A Survey of Parser Differential Anti-Patterns

Sameed Ali
Department of Computer Science
Dartmouth College
Hanover, USA
sameed.ali.gr@dartmouth.edu

Sean W. Smith
Department of Computer Science
Dartmouth College
Hanover, USA
sws@cs.dartmouth.edu

***Abstract*—Parser differentials emerge when two (or more) parsers interpret the same input in different ways. Differences in parsing behavior are difficult to detect due to (1) challenges in abstracting out the parser from complex code-bases and (2) proving the equivalence of parsers. Parser differentials remain understudied as they are a novel unexpected bug resulting from the interaction of software components—sometimes even independent modules—which may individually appear bug-free.**

We present a survey of many known parser differentials and conduct a root-cause analysis of them. We do so with an aim to uncover insights on how we can best conceptualize the underlying causes of their emergence. In studying these differentials, we have isolated certain design anti-patterns that give rise to parser differentials in software systems. We show how these differentials do not fit nicely into the state-of-the-art model of parser differentials and thus propose improvements to it.

***Index Terms*—parsing, parser differentials, LangSec**

I. Introduction

Parser differentials emerge due to differences in parsing behavior of multiple parsers. Prior literature (e.g. [1]) credits the idea to the study by Kaminsky et al of X.509 certificates [2]. However, one can dig up earlier examples—e.g., eXtropia WebStore in 2000 [3] or IE in 2003 [4], and even argue that ancient SQL injection vulnerabilities qualify. Subsequent to the seminal X.509 work, parser differentials have since been discovered in many other places, such as HTTP (e.g. [5]), URLs (e.g. [6]), PDF (e.g. [7]) and various other formats (e.g. [8–10]). The continued emergence of parser differentials in a wide variety of software demands a thorough study of their underlying causes.

Although prior results have demonstrated the existence of parser differentials, a study to classify them has not yet been conducted. A study of the root-causes of known parser differentials will help illuminate the common anti-patterns that give rise to them. Additionally, it will provide insight for discovering advanced techniques to detect parser differentials in software systems.

In this study, we do not claim to present an *exhaustive* list of all known parser differentials. Instead, we present and categorize a selection of known bugs whose root-cause could be ascertained with confidence from the publicly available information about the vulnerability. Additionally, we have limited the number of parser differentials per category as our goal is to illustrate the underlying reason for the emergence of the parser differentials in that category rather than to list all known parser differentials that fall under it. Therefore, in our analysis, the parser differential vulnerabilities considered in this paper function as *representatives* of design anti-patterns that give rise to parser differential vulnerabilities. We have, thus, carried out an analysis of the selected parser differentials and isolated their root-cause to demonstrate common patterns of parser differential vulnerabilities. To our knowledge, this is the first contribution in the area of parser differentials that aims to carry out such an analysis for a broad range of known parser differentials and provides a categorization to aid our understanding of parser differentials further.

We hope such an analysis will prove beneficial to researchers developing automated tools for detecting parser differentials. In addition, we hope it will function as advice to software engineers and system architects to be aware of anti-patterns that rise to parser differentials.

To this end, we make the following contributions:

- We present a survey of known parser differentials.
- We conduct a root-cause analysis for the parser differentials under consideration to discover *why* they occurred.
- We conceptualize the root-causes and discover anti-patterns that give rise to parser differentials.
- We suggest improvements to the conceptual model of parse tree differential analysis in light of our research.

II. Background

Software developers and system designers often assume that input will be formatted correctly, and that parsers for the same input format will interpret the same input the same way. They then code as if differences in the interpretation of a file either do not exist—or if they do exist, then they do not impact the safety of the system in a significant way. This assumption was explicitly challenged by the discovery of PKI Layer Cake vulnerabilities [2], where the differences in interpretation between parsers was shown to be exploitable. Moreover, it was demonstrated that ambiguity in a specification could result in vulnerabilities even among standard-compliant parsers.

In terms of formal grammars and parsers, one might initially model parser differentials as follows:

- Parsers P_1 and P_2 both accept some input language L.
- For some valid input $w \in L$, P_1 and P_2 both accept w but build different parse trees.

Throughout the paper, we will revisit and revise this model.

2023 IEEE Security and Privacy Workshops (SPW) | 979-8-3503-1236-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/SPW59333.2023.00016

DOI 10.1109/SPW59333.2023.00016

These discoveries naturally led to research on determining the absence of variations in parsers. Determining the absence of variations required proving the equivalence of the input languages of the parsers under test. Standard results from formal language theory (e.g., the computer science view of the Chomsky Hierarchy) state that equivalence was only decidable for some grammars and not others (although it appears that decidability for deterministic context-free grammars (CFGs) was not established [11] until 1997). Sassaman et al [12], therefore, made a proposal for keeping the complexity of input languages within those bounds where the equivalence is decidable namely, deterministic context-free languages. They also proposed a novel method for discovering differences among parsers which they called a parse tree differential attack. This attack, illustrated in Figure 1, works by giving two parser implementations of the same format (or protocol) the same input data and then observing the differences in the generated parse trees. Differences in the generated parse trees are indicative of potential vulnerabilities.

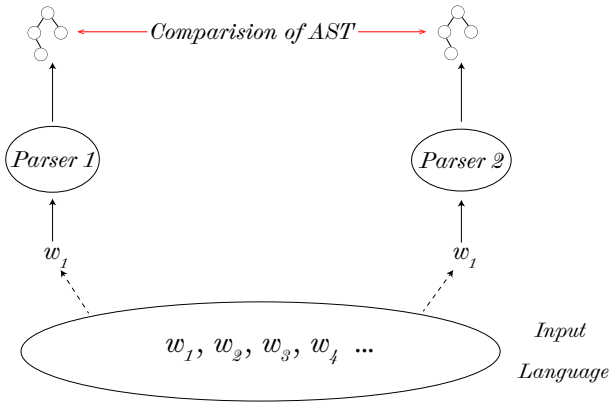


Fig. 1. A parse tree differential analysis involves comparing the parse trees generated from the parsers for differences.

III. Categories

This section categorizes the parser differentials according to their root-causes. We first note that a single parser differential can be caused by multiple factors. For example, the Zoom Stanza Smuggling vulnerability below arose because faulty parsing of the Unicode resulted in a higher-order parser differential in the XML parser. Therefore, when we list a parser differential under a category, we do not claim that it exclusively belongs to that category only. The aim of the categorization is to provide examples for an underlying anti-pattern that causes parser differentials so that it can be better understood. By focusing on one root-cause or anti-pattern at a time we aim to emphasise it in order to show how that particular pattern exhibits itself in multiple parser differentials.

Table I summarises the examples of parser differential we will consider. And figure 2 shows where the categories emerge in the software development process.

A. Programming Bugs: Overflow, Signed ints, off-by-one errors

Parser differentials caused by logic bugs in code or programming mistakes made by the programmers when writing the parser would fall in this category.

Example 1: Android Master Key Parser Differential 1: Freeman [13] documented a parser differential exploiting the Android Master Key bug. This parser differential occurred when the two parsers differed in parsing the local file headers of the APK archive. The local headers have a structure that is as follows: it begins with a fixed sized header, followed by a name field, followed by an extra header field, followed by the data. The extra header field is of variable length; its length is provided in the initial header field. The structure of this header (and an example exploit) is shown in Figure 3.

The Android OS has two parsers for APK files. One of them, written in C++, is responsible for file extraction. The second, written in Java, is responsible for verification.

If these two parsers differ on interpretation of the data bytes contained in the archive, the verifier would verify a different set of bytes from the ones extracted. An attacker can then exploit this difference to provide a benign set of bytes to be verified and a malicious set of bytes to be extracted.

The root cause of this parser differential was due to the verifier reading the length of the extra field erroneously. As mentioned earlier, this length value is parsed from the header. However, the verifier read those bytes as 16-bit signed integers rather than unsigned ones. Consequently, if a large enough value was provided as the extra field's length, the value read by the verifier would wrap around and become negative due to the two's complement based signed integer representation of Java. The verifier would, thus, miscalculate the start of the data bytes.

Figure 3 shows how the two parsers view the same file differently. It shows how if the value of the extra length field is read as negative three by the verifier, it results in the data bytes starting from a different position than that of the other parser.

This parser differential is a result of a logical bug/programming error. Parsing unsigned integer values as signed caused them to overflow which resulted in a miscalculated offset.

Example 2: Psychic Paper: In 2020, parser differentials among XML parsers in iOS lead to a zero day vulnerability. Security researcher Siguza, who discovered these vulnerabilities, notes [14] that an XML document with an invalid XML comment ("<!−−−>") was accepted by multiple parsers within iOS but was not interpreted by them in an identical way. The invalid XML comment was read by one parser as a start comment tag, by another as a start comment tag *followed by an* end comment tag. This difference causes one parser to treat parts of the XML as comments and ignore them while the second parser does not treat them as comments and consumes those parts.

More specifically, one parser after reading the "!−−" part of the invalid XML comment advances the pointer by three

TABLE I
PARSER DIFFERENTIALS AS DISCUSSED IN SECTION III

Name	Programming Bug	Delimiter	Higher-Order	Canonical	Ignored Semantics	Ambiguous Semantics	Human	Ambiguous Spec
1	✓							
2	✓							
3	✓		✓					
4		✓						
5		✓						
6		✓						
7	✓		✓					
8				✓				
9					✓			
10			✓					
11						✓		✓
12						✓		
13						✓		
14							✓	
15							✓	
16							✓	
17							✓	
18								✓

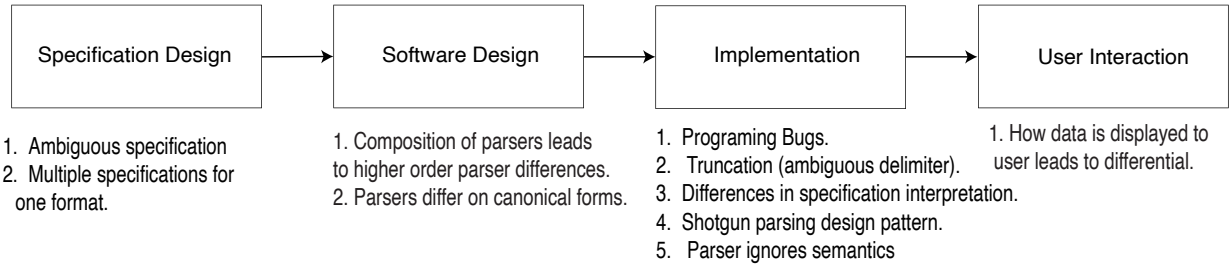


Fig. 2. This figure shows where in the software development process parser differential anti-patterns can come about. It shows the phase of software development where the root cause emerges: specification design: the process to writing down the specification of the protocol/format; software design: the process of defining the software architecture, and high level design of the software; implementation: writing out the code of the software; user interaction: The part when the software is in use and a user interacts with the software. The parser differential anti-patterns impact different stages of the software engineering process thus they are not mutually exclusive. This diagram lists the anti-patterns.

characters. It therefore does not read the remaining "−>" as a comment end. On the other hand, the second parser advances the pointer by two characters. It therefore reads the second "−" character twice and thus sees both "<!−−" and a "−−>" which leads it to believe the data represents a start and an end comment tag.

As iOS represents permissions in XML, these differences were quite useful to an attacker. These attacks were possible because the parser was more permissive in accepting an input than required by the specification.

Example 3: Multiple Common Name Fields in Certificates: The PKI Layer Cake work [2] also gives some instances of this anti-pattern, compounded with others we will discuss later.

Kaminsky et al discovered that parsers with integer overflow vulnerabilities could be exploited to create a source of disagreement among parsers about what constitutes a common name within an X.509 certificate. In its binary ASN.1 encoding an X.509 certificate's common name is an ASN.1 BER Sequence which consists of an Object Identifier (OID) followed by a String containing the name of the website. The OID is encoded as an unbounded integer. They found that if the value of an integer was large enough, then (for some consumers) it could overflow and get wrapped around. An attacker could, thus, exploit this bug by creating OID-String pair that some parsers would read as having an OID value identical to the common name leading to different interpretations for the common name.

Similarly, they found leading zeros in OIDs could be a source of disagreement among parsers. The leading zeros in an OID are ignored in some parsers and those OIDs are resolved to a common name whereas in other parsers OIDs with leading zeros are treated as a different OID.

They also reported that OpenSSL treats the leading zeros in an OID as significant when parsing but omits those in its

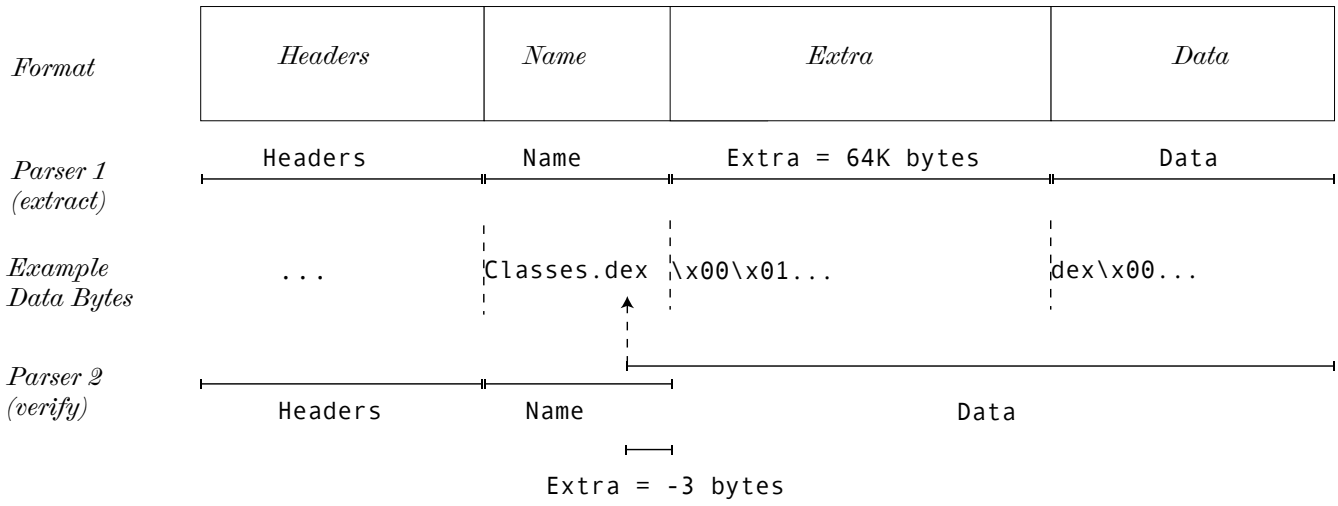


Fig. 3. This diagram shows a parser differential exploit for the Android Master Key vulnerability. Two parsers parse the same file differently, leading to differences between the data that is verified and the data that is extracted. The vulnerability occurs because Parser 2 misreads the length of the extra section as -3 which causes it to add -3 to the file pointer when it tries to calculate the start of the data section. This results in subtraction and hence Parser 2 thinks the data section starts from the third last character of the name bytes.

textual representation. Thus, OIDs with leading zeros—which OpenSSL internally treats as distinct—will appear as having the same OID when displayed. Consequently, an application using OpenSSL (via a command-line) to convert ASN.1 into an ASCII string representation, and then parsing the ASCII representation of ASN.1 to read the certificate would be susceptible to a parser differential attack.

B. Ambiguous Delimiters (Truncation)

Sometimes parsers disagree on the delimiters of fields in a format when parsing input data. If parsers differ on delimiters, then they may either prematurely terminate parsing and thus only parse a truncated input—or they may parse more data than they should.

That is: given input w,

- one parser acts correctly,
- but another sees w as some $w_1 w_2$ (for nonempty w_2) and parses only the prefix w_1 (e.g., due to some confusing characters).

Example 4: exTropia: In a vulnerability announced in 2000 [3], a parser differential in a web application resulted in the access of restricted web resources. The web application checked if the requested file was a valid html file by only checking if the string ended in the characters "html." If it did, the application considers it an html file and requested the file to returned. However, for a URL of the form

http://example.com/cgi-bin/Web_store/web_store.cgi?page=
../../../../../../../../etc/inetd.conf%00.html

something interesting happened. The web application (written in Perl) treated the "%00" character as just any other ordinary character and read the entire URL. However, when the open file call is issued the underlying language treats the "%00" as the end of string delimiter resulting in it returning the "inetd.conf" file—a forbidden resource.

Example 5: IE URL parsing: In an IE vulnerability from 2003 [4], one part of the browser parses the URL to figure out what site name to show to the user, and another parses it to figure out what to fetch. When given a crafted URL such as

http://www.trusted_site.com%01%00@malicious_site.com/
malicious.html

the parser responsible for displaying the URL would stop parsing when it encountered the URL-encoded null-byte "%00" whereas the parser responsible for fetching the URL resource read the entire URL. As a result, the browser informed the user that they were browsing "trusted_site.com" while they were actually visiting "malicious_site.com."

Example 6: PKI Layer Cake: Kaminksy et al [2] provide more examples of this pattern in a section entitled "Early null terminators." When parsing X.509 certificates. if null bytes are included in the domain name, then ambiguity over the string termination delimiter causes the certificate parsers to diverge. For instance, a CA may see www.bank.com[NULL].badguy.com as "badguy.com" and issue a cert for it. A victim's browser, going to badguy.com, would succeed in validating badguy.com against the certificate the CA issued for it. However, due to a parser differential, the browser will display "bank.com" as the validated website being visited to the user. (In some scenarios, yet another player is involved: parser confusion at the DNS with which the browser interacts.)

C. Higher Order Parser Differentials

Sassaman et al [12] introduced the concept of *order* of a parser differential. A *0th-order* differential involves only one protocol; a higher-order parser differential is a parser differential involving multiple protocols or data formats. A higher-order parser differential occurs when the transformation of data from one data format to another impacts the encoding or decoding of another one; Sassaman et al proposed a nicely behaved linear sequence.

Example 7: Zoom Stanza Smuggling: Zoom is a cross platform video-conferencing application. It has a built-in chat feature which allows the participants to send text-based messages to each other. Internally, the Zoom client application uses XML to encode the messages along with some metadata before sending it to the Zoom server. The Zoom server, after receiving this XML input from the client application, processes its contents and crafts an XML with the message embedded in it and forwards it to the recipient.

The overall software architecture of Zoom has multiple XML parsers and requires the XML parsers to have an identical understanding of the same XML to function correctly. This implicit assumption, regarding the parsing behaviour of its constituent parts, was demonstrated to be false and was a source of multiple vulnerabilities in the summer of 2022[15].

It was discovered that the zoom application allowed malicious adversaries to inject arbitrary data into the XML stanzas used to transfer the chat messages. This XML injection allowed a malicious adversary to inject XML into the input such that it was parsed differently by the server's XML parser and the receiving client's XML parser.

Exploiting these differences in the XML parsers allows an adversary to craft an XML message such that the receiving zoom client would interpret it as two separate XML messages. This attack was named "XML stanza smuggling" because it resulted in an XML stanza smuggled across the Zoom server to the receiving zoom client which interpreted the received XML as two XML stanzas instead of one. The zoom system architecture and the crafted XML document attack is shown in Figure 4.

The root cause of interpreting the XML stanzas as two messages by one XML parser and one message by another XML parser was a logic bug in the UTF-8 encoding of the XML characters. The UTF-8 encoding is a variable length character encoding. The starting bits of a UTF-8 encoded character encode how many bytes of data are going to encode that particular character. For instance, if the first byte of a UTF-8 character starts with 1110, then it is a three bytes long encoded character. Additionally, the UTF-8 spec *requires* the most significant bits of the remaining two bytes to start with 10.

In case of a crafted UTF-8 character where the first byte started with 1110—thus signaling a three byte UTF-8 character—but the remaining two bytes did not start with 10, the two XML parsers behaved differently. It is important to note that this byte is an invalid input as it does not confirm to the UTF-8 specification.

One XML parser interpreted it as a single UTF-8 character (which is three bytes in size) and thus consumes two bytes following the first byte as part of the same UTF-8 character. However, the other XML parser interprets it as three UTF-8 characters (each of which is a single byte in size). Consequently, if an XML tag boundary falls on such a ambiguous UTF-8 character, then one parser interprets it as a single XML tag and another interprets the XML as having multiple tags.

The root cause of this parser differential was due to a mistake in implementing the Unicode spec. Both parsers were accepting an *invalid* UTF-8 sequence. They, however, differed in how they interpreted this invalid sequence of UTF-8 bytes leading to this parser differential. Going through the bug fix for this vulnerability on GitHub [16], one notices a missing validation check to be the cause of this erroneous behavior.

Sassaman et al [12] would call this a *first-order* differential: a difference in UTF-8 parsing leads to a difference in XML parsing. However, we note there's an additional wrinkle here: the "higher-level" XML code at the server generates lower-level strings that preserve the buggy UTF-8, which are then sent on to the second XML parser; it's not just a simple one-direction channel.

We note that this example also differs from the simple model of Section II—the initial w here (the malformatted UTF-8) is actually not in the "correct" language. Both parsers are accepting strings beyond the correct language, and parsing things in this superset differently.

D. Differences in conversion to a canonical form

Differences in the conversion to a canonical form (or normalization) of a protocol (or file format) specification by software developers can lead to diverging parser behaviors. Conversion to a canonical or normal form is the standardization of an input by transforming it according to a set of predefined transformation rules in a consistent manner. For example, many web browsers convert the domain characters and URL scheme to lowercase before fetching a web resource. If the canonical form is modelled as a formal language, then it should be the identical in the different software systems processing it. Parser differentials may occur if they are not.

Example 8: TMUI RCE vulnerability: For instance, a parser differential was caused by the differing normalization of a URL in the TMUI RCE vulnerability (CVE-2020-5902) discovered in F5 networks [17], [18].

In this case, there were two http servers, Apache httpd and Apache Tomcat. The former was configured as a reverse proxy server whose job was to forward a subset of the received HTTP requests to the Tomcat server if the requests matched a certain criteria. This architecture is illustrated in Fig 5.

The vulnerability was due to differing URL normalization of Apache httpd server and the Apache Tomcat server. The difference lies in the parsing of the URL path parameters. Path segments of a URL are those parts of a URL that are delimited by slashes and a URL can have optional path

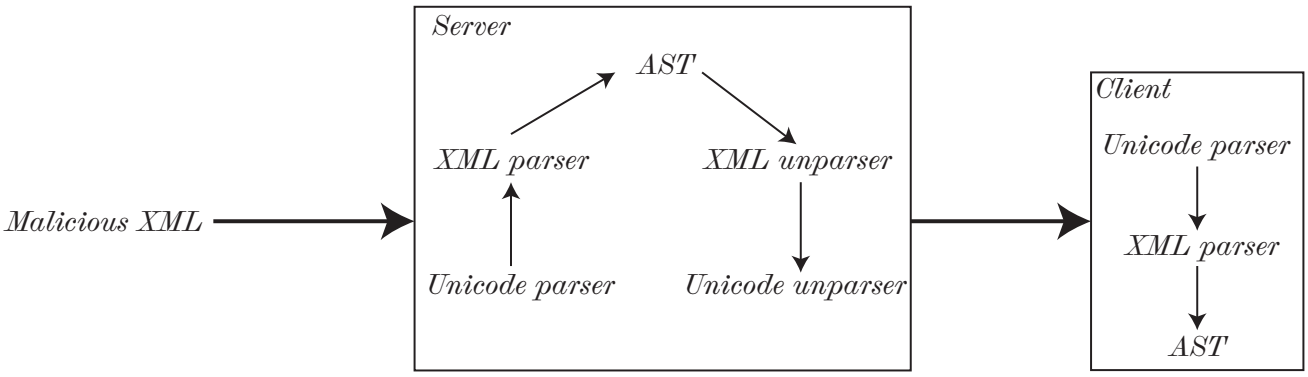
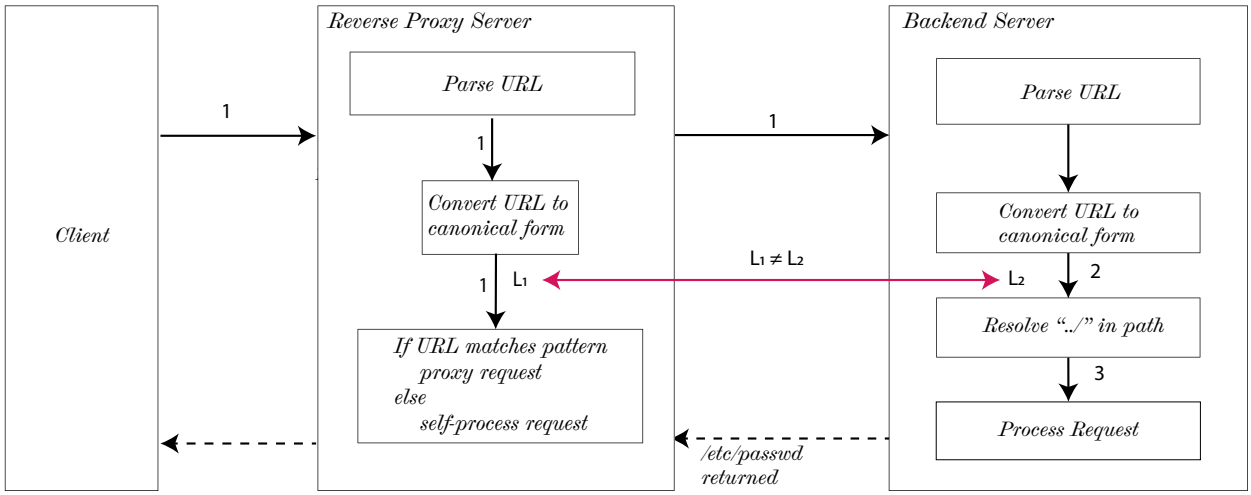


Fig. 4. This diagram illustrates the zoom video conferencing system architecture and the malicious XML sent by an attacker to the Zoom server. It shows the locations of the parsers inside the server and the client. It also shows how the XML and Unicode parsers interact with each other.



1. https://<IP>/tmui/login.jsp/..;/tmui/locallb/workspace/fileRead.jsp?fileName=/etc/passwd

2. https://<IP>/tmui/login.jsp/../tmui/locallb/workspace/fileRead.jsp?fileName=/etc/passwd
(colon removed as 1 gets transformed to 2. Resulting string in 2 has "../" which is a path!)

3. https://<IP>/tmui/tmui/locallb/workspace/fileRead.jsp?fileName=/etc/passwd

Fig. 5. This figure illustrates the parser differential which caused the TMUI F5 vulnerability (CVE-2020-5902). The reverse proxy server and the backend server both processed the URL into a canonical form but this form was not identical. Disagreement in understanding of URL led to the parser differential, as shown by the red arrow. The numbers show what the input URL looks like at various stages of the system.

parameters at the end of a path segment. Path parameters start after a semicolon character ";" and can contain non-slash, non-semicolon characters.

When the Apache httpd server parsed URL 1 shown in Fig 5 and converted it to a canonical form, it allowed the semicolon character ";" to remain in the http path. On the other hand, Apache Tomcat removed the URL contents from the ";" to the next forward slash when processing a URL because it considered these a path delimiter. After removing the semicolons, it normalized the path and executed the request. Fig 5 shows an overview of the inner workings of the parsers involved. Thus, in this case the parser differential resulted because of differing

notions of canonical forms conversion of the URLs between the reverse proxy and the backend server.

E. Semantic Parser Differentials: Ignoring Semantics

Semantic parser differentials occur when two parsers may produce the same tree, but then differ in their semantic interpretations of that tree.

Example 9: GitLab: HTTP reverse proxies are web servers which sit in front of the backend servers and relay HTTP traffic to the backend servers. They are often implemented to increase security, performance and reliability of a service.

These reverse proxies are often configured to act as request sanitizers—whose job is to only allow certain types of HTTP requests to the backend server. They achieve this by rewriting the HTTP requests as they proxy them to the backend server.

If parsing differences exist between the HTTP request parser of the proxy server and the backend server, then the reverse proxy may allow those HTTP requests that it considers benign to pass through but which result in malicious behaviour when the HTTP backend server processes them.

This was the case for the file upload GitLab security vulnerability [19]. The GitLab reverse proxy was configured to rewrite all PUT requests to the backend server. Thus, under normal system operating conditions PUT requests were not allowed to go through from the reverse proxy to the backend server without being processed first.

To understand how the vulnerability came about one needs to mention that some HTTP servers provide a way to override the HTTP method of a request by passing custom headers in the request. These custom headers specify the HTTP method name which takes precedence over the HTTP method specified at the beginning of the HTTP request. This additional functionality exists in some web frameworks to allow them to bypass restrictive web application firewalls [20]. Other times this functionality is added for compatibility reasons: some existing HTTP libraries, for example, do not allow the creation of HTTP methods other than GET and PUT. Thus, in those situations an alternative way to specify the additional HTTP methods is needed. This technique is referred to as "verb tunneling".

The GitLab differential was due to the reverse proxy HTTP parser ignoring the HTTP method override header and the backend's HTTP parser not doing so. The reverse proxy in this case was written in Go. The backend server was written in the Ruby programming language and used the Ruby On Rails web framework which supported the HTTP method override functionality. When a PUSH request with a HTTP method override header—asking the request to be considered a PUT request — was received by the reverse proxy, it was interpreted as an ordinary PUSH request and forwarded unmodified. The backend server upon receiving this request interpreted it as a PUT request and processed it. Thus a crafted HTTP request was able to bypass the request rewriting of the reverse proxy.

In this example, the differences in the parsers are not due to misidentifying the correct delimiter or some other such syntactical parsing difference. It is, instead, a semantic difference.

The simple model of parser differentials fails to capture such semantic differentials. Parser differentials of this kind would not be detected by checking if the parse trees generated by the parsers are identical. Furthermore, such examples show that merely looking at the Chomsky hierarchy is not enough. We need to consider semantics of a grammar to prevent parser differentials.

F. Semantic Parser Differentials: Semantic Ambiguities (Duplicate Keys)

Semantically ambiguous parser differentials are caused by an input which is syntactically valid but is semantically ambiguous. A semantic ambiguity could result in parsers interpreting the same input differently and generating differing parse trees from it.

For instance, in our analysis of known parser differentials, we came across parser differentials where the input language was designed to express key-value pairs. The parsers processing these input languages assumed the uniqueness of keys when processing the input. It was, however, possible to construct syntactically valid inputs which had duplicate keys. These inputs were syntactically valid but semantically ambiguous. When faced with input with duplicate entries (i.e. multiple entries with the same key value), a parser has to chose an entry among the duplicates which will provide the value for the key under consideration resulting in a semantic ambiguity. A parser differential results when different parsers chose different entries and the selected entries have different values.

The occurrence of this anti-pattern in wide variety of areas suggests a common anti-pattern of parser differentials.

Example 10: Android Master Key Vulnerability 2: The Android APK format—the file format used by Android applications—follows the ZIP format description. The ZIP format allows defining two files with the same name in a ZIP archive but does not specify which should be used in case of duplicates. In the case of a maliciously crafted APK, which has two files of the same name, Android OS verifies the signature the first file of that name, whereas the Package Installer would use the second file of that name for installation. As a consequence, an attacker is able to install unsigned code by placing the first file with a valid signature and the second file with the malicious code in it.

A root-cause analysis of the bug reveals that it was due a difference in the hash-table data structure used to store the parsed data. In one parser, which was implemented in Java, a hashmap was used to store the parsed data. When multiple items of the same key were added to the hashmap, the hashmap would store the *last* entry and ignored the previous entries.

On the other hand, in the second parser, which was implemented in C, the parser used a hash table with linear probing. This data structure, returned the *first* entry when multiple entries of the same key were added to it.

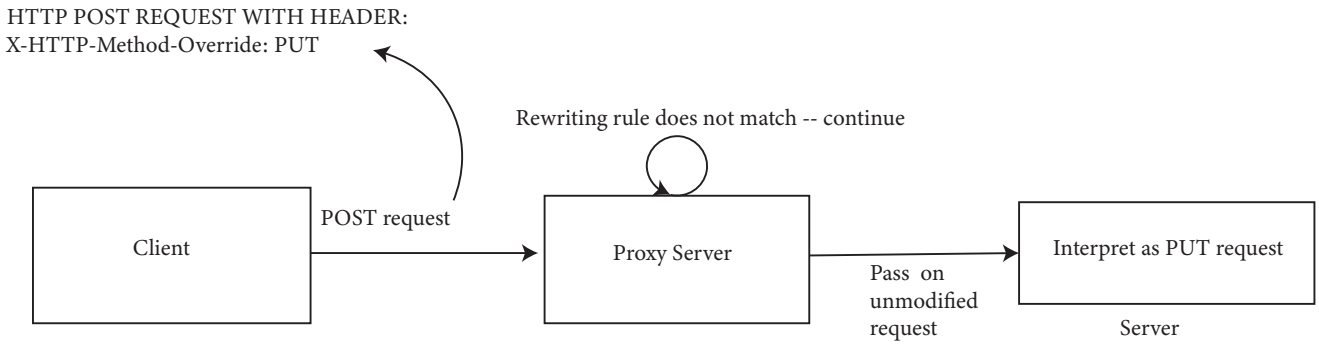


Fig. 6. This figure shows how the GitLab parser differential works. A post request with a method override is forwarded by the proxy server because it thinks it is a post request but the backend server interprets it as a put request.

Thus, when a ZIP file having multiple files with the same name in it was parsed by the Java parser, the last file of the same name was considered by the parser. However, in the C parser, the first file of that name was considered. Consequently, the interpretation of the ZIP archive differed between the two parsers resulting in the vulnerability.

The parsers made the implicit assumption that a ZIP archive will not have duplicates, but that was not the case as it was allowed by the ZIP format. Thus we note this is another example where a simple linear order for differentials does not quite work: another view of a root cause here is one protocol (where key-values should be unique) borrowing a format from another protocol (which permits duplicates) — a phenomenon we suggest calling a "side-order" differential.

Example 11: Duplicate Keys in JSON:

JSON is a lightweight data-interchange format, widely used on the internet to transfer data. The data format consists of key-value pairs and arrays. In this data format it is possible to construct inputs which are syntactically valid but semantically ambiguous by defining multiple keys of the same name with different values. The JSON specification, according to Section 4 of RFC 8259 [21], states that the names within an object *should* be unique. However, it does not *require* them to be so. The RFC, additionally, requires that all inputs conforming to the specification be accepted.

When JSON parsers are given JSON with duplicate keys having non-identical values as input to parse, their parse results are often not identical to other JSON parsers. Parsers have to prefer one entry among the duplicate ones as the authoritative entry. Often, parsers do not give precedence to the same entry which leads to parser differentials. Prior security research [10] has documented the existence of such parser differentials.

In this example, we note the ambiguity which is the cause of the parser differential stems from the format specification itself. By considering duplicate keys in JSON as valid, not specifying a criteria for precedence among them and requiring parsers to parse all valid JSON, the specification leaves enough details ambiguous and unclear that causes parser differentials to emerge. Further, parsers which enforce additional semantic conditions for a successful parse — such as rejecting JSON objects with duplicates or rejecting JSON with integer values in a field which are out of a semantically meaningful range — can also be susceptible to semantic parser differentials if all parsers in the system do not have the same criterion of accepting or rejecting JSON objects.

Example 12: Duplicate HTTP parameters: Consider the following HTTP URL

http://www.example.com/search?foo=1&foo=123

From the URL it is not clear what value should the parameter "foo" take? Should it be 1 or 123 or both?

As early as 2009 [22], researchers have pointed out that HTTP requests with multiple parameters having the same name results in differing parsing behavior among parsers of known HTTP servers.

The Go language's HTTP package says in its documentation that it will return the *first* parameter value [23] it sees (i.e. foo is 1). However, in Django (a python web framework) multiple values are stored in a dictionary, which returns the *last* value (foo = 123). Django implementations save the parsed values in a QueryDict structure This structure is designed to handle multiple values and this has been mentioned in the documentation [24].

In this case, the parser differential is due to an implementation difference because of the type of data structure used. However, this difference is intentional as the expected behavior of the data structures is noted in the source code comments and the documentation of the framework themselves. Nevertheless, if the system designer overlooks this difference would result in a vulnerable system.

Example 13: Email Sender Spoofing: A similar pattern appears in emails sent with multiple `From:` header entries. Shen et al [25] report that if multiple `From:` headers are provided when sending an email, some mail user agents verify one email address via DMARC but display the other one to the user. This parser differential may allow an attacker to spoof their email address.

Other Examples: The seminal PKI Layer Cake paper [2] documents many instances of trouble (some building on Example 3 above) due to differing interpretations of an X.509 item with multiple common names.

G. Parser Differentials Involving Human Interaction

Earlier in this paper we argued that parser differentials occur when two computers differ in their understanding of the same data. A similar situation may also occur when humans and computer interact with the same data and differ in their understanding of what that data means.

In an adversarial scenario, these parser differentials can be defined as those that occur when an attacker sends crafted input to an application such that when the application parses the input and displays it renders the input in manner that puts the user at risk of interpreting the displayed data differently than how the computers understands it.

Example 14: Information in Invalid Certificates: In ancient times, one such example was when a browser encountered an invalid or unsigned SSL certificate and warned the user. However, the warning repeated information from the unverified certificate—so if the adversary crafted a fake certificate from "Trustworthy Corporation," the user was told the the browser "could not verify this certificate from Trustworthy Corporation," and the user would conclude it was OK.

Example 15: PDF malware campaign: A more recent example is the PDF malware campaign reported by HP in May 2022 [26]. The attacker crafted a malicious PDF with an embedded docx file. However, the attacker sneakily named the file

has been verified. However PDF, Jpeg, xlsx

When the PDF reader then displays a prompt to confirm from the user whether this is trustworthy, it appears as if the filename was part of the text from the reader, not the filename. Fig 7 shows the window. The user is at risk of parsing the displayed sentence in a way that is at odds with how the computer understands the displayed message. The user is, hence, at risk of unknowingly accept the prompt and opening the embedded file.

Interestingly, in the attack, the prompt is not grammatical—we have yet another case of a parser—the human—"helpfully" correcting the input.

Example 16: Unicode Trojan Source: Boucher et al [27] have shown how the bidirectionality of Unicode characters can be utilized to create source code which looks innocuous to a programmer reading the displayed text but is actually code instructing the compiler to do something different.

The attack, which also referred to as the Unicode Trojan source code attack, is another interesting example of a differential where the user's understanding of what the computer is presenting differs from how the computer understands the presented data.

Example 17: PDF Shadow Attack: The Shadow attacks [7] are yet another example of a parser differential that falls in this category. The attacks involve modifying a signed PDF document with an incremental update. The PDF viewer tells the user that the document is signed. However this can be interpreted in two ways: (1) the document was signed then updated or (2) the document was updated then signed. The user cannot easily distinguish between the two and they might erroneously interpret the displayed document as (2) whereas the computer understands (1).

H. Ambiguous Specifications

The simple model from Section II implicitly assumed that all parsers (and their developers) were targeting a clearly defined language. However, this is not all the case. As our discussion in Example 11shows ambiguity of the specification can lead to parser differentials.

Example 18: Underscores in Email Addresses: Some email parsers do not allow the underscore character ("_") in the host name part of an email [28], but others do (such as Python's standard library's email parser). Going through discussions of developers online [29], [30], one finds confusion and disagreement among the developers regarding the use of underscore in an email addresses host name. The root of the confusion lies in the difference in a domain name and the host name. A domain name is a identifier of a resource on a DNS database but a host name is a special type of domain name which identifies internet hosts. Domain names allow underscores but host names do not. The RFC2181 [31] makes a distinction between a domain name and a host name. A host name must follow all rules for a domain name but in addition to those rules it should also satisfy the additional requirements of a host name. Whereas, a domain name only needs to satisfy the domain name requirements. Thus, when writing a parser for email, one needs to be aware of three specifications: the email specification, the domain name specification and the host name specification. When developers have to follow multiple specifications for a format then the clarity of a specification decreases. An otherwise clear specification becomes ambiguous in practice because many people misinterpret or misunderstand it.

Other Examples: Example 9 above (GitLab) also manifests this problem. The HTTP 1.1 spec [32] itself does not mention verb tunneling or method override. However, this functionality is a part of numerous HTTP web frameworks. This divergence from the specification for practicality is another reason that caused that parser differential to emerge.

IV. Revisiting the Model

In light of these examples, we now revisit the thinking of Section II.

A. Hunting Differentials in Practice

Looking at programs in the wild leads us to emphasize: discovering parser differentials in practice is hard. Not only

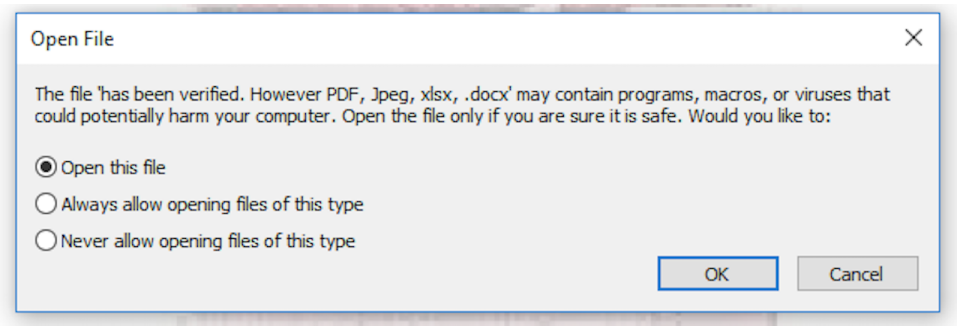


Fig. 7. This figure shows the prompt generated by the crafted malicious PDF file. The prompt is deliberately crafted to easily be misread by the end user. It achieves this by crafting a phrase which changes the meaning of the sentence presented by the prompt if careful attention is not paid when reading the prompt. This image is from [26].

do clearly defined abstract syntax trees often not exist in code; but parsers also often carry out "shotgun parsing" [33] which is a coding pattern where parsing and input validation code is spread across code that processes the input. Additionally, programs in the wild are under no obligation to use standard parsing algorithms for parsing the input (and typically do not do so). Instead, they often parse input in unexpected and unintuitive ways. For instance, in one parser we observed the command line program `grep` was used to search for a keyword in the given input for the sake of parsing the input. Therefore, carrying out a parser differential attack as described in Section II is ultimately limited in practice because software do not neatly convert the given input into clearly defined abstract syntax trees which can then be compared and contrasted with each other. Prior research [5] has used grammar based differential fuzzing to search for parser differentials. These approaches have been successful in discovering many parser differentials. However, fuzzers do not ensure they have discovered all parser differentials in a systems. Moreover, fuzzers too rely on our understanding of how a parser differential operates, and if our conceptual understanding of how a parser differential behaves is limited, then it impacts the fuzzers we construct to discover them. For instance, in the prior work only those differentials are noticed which have different parse trees, whereas we show how intermediate representations, and data structure differences (that may go unnoticed by a fuzzer) can also produce differentials. Therefore, to build better automated discovery tools, we first need to better comprehend how parser differentials operate. Prior work by Sassaman et al [12] provides a beneficial abstract model for understanding the assumptions made by software systems and how they should operate in an ideal scenario. Therefore, we suggest improvements to that model for the sake of better conceptual clarity in light of the parser differentials discussed in the sections above.

B. The Basic Language Model

Looking at real examples, we see the characterization "for some valid input $w \in L$, P_1 and P_2 both accept w but build different parse trees" is insufficient. Often, the w is outside the language being targeted—the parsers accept that which they should reject. Often, the parsers themselves have been designed (implicitly or explicitly) for different languages; that, combined with overly liberal acceptance, leads to lots of combinations of where the crafted w should lie.

C. Orders

Prior thinking suggested a simple linear order for differentials: if we have $n > 1$ levels of protocols and a differential in parsing protocol P_1 leads to a differential in P_n, then we have a differential of order $n - 1$.

However, we see that the flow can be more complicated than that. A P_1 problem can go to P_2, then back to P_1, then on to a second consumer of P_2, where the difference is finally manifested. For both protocol formats and specifications, we also see side-order differentials: a differential arises because one instance borrows structure from a peer instance.

D. Differentials and Decidability

As the literature observes, we might view testing if two parsers do the same thing as testing if they accept the same language. If we view the parser as a grammar, this means testing the equivalence of two grammars. As noted earlier, standard formal language theory tells us that such testing is undecidable for non-determinstic CFGs and higher, but decidable for deterministic CFGs and lower.

However, in some prior literature [11], [34],

- *weak equivalence* is when two grammars accept the same language
- *structural equivalence* (sometimes called *strong* equivalence) is when two grammars not only accept the same language, but also generate congruent parse trees for each accepted string

The standard textbook results, rephrased in terms of these definitions, only refer to weak equivalence. However, for parser differentials, one can argue that what we really care about is strong equivalence. Furthermore, the literature for strong equivalence [34] hides a surprising result: strong equivalence of CFGs is decidable—even of non-deterministic CFGs! This work goes further: suppose we define the *parenthesized* version of a CFG by

- adding two new non-terminals [and]

- then replacing each rule $A \longrightarrow w$ with $A \longrightarrow [w]$

Then two CFGs are strongly equivalent exactly when their parenthesized versions are weakly equivalent. Basically, the parentheses embody the parse tree structure. The more recent Reghizzi et al monograph [11] goes even further and observes: parenthesized CFGs are deterministic. Together, this suggests a decision procedure for parser differentials of the same format. Consider the CFGs G_1 and G_2 which represent the same format and are, hence, nearly identical albeit with minor differences. We can parenthesize them (and get deterministic CFGs), and then test for weak equivalence. If weak equivalence holds, then G_1 and G_2 accept the same things with the same parse trees. If weak equivalence does not hold, then we have at least one of the following:

- $L(G_1) \neq L(G_2)$
- there exists some w which G_1 and G_2 both accept, but parse differently.

Using these strong-equivalence results and applying them towards parser differential analysis would be an interesting area for future work — although likely made messier by the complications from Section IV-B above.

E. Semantics

Formal grammars which define the Chomsky hierarchy (i.e. regular, context free, context sensitive and recursively enumerable grammars) concern themselves with the syntax of a formal language and not its semantics. However, the GitLab parser differential (Subsection III-E) shows us parser differentials can occur due semantic differences in parsing a message. If one were to look merely for syntactical differences in parsing of that malicious HTTP request one would not find them. It is the semantics of the header that change the method of the request. Similarly, the examples discussed in Subsection III-F provide additional evidence that parser differentials can be caused by inputs that are syntactically valid but semantically ambiguous. In those cases it was the semantic ambiguity of duplicate key-values that caused the parsers to differ. Therefore, we suggest that one should consider modeling program input with grammars that are semantically aware. Attribute grammars, for instance, allow us to supplement a formal grammar with semantic information. Such grammars are able to capture the semantic aspects of the parser. Thus allowing semantic parser differentials to become a part of our analysis.

F. Intermediate languages in programs

Programs sometimes transform the input into an intermediate form before processing it. As shown in Subsection III-D, the differences in this conversion can also lead to parser differentials. In this case, we have a difference in the intermediate language generated from the input: the normalized URL. Thus, we suggest to consider not only the input and the abstract syntax tree of the input in our model but also consider the intermediate languages that systems implicitly assume programs will have an identical understanding of.

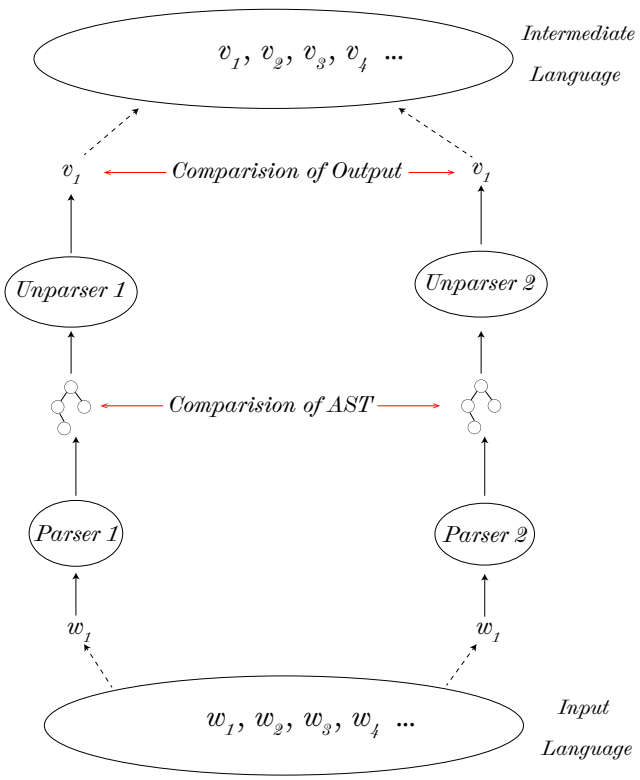


Fig. 8. This figure shows how we can include the transformation of the input into an intermediate language into account in our model. In addition to comparing the AST, one needs to compare the generated intermediate language word to ensure both models are consistent.

This would involve extending the model to include the conversion of the abstract syntax tree to the intermediate language in our analysis. This means in addition to comparing the abstract syntax tree, we also need to compare the intermediate language generated when an input is given to a parser to ensure consistency. This is illustrated in Figure 8.

V. Conclusion

We have shown how the various parser differentials do not fit nicely into the parser differential model proposed by Sassaman et al [12], and we propose improvements on the model.

Acknowledgment

This material is based in part upon work supported by the Defense Advanced Research Projects Agency (DARPA) under contracts HR001119C0075 and HR001119C0121. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

References

[1] L. Sassaman, M. L. Patterson, S. Bratus, and A. Shubina, "The halting problems of network stack insecurity," *USENIX; login*, vol. 36, no. 6, pp. 22–32, 2011.

[2] D. Kaminsky, M. L. Patterson, and L. Sassaman, "Pki layer cake: New collision attacks against the global x. 509 infrastructure," in *International Conference on Financial Cryptography and Data Security*. Springer, 2010, pp. 289–303.

[3] f0bic, "Security advisory: extropia webstore directory traversal vulnerability," https://seclists.org/bugtraq/2000/Oct/134, 2000.

[4] Secunia Advisories, "Internet explorer url spoofing vulnerability," https://web.archive.org/web/20041209035254/http://secunia.com/advisories/10395/, 2003.

[5] B. Jabiyev, S. Sprecher, K. Onarlioglu, and E. Kirda, "T-reqs: Http request smuggling with differential fuzzing," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 1805–1820.

[6] J. Reynolds, A. Bates, and M. Bailey, "Equivocal urls: Understanding thenbsp;fragmented space ofnbsp;url parser implementations," in *Computer Security – ESORICS 2022: 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26–30, 2022, Proceedings, Part III*. Berlin, Heidelberg: Springer-Verlag, 2022, p. 166–185. [Online]. Available: https://doi.org/10.1007/978-3-031-17143-7_9

[7] S. Ali, P. Anantharaman, Z. Lucas, and S. W. Smith, "What we have here is failure to validate: Summer of langsec," *IEEE Security & Privacy*, vol. 19, no. 3, pp. 17–23, 2021.

[8] James Kettle, "Http desync attacks: Request smuggling reborn," https://portswigger.net/research/http-desync-attacks-request-smuggling-reborn, 2019.

[9] David Buchanan, "Png parser differential," https://www.da.vidbuchanan.co.uk/widgets/pngdiff/, 2021.

[10] Jake Miller, "An exploration of json interoperability vulnerabilities," https://bishopfox.com/blog/json-interoperability-vulnerabilities, 2021.

[11] Reghizzi, Stefano Crespi and Breveglieri, Luca and Morzenti, Angelo, *Formal languages and compilation*. Springer, 2013.

[12] L. Sassaman, M. L. Patterson, S. Bratus, and M. E. Locasto, "Security applications of formal language theory," *IEEE Systems Journal*, vol. 7, no. 3, pp. 489–500, Sep. 2013.

[13] Jay Freeman, "Android bug superior to master key," http://www.saurik.com/id/18, 2023.

[14] Siguza, ""psychic paper"," https://blog.siguza.net/psychicpaper/, 2020.

[15] Ivan Fratric, "Zoom stanza smugglling or how i hacked zoom," https://www.blackhat.com/us-22/briefings/schedule/#xmpp-stanza-smuggling-or-how-i-hacked-zoom-26618, 2022.

[16] hartworkhartwork, "[CVE-2022-25235] lib: Protect against malformed encoding (e.g. malformed UTF-8) #562," https://github.com/libexpat/libexpat/pull/562/commits/3f0a0cb644438d4d8e3294cd0b1245d0edb0c6c6, 2022.

[17] Ollie Whitehouse, "Understanding the root cause of F5 Networks K52145254," https://research.nccgroup.com/2020/07/12/understanding-the-root-cause-of-f5-networks-k52145254-tmui-rce-vulnerability-cve-2020-5902/, 2020.

[18] Mitre, "Cve-2020-5902," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-5902, 2020.

[19] Joern Schneeweisz, "How to exploit parser differentials," https://about.gitlab.com/blog/2020/03/30/how-to-exploit-parser-differentials/, 2022.

[20] Microsoft, "X-http-method in ms-odata protocol," https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-odata/bdbabfa6-8c4a-4741-85a9-8d93ffd66c41?redirectedfrom=MSDN, 2019.

[21] Internet Engineering Task Force, "The javascript object notation data interchange format," https://www.rfc-editor.org/rfc/rfc8259, 2017.

[22] Luca Carettoni, "Http parameter pollution," https://owasp.org/www-pdf-archive/AppsecEU09_CarettoniDiPaola_v0.8.pdf, 2009.

[23] Google Go Authors, "Go Package source code," https://cs.opensource.google/go/go/+/master:src/net/url/url.go;drc=0765da5884adcc8b744979303a36a27092d8fc51;l=885, 1999.

[24] Django Documentation authors, "Django documentation," https://docs.djangoproject.com/en/4.1/ref/request-response/#django.http.QueryDict.

[25] K. Shen, C. Wang, M. Guo, X. Zheng, C. Lu, B. Liu, Y. Zhao, S. Hao, H. Duan, Q. Pan *et al.*, "Weak links in authentication chains: A large-scale analysis of email sender spoofing attacks," *arXiv preprint arXiv:2011.08420*, 2020.

[26] Patrick Schläpfer, "Pdf malware is not yet dead," https://threatresearch.ext.hp.com/pdf-malware-is-not-yet-dead/, 2022.

[27] N. Boucher and R. Anderson, "Trojan source: Invisible vulnerabilities," *arXiv preprint arXiv:2111.00169*, 2021.

[28] salesforce fourm, "Allow underscores in email domain fields," https://ideas.salesforce.com/s/idea/a0B8W00000Gdf2gUAB/allow-underscores-in-email-domain-fields.

[29] StackOverflow, "Can a subdomain have underscore in it?" https://stackoverflow.com/questions/2180465/can-domain-name-subdomains-have-an-underscore-in-it.

[30] google groups, "Underscores in Domain Names," https://groups.google.com/g/comp.protocols.dns.bind/c/kxJQspiOE8E.

[31] Network Working Group, "Clarifications to the DNS Specification," https://www.rfc-editor.org/rfc/rfc2181#section-11, 1997.

[32] The Internet Society, "Hypertext transfer protocol," https://www.rfc-editor.org/rfc/rfc2616, 1999.

[33] F. Momot, S. Bratus, S. M. Hallberg, and M. L. Patterson, "The seven turrets of babel: A taxonomy of langsec errors and how to expunge them," in *2016 IEEE Cybersecurity Development (SecDev)*, 2016, pp. 45–52.

[34] M. C. Paull and S. H. Unger, "Structural equivalence of context-free grammars," *Journal of Computer and System Sciences*, vol. 2, no. 4, pp. 427–463, 1968. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0022000068800376