library *yaml-rust* crashes by parsing the YAML test case file `valid_block_collection_huge_compact_in_line_notation.yml` for similar reasons as to why *YamlDotNet* crashes. The file `valid_block_collection_huge_compact_in_line_notation.yml`

contains a deeply nested structure, which causes the *yaml-rust* parser to call the functions `load_node` and `load_sequence` recursively without returning until the maximum stack size is reached. This eventually leads to a stack overflow. The *yaml-rust2* library, which is a fork of *yaml-rust*, deploys a fix for this problem by managing nesting via the heap and not the stack. [39] This shows that such issues can be resolved by changing the implementation of the parser library.

7.4. Revisiting research questions

This Subsection revisits the research questions introduced in Section 1 and discusses them regarding the experimental results that are presented in Section 6.

The results demonstrate that no parser fully conforms to the YAML 1.2.2 specification. While all YAML libraries can interpret more than half of the YAML test cases correctly, no parser library achieves a 100% pass rate. Thus, YAML parsers are observed to behave inconsistently. When parsing the same set of YAML test case files, no two parser libraries yield the same results. Some parser libraries reject syntactically valid inputs and abort parsing, whereas others successfully parse syntactically invalid inputs. All parser libraries under test record false positives — accepting inputs that are syntactically incorrect as per the YAML 1.2.2 specification without reporting errors — and false negatives — rejecting inputs that are syntactically correct as per the YAML 1.2.2 specification, which makes them only partially reliable and inconsistent.

Furthermore, several YAML 1.2.2 grammar rules are inconsistently implemented, and in certain cases even disregarded by every parser library under test. For example, most parser libraries reject the syntactically invalid YAML directive that specifies a higher major version by aborting parsing. However, five out of 16 parser libraries ignore this error, and continue parsing without reporting it. Another example is the parsing of a YAML test case file with a tab character between the token and the content. Some parsers do accept this syntax; however, the majority treat this input as a syntax error. These examples show that specific language rules from the YAML 1.2.2 specification are inconsistently implemented throughout the YAML parser libraries under test. Furthermore, the results present an example in which all parser libraries uniformly disregard a language rule from the YAML 1.2.2 specification. A YAML test case file that mixes plain scalar implicit keys with special characters is shown to be accepted by every parser library, although the YAML 1.2.2 specification explicitly states this use to be syntactically incorrect. The inconsistencies and uniform disregarding of YAML language rules undermine the YAML 1.2.2 specification as a dependable reference for YAML parser implementers and YAML users, indicating that certain rules are treated as optional in practice, despite being explicitly specified.

From a security perspective, multiple potential DoS attack vectors are identified in the results. Most parser libraries breach the predefined three-second timeout, the only exceptions are *libfyaml* and *js-yaml*. In certain parser libraries the execution time of a 23 MB YAML test case file takes several minutes to complete. In addition, the libraries *HsYAML* and *YamlReference* are observed to not finish parsing when supplied with two YAML test case files from the YAML Test Suite. Finally, *yaml-rust* and *YamlDotNet* record one crash during the run of the YAML Test Suite. Applications that utilize these libraries to parse user-supplied or unchecked YAML input, can be susceptible to DoS attacks, when no other measure is in place.

8. Conclusion

The goal of this research is to determine whether YAML 1.2 parser libraries that are listed on the official YAML website conform to the YAML 1.2.2 specification. Analysis of parsing behavior, the robustness of parser libraries, and security-relevant implications that can be deduced from this research are the primary objectives. For that, an automated YAML Test Suite is created to evaluate selected YAML 1.2 parser libraries against a set of 245 YAML test case files. These test case files are generated in accordance with the YAML 1.2.2 specification.

During this research, it was found that no parser library is fully compliant with the YAML 1.2.2 specification. While all parser libraries can parse more than half of the YAML test case files, various inconsistencies are identified in both acceptance and rejection behavior. In certain cases, all YAML parser libraries collectively neglect language rules that are defined in the YAML 1.2.2 specification. Different examples show that some YAML libraries can parse the YAML test case file correctly, but others fail. Lastly, there are parser libraries that accept syntax that is invalid as per the YAML 1.2.2 specification, but still parse it as if it were syntactically correct. Overall, at least once, each parser library fails when it should have succeeded and succeeds parsing when it should have failed due to syntax errors in the YAML input. The observed behavior undermines the effectiveness of a specification in ensuring interoperable and predictable parsing behavior.

Moreover, all parser libraries deliver different results during the run of the YAML Test Suite. When parsing the same set of YAML files, no two parser libraries yield identical outcomes. It is evident from these inconsistencies that YAML is neither predictable nor portable across parser library implementations.

In addition, some parser libraries are observed to parse an excessive amount of time when supplied with certain valid YAML test case files. The most timeouts during the run of the YAML Test Suite are recorded when parsing a YAML file that has a file size of 23 MB. However, the parser libraries *HsYAML* and *YamlReference* are shown to grow their execution times in a nonlinear manner, when supplied with a YAML file with many opening and closing square brackets, that lead to huge nesting depths. In a different example, these two parser libraries are supplied with a YAML file that stores an image, which causes both libraries to allocate huge amounts of memory, which slows down the parsing process to a point where parsing does not end. In all of these examples, most parser libraries are observed to utilize 100% of the CPU on one processor core.

Finally, two parser libraries crash upon parsing two different YAML parsing libraries. The problem is, however, identical. Both parser libraries crash due to YAML test case files that contain deeply nested structures. The result is a stack overflow, which is caused by recursive function calls that make the call stacks grow until they overflow.

In summary, no parser library is observed to be truly compliant with the YAML 1.2.2 specification, although all parser libraries can parse more than half of the YAML test case files correctly. A side effect of non-compliance is that all parsers behave non-uniformly, which is highlighted by the results, since no two parser libraries deliver the same results. And when certain features from the YAML 1.2.2 specification are adopted, they can lead to excessive execution time or crashes in some parser libraries, which, if used in applications for the processing of untrusted YAML input, can lead to DoS attacks.

8.1. Future Work

Future research could focus on defining more YAML test case files. The YAML Test Suite does not encompass all the features defined in the YAML 1.2.2 specification; therefore, there is still untapped potential for additional YAML test case files. For example, this research does not analyze different character encodings except for UTF–8.

Because the YAML test case files are written by reading the YAML 1.2.2 specification, there might still be potential for further security observations. Reverse engineering parser libraries and finding bugs in the parsing logic could yield further results in YAML parsing behavior. This research additionally does not encompass the use of fuzzers to generate YAML files that can potentially break parser libraries.

The approach of this research only checks whether YAML 1.2 parsing libraries can parse a YAML test case file or not, but it is not checked, whether the parsing library correctly interprets the YAML syntax. For example, a YAML directive declares the YAML version in which the YAML file shall be parsed. However, this research does not consider, whether a parsing library truly reads a YAML 1.1 file as a YAML 1.1 file.

Apart from generating further YAML test case files, future work could also analyze more YAML 1.2 parser libraries. For this research, parsing libraries are chosen that are listed on the official YAML website. But there are more YAML parsing libraries, such as *yaml-rust2*, which is a fork of the original *yaml-rust* repository,[39] or *yaml12* which is a YAML 1.2 parsing library for the programming language R.[40] Another option is to focus on YAML 1.1 parsing libraries, which are not touched in this research.

Finally, additional experiments could focus on trying to build a fully compliant YAML 1.2.2 parser library, that encompasses all features listed in the specification. Since all parser libraries, that are tested in this research, do not fully conform to the YAML 1.2.2 specification it might be subject to research upon whether it is even possible to create a YAML parsing library that is fully compliant with the specification.

Acronyms

AI – **Artificial Intelligence**

API – **Application Programming Interface**

BOM – **Byte Order Mark**: Specifies the used encoding for the YAML file

CPU – **Control Processing Unit**

CSS – **Cascading Style Sheets**: Language to customize the style of an HTML file

CSV – **Comma-separated values**

CVE – **Common Vulnerabilities and Exposures**

DOM – **Document Object Model**: The XML document is parsed and then placed into a tree structure. Standard way to interact with XML documents.

DoS – **Denial of Service**: An attack with the goal to make an application or host unavailable.

Flow Node: Nodes that have a start and an end character. Similar to JSON syntax.

flow sequence: An ordered list that starts with the charachter `[` and ends with the chracter `]` with zero or more content. Similar to JSON arrays.

GB – **Gigabyte**

GHC – **Glasgow Haskell Compiler**: Compiler for Haskell.

HTML – **Hypertext Markup Language**: Standard markup language for web pages

JSON – **JavaScript Object Notation**

KB – **Kilobyte**

MB – **Megabyte**

N/A – **Not Available**

PDF – **Portable Document Format**

RAM – **Random-Access Memory**

README: A file containing information about the current subset of YAML test files

SAX – **Simple API for XML**: Event-based parsing of XML documents.

SSH – **Secure Shell**: Network protocol for encrypted remote access

stderr – **Standard error**

stdout – **Standard output**

***SVG* – Scalable Vector Graphics**

***UTF* – Unicode Transformation Format**

***VM* – Virtual Machine**

***XML* – Extensible Markup Language**

***YAML* – Yaml Ain't Markup Language**: A human-friendly data serialization language

List of Figures

List of Tables

List of Listings

Bibliography

[1] O. Ben-Kiki, C. Evans, and I. döt Net, “YAML Ain’t Markup Language (YAML™) version 1.2 Revision 1.2.2.” Accessed: Nov. 03, 2025. [Online]. Available: https://yaml.org/spec/1.2.2/

[2] I. Predoaia, D. Kolovos, A. Garcia-Dominguez, M. Lenk, W. Ebel, and J. Burkl, “Towards Processing YAML Documents with Model Management Languages,” in *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, in MODELS Companion '24. Linz, Austria: Association for Computing Machinery, 2024, pp. 970–979. doi: 10.1145/3652620.3688219.

[3] S. Rasheed, J. Dietrich, and A. Tahir, “Laughter in the Wild: A Study Into DoS Vulnerabilities in YAML Libraries,” in *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, 2019, pp. 342–349. doi: 10.1109/TrustCom/BigDataSE.2019.00053.

[4] National Vulnerability Database (NVD), National Institute of Standards and Technology (NIST), “CVE-2022-41854 Detail.” Accessed: Jan. 12, 2026. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2022-41854

[5] National Vulnerability Database (NVD), National Institute of Standards and Technology (NIST), “CVE-2022-3064 Detail.” Accessed: Jan. 12, 2026. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2022-3064

[6] National Vulnerability Database (NVD), National Institute of Standards and Technology (NIST), “CVE-2019-6292 Detail.” Accessed: Jan. 12, 2026. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2019-6292

[7] The YAML Project, “The Official YAML Website.” Accessed: Dec. 06, 2025. [Online]. Available: https://yaml.org/

[8] N. Seriot, “Parsing JSON is a Minefield.” Accessed: Jan. 10, 2026. [Online]. Available: https://seriot.ch/software/parsing_json.html

[9] N. Harrand, T. Durieux, D. Broman, and B. Baudry, “The Behavioral Diversity of Java JSON Libraries,” in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, 2021, pp. 412–422. doi: 10.1109/ISSRE52982.2021.00050.

[10] J. Möller, F. Weißberg, L. Pirch, T. Eisenhofer, and K. Rieck, “Cross-Language Differential Testing of JSON Parsers,” in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, in ASIA CCS '24. Singapore, Singapore: Association for Computing Machinery, 2024, pp. 1117–1127. doi: 10.1145/3634737.3657003.

[11] S. C. Haw and G. S. V. R. K. Rao, “A Comparative Study and Benchmarking on XML Parsers,” in *The 9th International Conference on Advanced Communication Technology*, 2007, pp. 321–325. doi: 10.1109/ICACT.2007.358364.

[12] D. Wu, K. T. Chau, J. Wang, and C. Pan, “A comparative study on performance of XML parser APIs (DOM and SAX) in parsing efficiency,” in *Proceedings of the 3rd International Conference on Cryptography, Security and Privacy*, in ICCSP '19. Kuala Lumpur, Malaysia: Association for Computing Machinery, 2019, pp. 88–92. doi: 10.1145/3309074.3309124.

[13] W. Wei *et al.*, “An Extensive Study on Text Serialization Formats and Methods.” Accessed: Jan. 09, 2026. [Online]. Available: https://doi.org/10.48550/arXiv.2505.13478

[14] S. O. Bradner, “Key words for use in RFCs to Indicate Requirement Levels.” Accessed: Jan. 10, 2026. [Online]. Available: https://doi.org/10.17487/RFC2119

[15] O. Ben-Kiki, C. Evans, and B. Ingerson, “YAML Ain’t Markup Language (YAML™) version 1.0.” Accessed: Jan. 10, 2026. [Online]. Available: https://yaml.org/spec/1.0/

[16] O. Ben-Kiki, C. Evans, and I. döt Net, “YAML Ain’t Markup Language (YAML™) version 1.1.” Accessed: Jan. 10, 2026. [Online]. Available: https://yaml.org/spec/1.1/

[17] O. Ben-Kiki, C. Evans, and I. döt Net, “YAML Ain’t Markup Language (YAML™) version 1.2 Revision 1.2.1.” Accessed: Jan. 10, 2026. [Online]. Available: https://yaml.org/spec/1.2.1/

[18] H. K. Dhalla, “A Performance Analysis of Native JSON Parsers in Java, Python, MS.NET Core, JavaScript, and PHP,” in *2020 16th International Conference on Network and Service Management (CNSM)*, 2020, pp. 1–5. doi: 10.23919/CNSM50824.2020.9269101.

[19] The YAML Project, “YAML Test Suite.” Accessed: Jan. 11, 2026. [Online]. Available: https://github.com/yaml/yaml-test-suite

[20] T. Müller, “YAML Test Matrix.” Accessed: Jan. 11, 2026. [Online]. Available: https://github.com/perlpunk/yaml-test-matrix

[21] T. Müller, “YAML Test Matrix.” Accessed: Jan. 11, 2026. [Online]. Available: https://matrix.yaml.info/

[22] S. Rasheed, J. Dietrich, and A. Tahir, “Caught in the Web: DoS Vulnerabilities in Parsers for Structured Data,” in *Computer Security – ESORICS 2021*, E. Bertino, H. Shulman, and M. Waidner, Eds., Cham: Springer International Publishing, 2021, pp. 67–85. doi: 10.1007/978-3-030-88418-5_4.

[23] O. Ben-Kiki, C. Evans, and B. Ingerson, “Merge Key Language-Independent Type for YAML™Version 1.1.” Accessed: Jan. 11, 2026. [Online]. Available: https://yaml.org/type/merge.html

[24] The YAML Project, “yaml-grammar.” Accessed: Dec. 06, 2025. [Online]. Available: https://github.com/yaml/yaml-grammar/

[25] Symfony SAS, “The Yaml Component.” Accessed: Nov. 27, 2025. [Online]. Available: https://symfony.com/doc/current/components/yaml.html

[26] A. van der Neut, “ruamel.yaml.” Accessed: Nov. 27, 2025. [Online]. Available: https://yaml.dev/doc/ruamel.yaml/

[27] The YAML Project, “YAML Vocabulary Glossary.” Accessed: Dec. 25, 2025. [Online]. Available: https://yaml.org/spec/1.2.2/ext/glossary/

[28] snakeyaml, “SnakeYAML Engine 2.10 API.” Accessed: Dec. 27, 2025. [Online]. Available: https://javadoc.io/doc/org.snakeyaml/snakeyaml-engine/2.10/index.html

[29] Linux man-pages project, “time(1) — Linux manual page.” May 17, 2025. [Online]. Available: https://man7.org/linux/man-pages/man1/time.1.html

[30] A. Rumpf and Z. Karadjov, “Nim Manual.” Oct. 31, 2025. [Online]. Available: https://nim-lang.org/docs/syncio.html

[31] haskell-hvr, “HsYAML.” Accessed: Dec. 30, 2025. [Online]. Available: https://github.com/haskell-hvr/HsYAML

[32] O. Ben-Kiki, “YamlReference.” Accessed: Dec. 30, 2025. [Online]. Available: https://github.com/orenbenkiki/yamlreference

[33] D. Kästner and C. Ferdinand, “Proving the Absence of Stack Overflows,” in *Computer Safety, Reliability, and Security*, A. Bondavalli and F. Di Giandomenico, Eds., Cham: Springer International Publishing, 2014, pp. 202–213. doi: 10.1007/978-3-319-10506-2_14.

[34] A. Aubry, “YamlDotNet.” Accessed: Jan. 02, 2026. [Online]. Available: https://github.com/aaubry/YamlDotNet

[35] C. Yuheng, “yaml-rust.” Accessed: Jan. 02, 2026. [Online]. Available: https://github.com/chyh1990/yaml-rust

[36] haskell-hvr, “eo-yaml.” Accessed: Jan. 05, 2026. [Online]. Available: https://github.com/decorators-squad/eo-yaml

[37] A. van der Neut, “ruamel.yaml.” Accessed: Jan. 05, 2026. [Online]. Available: https://yaml.dev/doc/ruamel.yaml/

[38] A. Aubry, “YamlDotNet.” Accessed: Jan. 06, 2026. [Online]. Available: https://github.com/aaubry/YamlDotNet

[39] Ethiraric, “yaml-rust2.” Accessed: Jan. 06, 2025. [Online]. Available: https://github.com/Ethiraric/yaml-rust2

[40] T. Kalinowski, “yaml12: Fast 'YAML' 1.2 Parser and Formatter.” 2025. doi: 10.32614/CRAN.package.yaml12.